Bachelor thesis:

Space-time discontinuous Galerkin methods for the wave equation

Anders Skajaa

Student at the Technical University of Denmark
Exchange student at ETH-Zürich

Ultimo June, 2007

Supervised by:

Prof. Dr. Ralf Hiptmair
Seminar of Applied Mathematics, ETH-Zürich

Abstract

We attempt to construct an explicit method for solving the scalar wave equation based on interior penalty discontinuous finite element methods (IP-DG FEM). The method is explicit in the sense that the solution can be developed one element at a time. We derive the method formally and show how to construct a Matlab program to run the method on a tensor product mesh with vertical edges and bilinear elements. Unfortunately numerical experiments give evidence that the method is unstable for any choice of parameters. This is further investigated in a von Neumann stability analysis which results in strong (numerical) evidence that the method is unconditionally unstable - at least for this certain type of trial-/testspace.

A possible remedy could be to use a mesh that avoids vertical edges (except on the boundary). This is required in somewhat similar, but successful (non-IP) DG-methods, [1, 2, 3].

---

0 Preface

This report is the written result of a bachelor thesis carried out during the summer term of 2007, the second half of the authors exchange year at ETH-Zürich, Switzerland.

The following tasks were given:

1. Derivation of interior penalty DG equations for the wave equation

$$\frac{\partial^2 u}{\partial t^2} - \frac{\partial^2 u}{\partial x^2} = f(x,t)$$

2. Investigation of stability using linear stability analysis for different values of stabilization parameter.

3. Numerical experiments to study the convergence of the method, if conditional stability can be achieved

The project is a combination of theory (the first point) and implementation (the remaining two), the dominating part being the latter. The main portion of the time used *was* indeed spent implementing the method in MATLAB and Maple. The extend of the project is 15 ECTS points.

The project was supervised by Prof. Dr. Ralf Hiptmair whom the author thanks for his efforts.

Contents

0 Preface **2**

1 Introduction **4**

2 Problem and method **5**
- 2.1 The wave equation 5
- 2.2 Notation 5
- 2.3 Formulation of the method 6
- 2.4 Calculation in practice 7

3 Implementation **9**
- 3.1 Basis functions and mesh 9
- 3.2 Structure of code 10
- 3.3 Numerical experiments: Instability 12

4 Stability **13**
- 4.1 Theory 13
- 4.2 Numerical investigation 14

5 Another approach: G. Richter **16**
- 5.1 The method 16
- 5.2 Mesh requirements 17
- 5.3 Comparison 17

6 Concluding remarks **18**

References **19**

Appendix **20**

A Code: Matlab **20**
- A.1 `evolve.m` 20
- A.2 `calc_new_slab.m` 20
- A.3 `calc_one_elem.m` 20
- A.4 `calc_ICs.m` 21
- A.5 `defmatrices.m` 21
- A.6 `conscheck.m` 22
- A.7 `plotsolution.m` 22

B Code: Maple **24**
- B.1 `calc_elem_matrices.mws` 24

C Proof of the DG Magic Formula **27**

1 INTRODUCTION

The scalar wave equation is used for the simulation of many physical problems within electromagnetism, elastics and acoutics. Many numerical solution methods exist, the most dominant being some kind of spatial discretization followed by a timestepping method. Among these, using a discontinuous Galerkin finite element method (DG-FEM) as the spatial discretization gives good results, see e.g. [4]. One problem with these methods, however, is that they impose a global CFL condition, which makes local mesh-refinement difficult.

One way to avoid this problem is to instead treat space and time as equivalent variables. We then partition the entire space-time domain into a mesh and then apply some kind of FE-method. In 1990 Hulbert and Hughes presented a semi-discontinuous space-time method for second order hyperbolic equations, [5] - semidiscontinuous because they use functions that are continuous in space, but discontinuous in time.

An (almost) fully discontinuous space-time method for the wave equation was first introduced by Richter in 1994 in [1], a method he generalized first in 1999 in [2] with Falk and again in 2003 with Monk in [3]. In the latter article a fully discontinuous explicit space-time method for solving linear symmetric hyperbolic systems in possibly inhomogeneous media is devised and proved to be stable.

In this thesis we shall attempt something similar (though less general) as Richter et. al, but now using interior penalizing to enforce continuity, yielding an IP-DG-FEM. These methods were first introduced in the 1970's, originally motivated by Nitsche's idea to *weakly* enforce homogeneous boundary conditions on the elliptic problem $-\Delta u = f$. Similarly continuity of the solution is enforced weakly on all edges by penalizing jumps of the discontinuous test functions, first introduced in [6].

So far it seems this approach has only been used for elliptic and parabolic problems and in this thesis we will try to apply the IP-DG-FEM to the scalar wave equation on a tensor product mesh. In section 2 we introduce the required notation and formally derive the defining equations of the method. Section 3 presents the MATLAB-code that was written to run the method on a model problem. In section 4, we numerically investigate the stability of the method and finally in section 5 we briefly compare our method to the method proposed by Richter in [1].

2 Problem and method

2.1 The wave equation

Throughout this report, we deal with the *scalar second order inhomogeneous wave equation in one spatial variable*

$$u_{tt} - u_{xx} = f \qquad \text{in } \Omega \times J \tag{2.1}$$
$$u = 0 \qquad \text{on } \partial\Omega \times J \tag{2.2}$$
$$u = u_0 \qquad \text{in } \Omega \times \{t = 0\} \tag{2.3}$$
$$u_t = v_0 \qquad \text{in } \Omega \times \{t = 0\} \tag{2.4}$$

with homogeneous Dirichlet boundary conditions (2.2) and initial conditions as specified in (2.3) and (2.4). The spatial domain $\Omega \subset \mathbb{R}$ is assumed to be a bounded interval and J is a time interval $J =]0, T[\subset \mathbb{R}$ and we set $Q = \Omega \times J$.

2.2 Notation

We introduce the following notation for L^2-inner products over an area Q or a curve Γ:

$$(v, w)_Q = \int_Q vw \,\mathrm{d}Q$$
$$\langle v, w\rangle_\Gamma = \int_\Gamma vw \,\mathrm{d}\Gamma$$

For futher simplicity, we use the following abbreviations:

$$\nabla u = (u_x, u_t)$$
$$\Diamond u = (u_x, -u_t)$$
$$\nabla \cdot u = u_x + u_t$$
$$\Box u = u_{xx} - u_{tt}$$

Along an edge e in $\mathbb{R}^2$, for which the function v is defined on both sides of e, we define

$$\{v\} = (v^+ + v^-)/2$$
$$[v] = v^+ n^+ + v^- n^-$$

where v^+ and v^- are the traces of v from the two sides of e and n^+ and n^- are the corresponding outward pointing normal vectors (see figure 1).

We see that $\{v\}$ denotes the *average* of the values of the traces of v from the two sides of e, while $[v]$ denotes the *jump* between the two sides (v is not necessarily continuous across e).

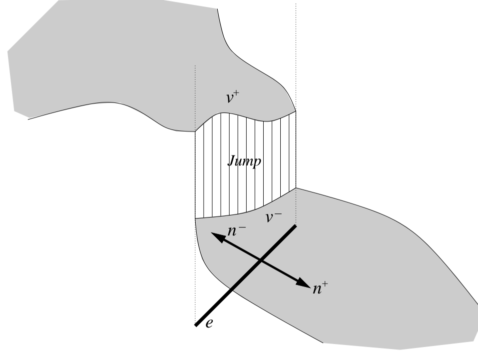


Figure 1: *Discontinuous function v defined across the edge e.*

2.3 Formulation of the method

With the notation of section 2.2, we see that we can write (2.1) as

$$-\Box u = -\nabla \cdot (\Diamond u) = f \tag{2.5}$$

and we note the similarity with the usual way of writing Poisson's equation: $-\Delta u = -\nabla \cdot (\nabla u) = f$. In a similar fashion as in the elliptic case, we can now develop the weak formulation of (2.5) acompanied by (2.2)-(2.4): Multiplying (2.5) by a test function v, integrating both sides over Q and using the integration by parts formula

$$\int_Q \frac{\partial \psi}{\partial x_i} \varphi \, \mathrm{d}\boldsymbol{x} = \int_{\partial Q} \psi \varphi n_i \, \mathrm{d}S - \int_Q \psi \frac{\partial \varphi}{\partial x_i} \, \mathrm{d}\boldsymbol{x} \tag{2.6}$$

element by element, we obtain

$$\int_Q \Diamond u \nabla v \, \mathrm{d}\boldsymbol{x} - \int_{\partial Q} \Diamond u \cdot n v \, \mathrm{d}S = \int_Q f v \, \mathrm{d}\boldsymbol{x} \tag{2.7}$$

or simply

$$(\Diamond u, \nabla v)_Q - \langle \Diamond u \cdot n, v \rangle_{\partial Q} = (f, v)_Q$$

This is the general weak formulation of the problem. However to solve this directly in practice would require boundary conditions specified at the endtime T, which is not desirable for obvious physical reasons. Instead we consider a mesh $\mathcal{M} = \{K_i\}$ that covers Q. Then

$$\int_Q (u_{tt} - u_{xx}) v \, \mathrm{d}\boldsymbol{x} = \int_Q f v \, \mathrm{d}\boldsymbol{x}$$

is equivalent to

$$\sum_{K \in \mathcal{M}} \int_K (u_{tt} - u_{xx}) v \, \mathrm{d}\boldsymbol{x} = \sum_{K \in \mathcal{M}} \int_K f v \, \mathrm{d}\boldsymbol{x}$$

Now using (2.6) over K gives

$$\sum_{K\in\mathcal{M}} (\Diamond u, \nabla v)_K - \sum_{K\in\mathcal{M}} \langle \Diamond u \cdot n, v\rangle_{\partial K} = \sum_{K\in\mathcal{M}} (f, v)_K$$

The DG magic formula, $\sum_{K\in\mathcal{M}} \langle \Diamond u \cdot n, v\rangle_{\partial K} = \langle \{\Diamond u\}, [v]\rangle_{\mathcal{E}} + \langle [\Diamond u], \{v\}\rangle_{\mathcal{E}_I}$ (proof in appendix C), then gives the equation

$$\sum_{K\in\mathcal{M}} (\Diamond u, \nabla v)_K - \langle \{\Diamond u\}, [v]\rangle_{\mathcal{E}} - \langle [\Diamond u], \{v\}\rangle_{\mathcal{E}_I} = \sum_{K\in\mathcal{M}} (f, v)_K$$

where $\mathcal{E}$ denotes the edge set of $\mathcal{M}$ and $\mathcal{E}_I = \mathcal{E} - \partial\mathcal{M}$, i.e. the interior edges. Obviously the term $\langle [\Diamond u], \{v\}\rangle_{\mathcal{E}_I}$ is irrelevant for consistency, so we replace it with the term $\langle \{\Diamond v\}, [u]\rangle_{\mathcal{E}}$ (which likewise does not affect consistency) to obtain a symmetric variational problem. We then arrive at

$$\sum_{K\in\mathcal{M}} (\Diamond u, \nabla v)_K - \langle \{\Diamond u\}, [v]\rangle_{\mathcal{E}} - \langle \{\Diamond v\}, [u]\rangle_{\mathcal{E}} = \sum_{K\in\mathcal{M}} (f, v)_K \tag{2.8}$$

Since it is our intention to search for u in (and test with functions from) a space that contains discontinuous functions, we add the term $\langle \alpha[u], [v]\rangle_{\mathcal{E}}$ to enforce continuity weakly across edges of the mesh. Here α is the so called *stabilization parameter*, usually chosen to be k/h, where k is some constant and h is the minimal sidelength in each element. Hence α is in fact some piecewise constant function of x and t. In each element K, we just need to solve the equation

$$a_K(u, v) = \ell_K(v) \tag{2.9}$$

where

$$a_K(u, v) = (\Diamond u, \nabla v)_K - \langle \{\Diamond u\}, [v]\rangle_{\partial K} - \langle \{\Diamond v\}, [u]\rangle_{\partial K} + \langle \alpha[u], [v]\rangle_{\partial K}$$

and $\ell_K(v) = (f, v)_K$. It is apparent from (2.9) that the solution in each element only depends on its immediate neighbours. This implies that we can solve the entire problem in a piecewise fashion.

Now the numerical scheme is obvious: For all K in $\mathcal{M}$,

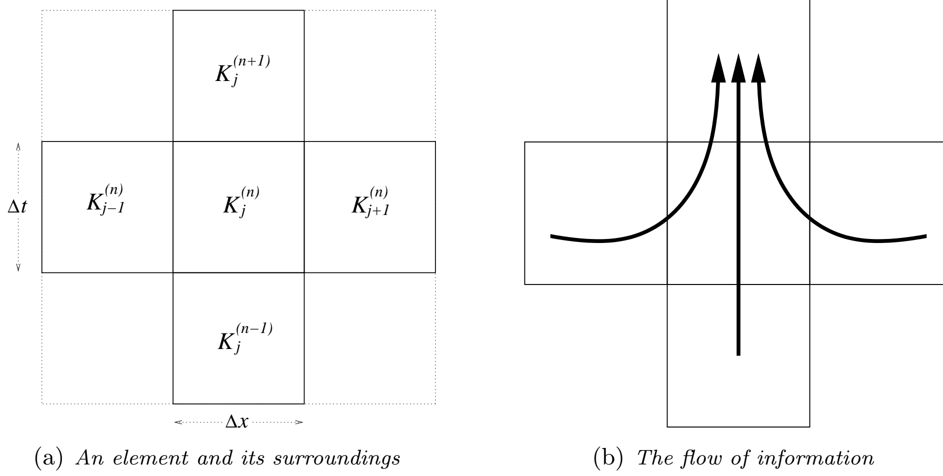
$$\text{find } u_h \in V_h(\mathcal{M}) \text{ such that} \quad a_K(u_h, v_h) = \ell_K(v_h), \quad \text{for all } v_h \in P_p(K) \tag{2.10}$$

where $V_h(\mathcal{M}) = \bigotimes_{K\in\mathcal{M}} P_p(K)$ and $P_p(K)$ is some space of suitable polynomials of (total or maximum) degree p on K.

2.4 Calculation in practice

As already mentioned, solving (2.10) is a local problem for each element. Choosing a basis $\mathcal{B} = \{b_i\}$ for $V_h(\mathcal{M}) = \bigotimes_{K\in\mathcal{M}} P_p(K)$, writing the solution u and the test function (which is only supported in K) as linear combinations of these basis functions, and plugging all this into (2.9) yields the equation

$$a_K(\sum_i \mu_i b_i, \sum_j q_j b_j) = \ell_K(\sum_j q_j b_j)$$



(a) *An element and its surroundings* (b) *The flow of information*

Figure 2: *Stencil in a tensor product mesh*

Since this equation must hold for *all* test functions, it must hold for any choice of the coefficients q_i. Choosing in particular one of these coefficients (e.g. the mth) to be 1 and the rest to be zero then gives

$$a_K(\sum_i \mu_i b_i, b_m) = \ell_K(b_m), \quad m = 1, 2, \ldots$$

which, due to the linearity of a_K, is equivalent to

$$\sum_i \mu_i a_K(b_i, b_m) = \ell_K(b_m), \quad m = 1, 2, \ldots$$

which can be written

$$A\vec{\mu} = \vec{\ell}, \quad \text{where } A_{ij} = a_K(b_i, b_j), \text{ and } \vec{\ell}_i = \ell_K(b_i) \tag{2.11}$$

However, since the terms $a_K(b_i, b_m)$ are non-zero only when b_i and b_m are supported in the same or adjacent elements, and since the b_m are only supported in K (and not in adjacent elements), A is a *belt*-matrix, implying that we can write (2.11) as

$$A_j^{(n-1)}\vec{\mu}_j^{(n-1)} + A_{j-1}^{(n)}\vec{\mu}_{j-1}^{(n)} + A_j^{(n)}\vec{\mu}_j^{(n)} + A_{j+1}^{(n)}\vec{\mu}_{j+1}^{(n)} + A_j^{(n+1)}\vec{\mu}_j^{(n+1)} = \vec{\ell} \tag{2.12}$$

where we used the notation of figure 2(a). Here the $\vec{\mu}$'s contain the coefficients of the solution in the corresponding element and the corresponding matrix contains the only non-zero $a_K(b_i, b_j)$ that exists for that element. We then finally arrive at the actual computational method by simply isolating $\vec{\mu}_j^{(n+1)}$ in (2.12):

$$\vec{\mu}_j^{(n+1)} = \left(A_j^{(n+1)}\right)^{-1} \left(\vec{\ell} - A_j^{(n-1)}\vec{\mu}_j^{(n-1)} - \sum_{i=j-1}^{j+1} A_i^{(n)}\vec{\mu}_i^{(n)}\right) \tag{2.13}$$

We see from the formula, that the solution in the future (larger t) only depends on the solution in the past, which makes physical sense. Figure 2(b) shows how the information flows. The solution in $K_j^{(n+1)}$ depends on the solution in $K_{j-1}^{(n)}$, $K_j^{(n)}$, $K_{j+1}^{(n)}$ and $K_j^{(n-1)}$ giving a so called *5 point stencil*.

3 IMPLEMENTATION

3.1 BASIS FUNCTIONS AND MESH

Let now

- $\mathcal{G}_I(N)$ denote an *equidistant* grid on the interval I with N gridpoints.
- Δx denote the distance between gridpoints in the x-direction
- Δt denote the distance between gridpoints in the t-direction
- T denote the *endtime*, i.e. the time at which the evolution should stop

We then use the mesh of rectangles defined by the points $\mathcal{G}_\Omega(N) \times \mathcal{G}_{[0,T]}(T/\Delta t)$, and this mesh then complety covers the domain $Q = \Omega \times [0,T]$, which is where we want to solve (2.1)-(2.4). This tensorproduct mesh consists of rectangles with corners $a_K^1, \dots, a_K^4$ and they can all be written

$$K_{a^1} = [a_x^1, a_x^1 + \Delta x] \times [a_t^1, a_t^1 + \Delta t]$$

see figure 3.

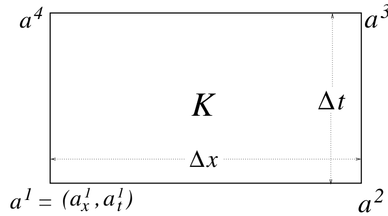


Figure 3: *A generic element of the mesh*

Note that in this construction we have taken equidistant grids in each coordinate direction. The consequence is that the sidelengths of the elements of $\mathcal{M}$ are the same for all elements, namely Δx and Δt. This is chosen for simplicity. The code that will be presented in section 3.2 is however perfectly capable of handling the more general non-equidistant mesh.

Now let K be some element of $\mathcal{M}$. We denote by $\mathcal{Q}_1(K)$ the space of polynomials of degree at most 1 in each coordinate direction, supported only in K, i.e.

$$\mathcal{Q}_1(K) = \text{span}\{1, x, t, xt\} \quad \text{in } K$$

The test and trial space we use is then $V_h = \bigotimes_{K \in \mathcal{M}} Q_1(K)$. Note that this mesh dependent space contains discontinuous functions, since we do not require continuity across the edges. Instead this continuity is weakly enforced, as described in section 2.3.

For easier implementation we use the basis $\mathcal{B} = \{b_1, b_2, b_3, b_4\}$ of the space $\mathcal{Q}_1(K)$ where

$$b_i = \hat{b}_i \circ \Phi_K^{-1}, \quad i = 1, \dots, 4$$

where the $\hat{b}_i$ are the standard bilinear basis functions on the reference square $\hat{K} = [0,1]^2$ and Φ_K is the affine mapping, that maps $\hat{K}$ bijectively to K. Explicitly

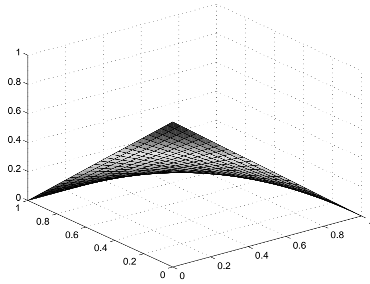
$$\begin{aligned}
\hat{b}_1(\hat{x},\hat{t}) &= (1-\hat{x})(1-\hat{t}) \\
\hat{b}_2(\hat{x},\hat{t}) &= \hat{x}(1-\hat{t}) \\
\hat{b}_3(\hat{x},\hat{t}) &= \hat{x}\hat{t} \\
\hat{b}_4(\hat{x},\hat{t}) &= (1-\hat{x})\hat{t}
\end{aligned}$$

and

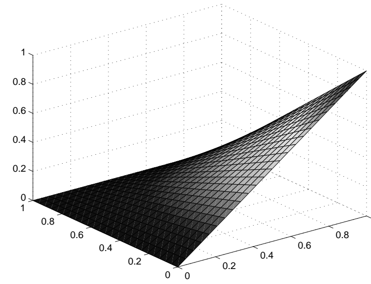
$$\Phi_K(\hat{x},\hat{t}) = \begin{pmatrix} \Delta x & 0 \\ 0 & \Delta t \end{pmatrix} \begin{pmatrix} \hat{x} \\ \hat{t} \end{pmatrix} + \begin{pmatrix} a_x^1 \\ a_t^1 \end{pmatrix}$$

where $a^1 = (a_x^1, a_t^1)$ is the lower left corner of the element K.

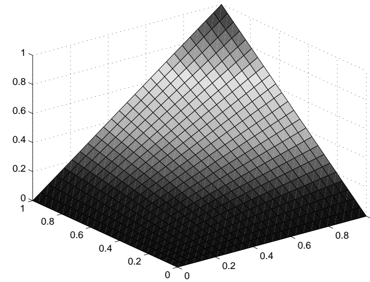
These basis functions have the nice property that $b_i(a^j) = \delta_{ij}$ (the Kroenecker delta), which makes many terms in the integrals that we need to calculate vanish. In figure 4 the functions $\hat{b}_1, \ldots, \hat{b}_4$ are depicted on the reference square.



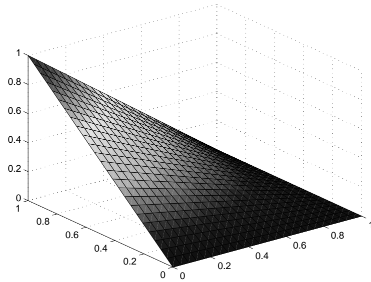
(a) *The function* $\hat{b}_1(\hat{x},\hat{t}) = (1-\hat{x})(1-\hat{t})$



(b) *The function* $\hat{b}_2(\hat{x},\hat{t}) = \hat{x}(1-\hat{t})$



(c) *The function* $\hat{b}_3(\hat{x},\hat{t}) = \hat{x}\hat{t}$



(d) *The function* $\hat{b}_4(\hat{x},\hat{t}) = (1-\hat{x})\hat{t}$

Figure 4: *Basis functions for the space* $\mathcal{Q}_1(\hat{K})$

3.2 STRUCTURE OF CODE

Since we are more interested in the theoretical aspects of the computational method, the code was written so that it is easy understandable, sometimes at the cost of speed.

We store and compute the solution in *timeslabs*. One timeslab is simply one *row* of elements, e.g. the part of the mesh covering $\Omega \times [t_0, t_0 + dx]$, where t_0 is the t-value of a lower corner of an element in the timeslab. We can then store the solution in a 3D-array $S \in \mathbb{R}^{4 \times N \times (T/\Delta t)}$. In the mth layer $L_m \in \mathbb{R}^{4 \times N}$ of S we then store the point values of the four corners in a column and we do this for each of the N elements. Then $S = [L_1, \ldots, L_{(T/\Delta t)}]$.

Below an overview of the structure of the code is presented, starting with the highest level functions and then descending:

- `evolve`(Ω, Δx, Δt, T, α)

  → Calculate L_1 and L_2 from initial conditions (2.3)-(2.4) and store in $S(\cdot,\cdot,1)$ and $S(\cdot,\cdot,2)$.

  → for `step` $= 3 \ldots T/\Delta t$
  $S(\cdot,\cdot,$`step`$) =$ `calc_new_slab`(two previous slabs)
  `end`

  → Plot solution with `plot_solution`(S).

  `end`

- `new_slab` = `calc_new_slab`(two previous slabs)

  → for `j` $= 2 \ldots N-1$
  $\vec{\mu}_{\mathrm{j}}^{(n+1)} =$ `calc_one_elem`($\vec{\mu}_{\mathrm{j}-1}^{(n)}$, $\vec{\mu}_{\mathrm{j}}^{(n)}$, $\vec{\mu}_{\mathrm{j}+1}^{(n)}$, $\vec{\mu}_{\mathrm{j}}^{(n-1)}$)
  `end`

  → Calculate point values for elements on boundary

  `end`

- $\vec{\mu}_j^{(n+1)} =$ `calc_one_elem`($\vec{\mu}_{j-1}^{(n)}$, $\vec{\mu}_j^{(n)}$, $\vec{\mu}_{j+1}^{(n)}$, $\vec{\mu}_j^{(n-1)}$)

  → Define elements matrices from (2.11)-(2.12)

  → Calculate $\mu_j^{(n+1)}$ from (2.13):

$$\vec{\mu}_j^{(n+1)} = \left(A_j^{(n+1)}\right)^{-1} \left(\vec{\ell} - A_j^{(n-1)} \vec{\mu}_j^{(n-1)} - \sum_{i=j-1}^{j+1} A_i^{(n)} \vec{\mu}_i^{(n)} \right)$$

  `end`

The full transcripts of the MATLAB-code used can be seen in appendix A. As seen, the code is rather simple, and the only real challenge lies in the computation of the entries of the element matrices. When using the local space $\mathcal{Q}_p(K)$ on each element K, we obtain $[(p+1)^2] \times [(p+1)^2]$ matrices. In the implementation done here, we used $p = 1$, and hence the element matrices were of size 4×4. With this relatively small size, the integrations needed for the calculation of the entries in these matrices can be carried through analytically. Even though the same is possible for larger p, this quickly gets messy, and one could use numerical quadrature instead.

In appendix B the Maple-code for analytically calculating the entries of the element matrices for $p = 1$ is included.

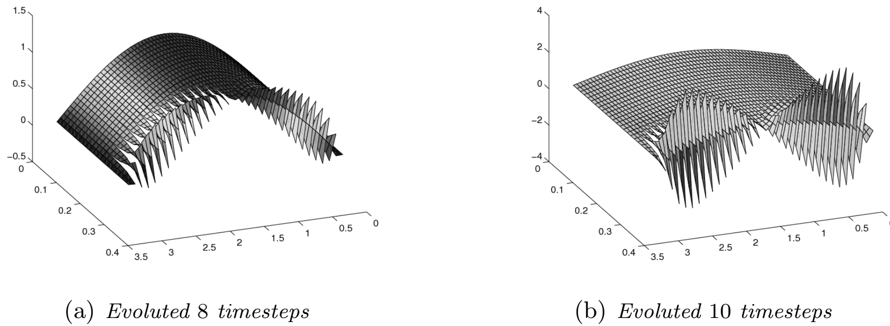
Throughout the process of writing the code, MATLAB was used to check if the method was consistent (as an indicator of possible errors or typos). This was done by the function seen in appendix A.6. That turned out to be a valuable tool, eliminating many mistakes on the way.

3.3 Numerical experiments: Instability

The program was tested on the following model problem:

$$
\begin{aligned}
u_{tt} - u_{xx} &= 0 && \text{in } [0,\pi] \times J \\
u(0,t) = u(\pi,t) &= 0 && \forall t \in J \\
u &= \sin(x) && \text{in } [0,\pi] \times \{t=0\} \\
u_t &= 0 && \text{in } \Omega \times \{t=0\}
\end{aligned}
$$

which has the solution $u(x,t) = \sin(x)\cos(t)$. It was however imidiately clear that the program produced an exploding solution for this problem:



(a) *Evoluted* 8 *timesteps* (b) *Evoluted* 10 *timesteps*

Figure 5: *Result of running program with* $\Delta x = 0.1$*,* $\Delta t = 0.03$ *and* $\alpha = 1/3$

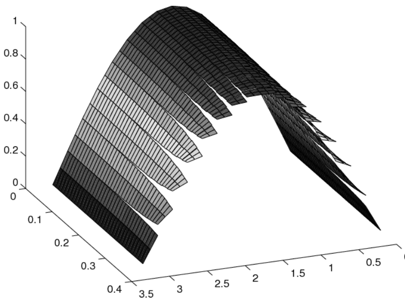


Figure 6: *Result of running program with* $\Delta x = 0.2$*,* $\Delta t = 0.03$ *and* $\alpha = 0$.

Clearly we have produced something that, for this problem, is unstable. This unfortunate property will be closer investigated in the next section.

4 STABILITY

4.1 THEORY

We want to investigate the numerical stability of (the homogeneous part of) our method, which is defined by

$$A_j^{(n+1)}\vec{\mu}_j^{(n+1)} = -A_j^{(n)}\vec{\mu}_j^{(n)} - A_{j+1}^{(n)}\vec{\mu}_{j+1}^{(n)} - A_{j-1}^{(n)}\vec{\mu}_{j-1}^{(n)} - A_j^{(n-1)}\vec{\mu}_j^{(n-1)} \tag{4.1}$$

Since this is a linear evolution, we can do a classical von Neumann analysis: We make the substitution $\vec{\mu}_j^{(n)} = e^{i\kappa j\Delta x}\vec{v}^{(n)}$ and plug this into (4.1). After simplifying, we then have

$$\begin{aligned} A_j^{(n+1)}\vec{v}^{(n+1)} &= -A_j^{(n)}\vec{v}^{(n)} - e^{i\kappa\Delta x}A_{j+1}^{(n)}\vec{v}^{(n)} - e^{-i\kappa\Delta x}A_{j-1}^{(n)}\vec{v}^{(n)} - A_j^{(n-1)}\vec{v}^{(n-1)} \\ &= -\left(A_j^{(n)} + e^{i\kappa\Delta x}A_{j+1}^{(n)} + e^{-i\kappa\Delta x}A_{j-1}^{(n)}\right)\vec{v}^{(n)} - A_j^{(n-1)}\vec{v}^{(n-1)} \end{aligned}$$

Now taking $v^{(n+s)} = \vec{a}q^s$ gives

$$\left(A_j^{(n+1)}q\right)\vec{a} = -\left(A_j^{(n)} + e^{i\kappa\Delta x}A_{j+1}^{(n)} + e^{-i\kappa\Delta x}A_{j-1}^{(n)}\right)\vec{a} - \left(A_j^{(n-1)}q^{-1}\right)\vec{a}$$

This is a quadratic eigenvalue problem of the type

$$(qX + Y + q^{-1}Z)\vec{a} = 0 \quad \Leftrightarrow \quad (q^2X + qY + Z)\vec{a} = 0 \tag{4.2}$$

with

$$\begin{aligned} X &= A_j^{(n+1)} \\ Y &= A_j^{(n)} + e^{i\kappa\Delta x}A_{j+1}^{(n)} + e^{-i\kappa\Delta x}A_{j-1}^{(n)} \\ Z &= -A_j^{(n-1)} \end{aligned} \tag{4.3}$$

This problem can be recast into a normal generalized eigenvalue problem in the following manner:

With the substitution $\vec{b} = (qX + Y)\vec{a}$, (4.2) turns into

$$\begin{pmatrix} qI & Z \\ -I & qX + Y \end{pmatrix}\begin{pmatrix} b \\ a \end{pmatrix} = 0$$

which is equivalent to

$$\left[\begin{pmatrix} 0 & Z \\ -I & Y \end{pmatrix} + q\begin{pmatrix} I & 0 \\ 0 & X \end{pmatrix}\right]\begin{pmatrix} b \\ a \end{pmatrix} = 0$$

Now this is a general eigenvalue problem of the form

$$E\vec{\xi} = qF\vec{\xi}, \quad \text{with} \quad E = \begin{pmatrix} 0 & Z \\ -I & Y \end{pmatrix} \quad \text{and} \quad F = \begin{pmatrix} -I & 0 \\ 0 & -X \end{pmatrix} \tag{4.4}$$

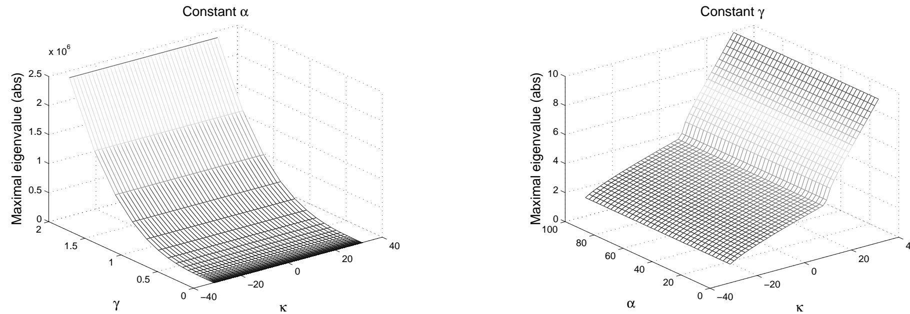
which we can easily solve numerically. If for all $\kappa \in]-\pi/\Delta x, \pi/\Delta x[$ the largest eigenvalue (measured in absolute value) is less than 1, the method is stable - possibly with a certain condition on Δx and Δt, i.e. a local CFL-like condition.

4.2 Numerical investigation

Now we want to solve the generalized eigenvalue problem with the matrices specified in (4.4). We use again the bilinear elements specified in section 3.1, which means that the element matrices are as given in appendix A.5 and on page 26. This is done with the following MATLAB-code:

```
1  function maxev = getmaxev(dx,dt,alph,kappa)     % function start
2                                                  %
3  [As,Aw,Ac,Ae,An] = defmatrices(dx,dt,alph);     % Define element matrices
4
5  X = An;                                                 % Define X,...
6  Y = Ac + exp(i*kappa*dx)*Ae + exp(-i*kappa*dx)*Aw;      % Y and
7  Z = - As;                                               % Z
8  zrs = zeros(4); ID = eye(4);                            % 0- and ID-matrix
9
10 E = [zrs, Z; -ID, Y];                   % Define E and
11 F = [-ID, zrs; zrs -X];                 % F of the generalized EV-problem
12                                         %
13 maxev = max(abs(eig(E,F)));             % Calculate maximal eigenvalue
14                                         %
15 end                                     % function end
16
17 % Auxiliary function that defines the element matrices (Appendix A.5)
18 function [As,Aw,Ac,Ae,An] = defmatrices(dx,dt,alph)
19 ...
20 end
```

Now running this code many times with different values of $\gamma = \Delta t/\Delta x$, α and sampling κ from the interval $]-\pi/\Delta x, \pi/\Delta x[$ gives different maximal eigenvalues. In practice we of course do not redefine the element matrices again and again. The code above serves only for clarification of the principle. The results are plotted in figure 7. Already from figure 7, it is evident that the method suffers from

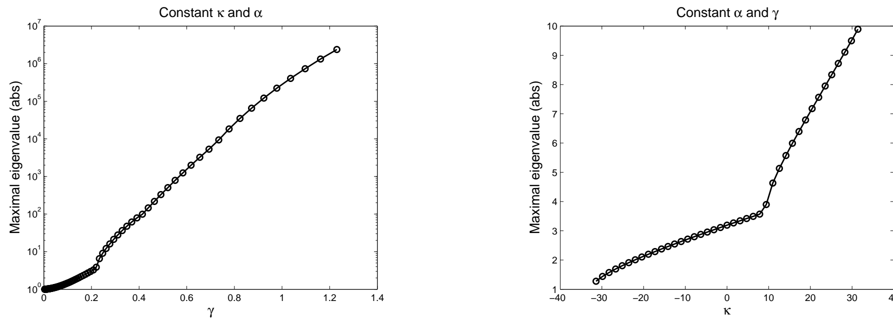


(a) *Constant $\alpha = 1/\Delta x$ and $\Delta x = 1/10$. Varying $\kappa \in]-\pi/\Delta x, \pi/\Delta x[$ and varying Δt.*

(b) *Constant $\gamma = 1/10$ and $\Delta x = 1/10$. Varying $\kappa \in]-\pi/\Delta x, \pi/\Delta x[$ and varying α.*

Figure 7: *The maximal eigenvalue (magnitude) of the eigenvalue problem (4.4) for different choises of the parameters $\gamma = \Delta t/\Delta x$, α and κ.*

instability for certain choices of the parameters. It seems the maximal eigenvalue grows exponentially with linearly growing γ, while the dependence of κ is weaker - even though it looks as if larger κ gives a worse maximal eigenvalue. Figure 8 shows the maximal eigenvalue with yet another parameter held constant. We see strong evidence that the method is *unconditionally* unstable. In figure 8(a), we see that



(a) *Constant $\alpha = 1/\Delta t$ and constant $\kappa = 1$. Varying Δt and $\Delta x = 1/10$.*

(b) *Constant $\alpha = 1/\Delta t$ and $\gamma = 1/10$. Varying $\kappa \in]-\pi/\Delta x, \pi/\Delta x[$ and $\Delta x = 1/10$.*

Figure 8: *The maximal eigenvalue (magnitude) of the eigenvalue problem (4.4) for different choices of the parameters $\gamma = \Delta t/\Delta x$, α and κ.*

when γ approaches zero, the maximal eigenvalue semms to approach 1, but from above, indicating that under no conditions, we get a maximal eigenvalue below 1. Figure 8(b) shows that for a constant γ greater than 0 (namely 1/10), the maximal eigenvalue is greater than 1 *for all* κ, again indicating instability. In figure 9 we see a close-up graph of what happens for γ close to zero and we simply see again, that the eigenvalue approaches 1 from above. This evidence of instability is of course

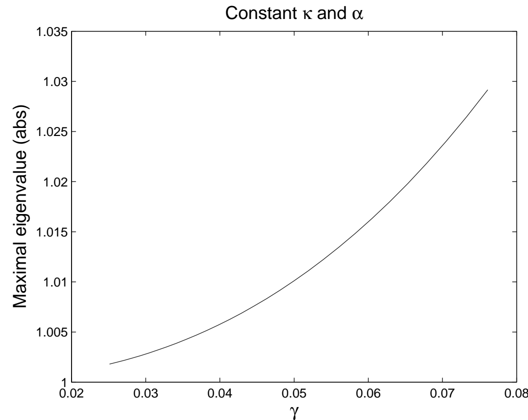


Figure 9: *Constant $\alpha = 1/\Delta t$ and constant $\kappa = 1$. Varying Δt and $\Delta x = 1/10$.*

what we expected from the numerical experiments that we saw in section 3.3 and it unfortunately confirms the fact that we have developed a method that is certainly unstable if bilinear elements are used on a tensor product mesh. This however says nothing about if the method is stable for other elements, since the matrices used in the calculation of the eigenvalues are specific to this particular finite element space.

5 Another approach: G. Richter

In this section we take a brief look at a somewhat similar semi-discontinuous space-time FE-method devised by G.R. Richter in [1] from 1994.

5.1 The method

We first generalize the notation introduced in section 2.2 to d spatial dimensions by simply redefining as follows

$$\begin{aligned} \bar{\nabla} u &:= (u_{x_1}, \dots, u_{x_d}) \\ \nabla u &:= (\bar{\nabla} u, u_t) \\ \Diamond u &:= (\bar{\nabla} u, -u_t) \\ \nabla \cdot u &:= u_{x_1} + \cdots + u_{x_d} + u_t \\ \bar{n} &:= (n_{x_1}, \dots, n_{x_d}) \\ n &:= (\bar{n}, n_t) \end{aligned}$$

Then

$$-\nabla \cdot (\Diamond u) = f \quad \Leftrightarrow \quad u_{tt} - \Delta u = f$$

which is exactly the slighty more general wave equation Richter tries to solve in his paper.

For an element K of the mesh, Richter divides the boundary of it in three categories: the *inflow* boundary $\Gamma_{\text{in}}(K)$ if $n_t < 0$, *outflow* boundary $\Gamma_{\text{out}}(K)$ if $n_t > 0$ and the ∂Q-boundary if $n_t = 0$ and generally requires that $n_t = 0$ only on the boundary of Q.

Richter then uses the bilinear form

$$\hat{a}_K(u, v) = (u_{tt} - \Delta u, v_t)_K - \int_{\Gamma_{\text{in}}(K)} \left\{ ([u_t] v_t + [\bar{\nabla} u] \cdot \bar{\nabla} v) n_t - ([u_t] \bar{\nabla} v + [\bar{\nabla} u] v_t) \bar{n} \right\}$$

With the integration by parts formula (2.6) we see that

$$(u_{tt} - \Delta u, v_t)_K = -(\nabla \cdot \Diamond u, v_t) = (\Diamond u, \nabla v_t)_K - \langle \Diamond u \cdot n, v_t \rangle_{\partial K}$$

which after usage of the DG magic formula turns into exactly our bilinear form of (2.8) (i.e. the final bilinear form excluding the interior penalty term).

Richther, however, does *not* make use of an interior penalty term. His numerical scheme reads:

For all $K \in \mathcal{M}$, seek $u_h \in V_h$ such that

1. $\hat{a}(u_h, v_h) = (f, (v_h)_t)_K$ for all $v_h \in P_p(K)$
2. $[u_h(x^*)] = 0,$ for *one* arbitrarily chosen point $x^* \in \Gamma_{\text{in}}(K)$

where $P_p(K)$ and V_h are as in section 2.3.

The second constraint of continuity in one point (this could for example be the midpoint of the edge) serves two purposes. Since we test with $\partial_t v_h$, we actually indirectly reduce the size of the testspace, and thus the second requirement is needed to close the system, i.e. bring the number of equations up to the number of unknowns. Secondly it enforces the continuity of the solution in combination with the weak enforcement from the second term (the integral over $\Gamma_{\text{in}}(K)$) of $\hat{a}_K$. Finally Richter enforces the boundary conditions strongly.

5.2 Mesh requirements

Possibly the most interesting difference between Richters method and the method presented in section 2 is the mesh requirements imposed by Richter. He requires that all non-boundary edges satisfy

$$|n_t| - |\bar{n}| \geq \gamma > 0 \tag{5.1}$$

which is equivalent to requiring that n makes an angle less than $\pi/4$ from the t-direction. This requirement specifically rules out vertical edges in the interior, since we there have $|n_t| = 0$.

Since this requirement limits the maximal time-distance within one element in relation to the maximal space-distance, this requirement is in fact some kind of CFL-condition. If we restrict ourselves to using meshes that repeat the same pattern over and over again, it is not difficult to imagine meshes that satisfy this condition.

However when we are in several spatial dimensions the creation of meshes that still satisfy (5.1) is non-trivial. Richter, Falk and Monk give examples of such meshes in [2, 3]. One of the strengths of even using a method on a true space-time mesh is the possibility of refining the mesh locally according to possible irregularities in the solution. Clearly this also requires some thought when constructing meshes that still must satisfy (5.1). This is a drawback of the Richters method - even though it can be handled.

5.3 Comparison

There are two obvious differences between Richters method and the method we devised:

1. The different ways of enforcing continuity

2. The difference in the meshes allowed

Exactly to determine the importance of the first point above requires a deeper analysis and is beyond our scope here. But we can of course note that the requirement of continuity at one point on every edge required by Richter plays an essential role in the proof of stability of his method.

It is easier to figure out whether or not the usage of a mesh with vertical edges is what causes instability in our method. This can be tested simply by implementing it and perform a few experiments. This is something that we will do in the very near future.

6 Concluding remarks

In this thesis we have attempted to construct an interior penalty space-time discontinuous Galerkin finite element method for the wave equation. The motivation was the succes of the same method for elliptic problems such as Poisson's equation, originally in [6].

We implemented the method in Matlab and unfortunately numerical experiments showed blow up in the solution - and this regardless of the choice of parameters. This property was confirmed by a Von Neumann stability analysis, which gave numerical evidence that the method indeed is unconditionally unstable for the type of finite element space we used.

There are however other space-time DG and semi-DG methods that have succeeded for hyperbolic equations, [1, 2, 3, 5]. These methods differ from our method firstly in that they do *not* use the method of interior penalty of interelement jumps to enforce continuity. Secondly, and this is specific for [1, 2, 3], certain (quite strict) conditions are imposed on the mesh. The mesh we used does *not* satisfy these requirements.

Whether or not imposing these mesh-conditions on our method would remedy the problems of our method is unclear since we currently do not know if the problem is caused by the interior penalty, the mesh structure, the local FE space used or something else. Our implementation was facilitated considerably by the use of a tensorproduct mesh. It would be a more sizeable job to implement the method on more general meshes and this is certainly one of our tasks for the future.

References

[1] Gerard R. Richter. An explicit finite element method for the wave equation. *Applied Numerical Mathematics*, 16, 1994.

[2] Richard S. Falk and Gerard R. Richter. Explicit finite element methods for symmetric hyperbolic equations. *SIAM J. Numer. Anal.*, 36(3), 1999.

[3] Peter Monk and Gerard R. Richter. A discontinuous galerkin method for linear symmetric hyperbolic systems in inhomogeneous media. 2003.

[4] Marcus J. Grote, Anna Schneebeli, and Dominik Schötzau. Discontinuous galerkin finite element method for the wave equation. *Submitted to SIAM J. Numer. Anal.*, May 2005. Available at http://www.math.unibas.ch/preprints/preprints05/preprint2005-03.pdf.

[5] Gregory M. Hulbert and Thomas J.R. Hughes. Space-time finite element methods for second-order hyperbolic equations. *Computer Methods in Applied Mechanics and Engineering*, 84, 1990.

[6] Mary Fanett Wheeler. An elliptic collocation-finite element method with interior penalties. *SIAM Journal on Numerical Analysis*, 15, 1978.

[7] D. Arnold, F. Brezzi, B. Cockburn, and L. Marini. Unified analysis of discontinuous galerkin methods for elliptic problems. *SIAM J. Numer. Anal.*, 39, 2002. Available at http://128.101.10.128/~arnold/papers/dgerr.pdf.

[8] R. Hiptmair and C. Schwab. Numerics of elliptic and parabolic boundary value problems. Online lecture slides, 2006. Available at http://www.sam.math.ethz.ch/hiptmair/NAPDE_06.pdf.

[9] R. Hiptmair. Numerics of hyperbolic partial differential equations. Online lecture slides, 2007. Available at http://www.sam.math.ethz.ch/hiptmair/tmp/NUMHYP_07.pdf.

[10] Eric T. Chung and Björn Engquist. Optimal discontinuous galerkin methods for wave propagation. *SIAM J. Numer. Anal.*, 2006. Available at http://www.math.uci.edu/~tchung/preprint/sdg.pdf.

[11] B. Cockburn. Discontinuous galerkin methods. *ZAMM*, 83(11), October 2003.

[12] Lawrence C. Evans. *Partial Differential Equations*. American Mathematical Society, 1998.

[13] Lonny L. Thompson and Peter M. Pinsky. A space-time finite element method for the exterior acoustics problem. *The Journal of the Acoustical Society of America*, 99, 1996.

A Code: Matlab

A.1 evolve.m

```
1  function sol = evolve(xs,xe,dx,timesteps,dt,k)
2
3  N = ceil(abs(xe-xs) / dx);          % Number of gridpoints
4  spaceGRID = linspace(xs,xe,N+1);    % The spacegrid
5  sol = zeros(4,N,timesteps+2);       % Initialize solution 3D-array
6
7  % Calculate initial conditions:
8  [sol(:,:,1),sol(:,:,2)] = calc_ICs(spaceGRID);
9
10 % Iteration
11 for j=3:timesteps+2
12
13     % Calculate next timeslab given the two previous
14     sol(:,:,j) = calc_new_slab(sol(:,:,j-1),sol(:,:,j-2),N,dx,dt,k);
15
16 end
17
18 % Plot the solution:
19 plotsolution(sol,spaceGRID,dt,timesteps,N);
```

A.2 calc_new_slab.m

```
1  function new_slab = calc_new_slab(slab_im1,slab_im2,N,dx,dt,k)
2
3  % N = Number of elements per slab
4
5  new_slab = zeros(size(slab_im1));  % Initialize new slab
6
7  % For each element in slab, calculate its value,
8  % given the relevant values from the previous two slabs:
9  for i=2:N-1
10     new_slab(:,i) = ...
11      calc_one_elem( slab_im1(:,i-1),slab_im1(:,i),...
12                       slab_im1(:,i+1),slab_im2(:,i),dx,dt,k );
13 end
14 % Calculate boundary elements:
15 % These depend on one less element, hence the special function:
16 new_slab(:,1) = calc_one_elem_BDL_ex(slab_im1(:,1),...
17                 slab_im1(:,2),slab_im2(:,1),dx,dt,k);
18 new_slab(:,N) = calc_one_elem_BDR_ex(slab_im1(:,N-1),...
19                 slab_im1(:,N),slab_im2(:,N),dx,dt,k);
```

A.3 calc_one_elem.m

```
1  function [muN,As,Aw,Ac,Ae,An] = calc_one_elem(muW,muC,muE,muS,dx,dt,k)
2
3  % function that calculates alpha from h
4  % where h is the min. sidelength of the element:
5  alpha_func = @(h)( k / h );
6  alph = alph_func(dt);         % Current alpha
7
8  % Define element matrices:
9  [As,Aw,Ac,Ae,An] = defmatrices(dx,dt,alph)
10
11 % Calculate new element:
12 muN = - AnINV * (As*muS + Aw*muW + Ac*muC + Ae*muE );
```

A.4 calc_ICs.m

```
1  function [slab_m1,slab_0] = calc_ICs(spaceGRID);
2
3  u0 = @(x)( sin(x) );
4  %v0 = @(x)( ... );
5
6  N = length(spaceGRID)-1;
7
8  temp = zeros(4,N);
9
10 % Assign values to temp according to the numbering used throughout:
11 for i=1:N
12
13     temp([1,4],i) = u0( spaceGRID(i) );
14     temp([2,3],i) = u0( spaceGRID(i+1) );
15
16 end
17
18 % Assuming v(x) = 0
19 % the two first slabs are then identical:
20 slab_m1 = temp;
21 slab_0 = temp;
```

A.5 defmatrices.m

```
1  function [As,Aw,Ac,Ae,An] = defmatrices(dx,dt,alph)
2
3  % This function defines the element matrices for p = 1
4  % so the matrices are 4x4.
5  % Notation used: As = A-south, Aw = A-west, Ac = A-center, etc.
6  % All entries stem from exact integrations (See Maple code)
7
8  gam = dt / dx; gamINV = 1/gam;
9
10 % ---------------------------- As ------------------------------- %
11 As = zeros(4);
12 As([1 6 11 16]) = 1/6;
13 As([2 5 12 15]) = 1/12;
14 As([9 14])      = -1/6 * (1 + alph * dt);
15 As([10 13])     = -1/3 * (1 + alph * dt);
16 As = As*gamINV;
17
18 % ---------------------------- Aw ------------------------------- %
19 Aw = zeros(4);
20 Aw([1 6 11 16]) = - 1/6;
21 Aw([4 7 10 13]) = - 1/12;
22 Aw([5 12])      = - 1/3 * (-1 + alph * dx);
23 Aw([8 9])       = - 1/6 * (-1 + alph * dx);
24 Aw = Aw*gam;
25
26 % ---------------------------- Ac ------------------------------- %
27 Ac = zeros(4);
28 Ac([1 6 11 16]) = 1/3 * (dx * alph + dt * alph);
29 Ac([2 5 12 15]) = 1/6 * dx * alph;
30 Ac([4 7 10 13]) = 1/6 * dx * alph;
31
32 % ---------------------------- Ae ------------------------------- %
33 Ae = zeros(4);
34 Ae([1 6 11 16]) = - 1/6;
35 Ae([4 7 10 13]) = - 1/12;
36 Ae([2 15])      = - 1/3 * (-1 + alph * dx);
```

```
37 Ae([3 14])       = - 1/6 * (-1 + alph * dx);
38 Ae = Ae*gam;
39
40 % ---------------------------- An ------------------------------- %
41 An = zeros(4);
42 An([1 6 11 16]) = 1/6;
43 An([2 5 12 15]) = 1/12;
44 An([3 8])       = -1/6 * (1 + alph * dt);
45 An([4 7])       = -1/3 * (1 + alph * dt);
46 An = An * gamINV;
47
48 end
```

A.6 CONSCHECK.M

```
 1 function diff = conscheck(dx,dt,k)
 2
 3 % This function checks if the entire method is consistent with
 4 % the function f defined below:
 5
 6 f = @(x,t)( x+t );  % The method should be consistent with
 7                     % e.g. linear functions like this one
 8
 9 % Calculate values of "previous elements":
10 muS = zeros(4,1); muW = muS; muC = muS; muE = muS;
11 muS = [f(dx,0), f(2*dx,0), f(2*dx,dt), f(dx,dt)]';
12 muW = [f(0,dt), f(dx,dt), f(dx,2*dt), f(0,2*dt)]';
13 muC = [f(dx,dt), f(2*dx,dt), f(2*dx,2*dt), f(dx,2*dt)]';
14 muE = [f(2*dx,dt), f(3*dx,dt), f(3*dx,2*dt), f(2*dx,2*dt)]';
15
16 % What the method SHOULD produce:
17 REALMUN = [f(dx,2*dt), f(2*dx,2*dt), f(2*dx,3*dt), f(dx,3*dt)]';
18
19 % Calculate what the method DOES produce:
20 muN = calc_one_elem(muW,muC,muE,muS,dx,dt,k);
21
22 % Compare the two:
23 diff = muN - REALMUN;
24 reldiff = diff ./ REALMUN;
25
26 thr = 1e-9;
27
28 % Output if method is consistent, relative to thr:
29 if norm(diff) < thr
30    fprintf('Consistent.')
31 else
32    fprintf('Not consistent.')
33 end
```

A.7 PLOTSOLUTION.M

```
 1 function plotsolution(sol,spaceGRID,dt,timesteps,N);
 2
 3 % This function plots the solution calculated with evolute
 4
 5 % The resolution to be used in EACH dimension in EACH element:
 6 PLOTRES = 3;
 7
 8 figure; hold on; view([12 30 20]);   % INIT figure
 9
10 % For each timeslab:
11 % Plot timeslab with the function plot_time_slab (see below)
```

```
12  for i = 1:timesteps+2
13      cur_points = gen_points_slab(spaceGRID,(i-1)*dt,i*dt);
14      temp = sol(:,1:N,i);
15      cur_vals = temp(:);
16      plot_time_slab(cur_points,cur_vals,PLOTRES);
17
18  end  % END FOR
19
20  % Generate (x,t) points according to numbering used throughout
21  function points = gen_points_slab(spaceGRID,ts,te)
22
23  Nelems = length(spaceGRID)-1;
24  points = zeros(4*Nelems,2);
25
26  for j = 1:Nelems;
27
28      points(j*4-3,:) = [spaceGRID(j),ts];    % Setting the correct
29      points(j*4-2,:) = [spaceGRID(j+1),ts];  % values, numbering from
30      points(j*4-1,:) = [spaceGRID(j+1),te];  % lower left corner of square
31      points(j*4-0,:) = [spaceGRID(j),te];    % and moving counterclockwise
32
33  end % END FOR
34  end % END FUNCTION gen_points_slab
35
36  % Plot ONE time slab:
37  function plot_time_slab(points,vals,plotres)
38
39  % For each element in time slab, plot element using plot_elem_func:
40  for k = 1:4:length(vals);
41
42      cur_points  = points(k:k+3,:) % The (x,t)-points and the function
43      cur_vals    = vals(k:k+3);    % values in currect element
44
45      plot_elem_func(cur_points,cur_vals,plotres);  % Plot the element
46
47  end % END FOR
48  end % END FUNCTION plot_time_slab
49
50  % Plot one element function:
51  function plot_elem_func(a,z,plotres)
52
53  dx = abs( a(2,1) - a(1,1) );              % Delta x
54  dt = abs( a(4,2) - a(1,2) );              % Delta t
55  x=linspace(a(1,1),a(1,1)+dx,plotres);    % x-Gridpoints WITHIN element
56  t=linspace(a(1,2),a(1,2)+dt,plotres);    % t-Gridpoints WITHIN element
57  [X,T]=meshgrid(x,t);                      % Create meshgrid for plotting
58  XH = (X-a(1,1))/dx;                       % Calculate X-hat (from PHI-map)
59  TH = (T-a(1,2))/dt;                       % Calculate T-hat (from PHI-map)
60
61  % Calculate function values in element:
62  ZH = z(1)*(1-XH).*(1-TH) + z(2)*XH.*(1-TH) + z(3)*XH.*TH+z(4)*TH.*(1-XH);
63
64  surf(X,T,ZH); % Plot the function
65
66  end % END FUNCTION plot_elem_func
67  end % END FUNCTION plotsolution
```

B CODE: MAPLE

B.1 CALC_ELEM_MATRICES.MWS

```
> restart:  with(LinearAlgebra):
```

Defining the map Φ for the element and all surrounding elements:

```
> xh[1]:=(x-a1x)/(dx):       th[1]:=(t-(a1t-dt))/(dt):
> xh[2]:=(x-(a1x-dx))/(dx):  th[2]:=(t-a1t)/(dt):
> xh[3]:=(x-a1x)/(dx):       th[3]:=(t-a1t)/(dt):
> xh[4]:=(x-(a1x+dx))/(dx):  th[4]:=(t-a1t)/(dt):
> xh[5]:=(x-a1x)/(dx):       th[5]:=(t-(a1t+dt))/(dt):
```

Defining basis funcstion in the element and all surrounding elements:

```
> for i from 1 by 1 to 5 do
     b1[i]:=unapply((1-th[i])*(1-xh[i]),(x,t)):
  end do:
> for i from 1 by 1 to 5 do
     b2[i]:=unapply(xh[i]*(1-th[i]),(x,t)):
  end do:
> for i from 1 by 1 to 5 do
      b3[i]:=unapply(xh[i]*th[i],(x,t)):
  end do:
> for i from 1 by 1 to 5 do
     b4[i]:=unapply(th[i]*(1-xh[i]),(x,t)):
  end do:
```

Defining outer unit normal vectors to the element and their negatives:

```
> N[1] := Vector([0, -1]):       Nm[1] := -N[1]:
> N[2] := Vector([-1, 0]):       Nm[2] := -N[2]:
> N[4] := Vector([1, 0]):        Nm[4] := -N[4]:
> N[5] := Vector([0, 1]):        Nm[5] := -N[5]:
```

Defining functions used in calculation of bilinear form:

```
> dia := (u,x,t) -> Vector([D[1](u)(x,t), -D[2](u)(x,t) ]):
> jump := (vp,vm,vecp,vecm,x,t) -> vp(x,t)*vecp + vm(x,t)*vecm:
> avg := (up,um) -> 1/2 * (up + um):
```

The bilinear form broken into pieces corresponding to the elements on each side of the center element. Number 1 is south, 2 west, 3 center, 4 east and 5 is north. In these definitions, `u` must be a basis function supported *outside* the center element, while `v` must be supported *inside* the center element. The definitions are from (2.9).

```
> bilin[1] := (um,vp) -> int( -avg(dia(0,x,a1t),dia(um,x,a1t)).
  jump(vp,0,N[1],Nm[1],x,a1t) - avg(dia(vp,x,a1t),dia(0,x,a1t)).
  jump(0,um,N[1],Nm[1],x,a1t) + alphS * jump(0,um,N[1],Nm[1],x,a1t).
  jump(vp,0,N[1],Nm[1],x,a1t) , x=a1x..a1x+dx):

> bilin[2] := (um,vp) -> dt*int( -avg(dia(0,a1x,a1t+(1-s)*dt),
  dia(um,a1x,a1t+(1-s)*dt)).jump(vp,0,N[2],Nm[2],a1x,a1t+(1-s)*dt)
   - avg(dia(vp,a1x,a1t+(1-s)*dt),dia(0,a1x,a1t+(1-s)*dt)).
  jump(0,um,N[2],Nm[2],a1x,a1t+(1-s)*dt) + alphW * jump(0,um,N[2],Nm[2],
  a1x,a1t+(1-s)*dt).jump(vp,0,N[2],Nm[2],a1x,a1t+(1-s)*dt) , s=0..1):
```

```
> bilin[4] := (um,vp) -> int( -avg(dia(0,a1x+dx,t),dia(um,a1x+dx,t)).
  jump(vp,0,N[4],Nm[4],a1x+dx,t) - avg(dia(vp,a1x+dx,t),dia(0,a1x+dx,t)).
  jump(0,um,N[4],Nm[4],a1x+dx,t) + alphE * jump(0,um,N[4],Nm[4],a1x+dx,t).
  jump(vp,0,N[4],Nm[4],a1x+dx,t) , t=a1t..a1t+dt):

> bilin[5] := (um,vp) -> dx*int( -avg(dia(0,a1x+(1-s)*dx,a1t+dt),
  dia(um,a1x+(1-s)*dx,a1t+dt)).jump(vp,0,N[5],Nm[5],a1x+(1-s)*dx,a1t+dt)
   - avg(dia(vp,a1x+(1-s)*dx,a1t+dt),dia(0,a1x+(1-s)*dx,a1t+dt)).
  jump(0,um,N[5],Nm[5],a1x+(1-s)*dx,a1t+dt) + alphN * jump(0,um,N[5],
  Nm[5],a1x+(1-s)*dx,a1t+dt).jump(vp,0,N[5],Nm[5],a1x+(1-s)*dx,
  a1t+dt) , s=0..1):
```

Defining function to calculate the element matrices 1, 2, 4 and 5:

```
> A := i -> Matrix([[
  simplify( bilin[i](b1[i],b1[3]) ),simplify( bilin[i](b2[i],b1[3])),
  simplify( bilin[i](b3[i],b1[3]) ),simplify( bilin[i](b4[i],b1[3]))],
  [simplify( bilin[i](b1[i],b2[3])),simplify( bilin[i](b2[i],b2[3])),
  simplify(bilin[i](b3[i],b2[3])),simplify(bilin[i](b4[i],b2[3]))],
  [simplify( bilin[i](b1[i],b3[3])),simplify( bilin[i](b2[i],b3[3])),
  simplify( bilin[i](b3[i],b3[3])),simplify( bilin[i](b4[i],b3[3]))],
  [simplify( bilin[i](b1[i],b4[3])),simplify( bilin[i](b2[i],b4[3])),
  simplify( bilin[i](b3[i],b4[3])),simplify( bilin[i](b4[i],b4[3]))]]):
```

Now for the element matrix corresponding to the center element, there are more that one integral contributing. We define all contributions:

```
> bilinc[1] := (up,vp) -> int( -avg(dia(up,x,a1t),dia(0,x,a1t)).
  jump(vp,0,N[1],Nm[1],x,a1t) - avg(dia(vp,x,a1t),dia(0,x,a1t)).
  jump(up,0,N[1],Nm[1],x,a1t) + alphS * jump(up,0,N[1],Nm[1],x,a1t).
  jump(vp,0,N[1],Nm[1],x,a1t) , x=a1x..a1x+dx):

> bilinc[2] := (up,vp) -> dt*int(-avg(dia(up,a1x,a1t+(1-s)*dt),
  dia(0,a1x,a1t+(1-s)*dt)).jump(vp,0,N[2],Nm[2],a1x,a1t+(1-s)*dt)
  -avg(dia(vp,a1x,a1t+(1-s)*dt),dia(0,a1x,a1t+(1-s)*dt)).jump(up,0,N[2],
  Nm[2],a1x,a1t+(1-s)*dt) + alphW * jump(up,0,N[2],Nm[2],a1x,a1t+
  (1-s)*dt).jump(vp,0,N[2],Nm[2],a1x,a1t+(1-s)*dt) , s=0..1);

> bilinc[3] := (u,v)->int(int( D[1](u)(x,t)*D[1](v)(x,t)-
  D[2](u)(x,t)*D[2](v)(x,t) ,x=a1x..a1x+dx),t=a1t..a1t+dt);

> bilinc[4] := (up,vp) -> int( -avg(dia(up,a1x+dx,t),dia(0,a1x+dx,t)).
  jump(vp,0,N[4],Nm[4],a1x+dx,t) - avg(dia(vp,a1x+dx,t),dia(0,a1x+dx,t)).
  jump(up,0,N[4],Nm[4],a1x+dx,t) + alphE * jump(up,0,N[4],Nm[4],a1x+dx,t).
  jump(vp,0,N[4],Nm[4],a1x+dx,t) , t=a1t..a1t+dt);

> bilinc[5] := (up,vp) -> dx*int( -avg(dia(up,a1x+(1-s)*dx,a1t+dt),
  dia(0,a1x+(1-s)*dx,a1t+dt)).jump(vp,0,N[5],Nm[5],a1x+(1-s)*dx,
  a1t+dt) - avg(dia(vp,a1x+(1-s)*dx,a1t+dt),dia(0,a1x+(1-s)*dx,a1t+dt)).
  jump(up,0,N[5],Nm[5],a1x+(1-s)*dx,a1t+dt) + alphN * jump(up,0,N[5],
  Nm[5],a1x+(1-s)*dx,a1t+dt).jump(vp,0,N[5],Nm[5],a1x+(1-s)*dx,
  a1t+dt) , s=0..1);
```

Similar to the function `A` above, we define:

```
> Ac := i -> Matrix([[
  simplify(bilinc[i](b1[3],b1[3])),simplify(bilinc[i](b2[3],b1[3])),
  simplify(bilinc[i](b3[3],b1[3])),simplify(bilinc[i](b4[3],b1[3]))],
  [simplify(bilinc[i](b1[3],b2[3])),simplify(bilinc[i](b2[3],b2[3])),
  simplify(bilinc[i](b3[3],b2[3])),simplify(bilinc[i](b4[3],b2[3]))],
  [simplify(bilinc[i](b1[3],b3[3])),simplify(bilinc[i](b2[3],b3[3])),
  simplify(bilinc[i](b3[3],b3[3])),simplify(bilinc[i](b4[3],b3[3]))],
  [simplify(bilinc[i](b1[3],b4[3])),simplify(bilinc[i](b2[3],b4[3])),
  simplify(bilinc[i](b3[3],b4[3])),simplify(bilinc[i](b4[3],b4[3]))]]);
```

Now the following commands produce the element matrices that we wanted:

```
> As:=A(1):  Aw:=A(2):  Ae:=A(4):  An:=A(5):
> Ac_TOTAL := Ac(1) + Ac(2) + Ac(3) + Ac(4) + Ac(5):
```

And the result is

$$\texttt{As} = \frac{\Delta x}{\Delta t}\begin{pmatrix} 1/6 & 1/12 & -(1+\alpha\Delta t)/6 & -(1+\alpha\Delta t)/3 \\ 1/12 & 1/6 & -(1+\alpha\Delta t)/3 & -(1+\alpha\Delta t)/6 \\ 0 & 0 & 1/6 & 1/12 \\ 0 & 0 & 1/12 & 1/6 \end{pmatrix}$$

$$\texttt{Aw} = \frac{\Delta t}{\Delta x}\begin{pmatrix} -1/6 & (1-\alpha\Delta x)/3 & (1-\alpha\Delta t)/6 & -1/12 \\ 0 & -1/6 & -1/12 & 0 \\ 0 & -1/12 & -1/6 & 0 \\ -1/12 & (1-\alpha\Delta x)/6 & (1-\alpha\Delta t)/3 & -1/6 \end{pmatrix}$$

$$\texttt{Ae} = \frac{\Delta t}{\Delta x}\begin{pmatrix} -1/6 & 0 & 0 & -1/12 \\ (1-\alpha\Delta x)/3 & -1/6 & -1/12 & (1-\alpha\Delta x)/6 \\ (1-\alpha\Delta x)/6 & -1/12 & -1/6 & (1-\alpha\Delta x)/3 \\ -1/12 & 0 & 0 & -1/6 \end{pmatrix}$$

$$\texttt{An} = \frac{\Delta x}{\Delta t}\begin{pmatrix} 1/6 & 1/12 & 0 & 0 \\ 1/12 & 1/6 & 0 & 0 \\ -(1+\alpha\Delta t)/6 & -(1+\alpha\Delta t)/3 & 1/6 & 1/12 \\ -(1+\alpha\Delta t)/3 & -(1+\alpha\Delta t)/6 & 1/12 & 1/6 \end{pmatrix}$$

$$\texttt{Ac} = \alpha\begin{pmatrix} (\Delta x+\Delta t)/3 & \Delta x/6 & 0 & \Delta t/6 \\ \Delta x/6 & (\Delta x+\Delta t)/3 & \Delta t/6 & 0 \\ 0 & \Delta t/6 & (\Delta x+\Delta t)/3 & \Delta x/6 \\ \Delta t/6 & 0 & \Delta x/6 & (\Delta x+\Delta t)/3 \end{pmatrix}$$

which is exactly the expressions we used in the matlab implementation (see appendix A.5).

C Proof of the DG Magic Formula

We want to proof the *DG Magic Formula*, which in the version we need it, is

$$\sum_K \int_{\partial K} \Diamond u \cdot \boldsymbol{n} v \,\mathrm{d}S = \int_{\mathcal{E}} \{\Diamond u\}[v]_{\boldsymbol{n}} + \int_{\mathcal{E}_I} [\Diamond u]_{\boldsymbol{n}}\{v\}$$

where the sum runs over all elements K of the mesh $\mathcal{M}$. The set of edges of $\mathcal{M}$ is denoted $\mathcal{E}$, the set of boundary edges is $\mathcal{E}_\partial$ and $\mathcal{E}_I = \mathcal{E} - \mathcal{E}_\partial$. In addition, we let v^+ and v^- denote the trace of the function v on either side of an edge e. Similarly are $\boldsymbol{n}^+$ and $\boldsymbol{n}^-$ outwards directed unit normal vectors corresponding to v^+ and v^-, and obviously $\boldsymbol{n}^- = -\boldsymbol{n}^+$.

First we expand the first term

$$\begin{aligned}\int_{\mathcal{E}} \{\Diamond u\}[v]_{\boldsymbol{n}} &= \sum_{\mathcal{E}_I} \int_e \{\Diamond u\}[v]_{\boldsymbol{n}} + \sum_{\mathcal{E}_\partial} \int_e \{\Diamond u\}[v]_{\boldsymbol{n}} \\ &= \sum_{e \in \mathcal{E}_I} \int_e \frac{1}{2}(\Diamond u^+ + \Diamond u^-)(\boldsymbol{n}^+ v^+ + \boldsymbol{n}^- v^-) + \sum_{e \in \mathcal{E}_\partial} \int_e \Diamond u \cdot \boldsymbol{n} v\end{aligned}$$

The second term is

$$\int_{\mathcal{E}_I} [\Diamond u]_{\boldsymbol{n}}\{v\} = \sum_{e \in \mathcal{E}_I} \int_e (\boldsymbol{n}^+ \Diamond u^+ + \boldsymbol{n}^- \Diamond u^-)\frac{1}{2}(v^+ + v^-)$$

Summing the two and expanding gives

$$\begin{aligned}&\int_{\mathcal{E}} \{\Diamond u\}[v]_{\boldsymbol{n}} + \int_{\mathcal{E}_I} [\Diamond u]_{\boldsymbol{n}}\{v\} = \\ &\sum_{e \in \mathcal{E}_I} \int_e \frac{1}{2}(\Diamond u^+ + \Diamond u^-)(\boldsymbol{n}^+ v^+ + \boldsymbol{n}^- v^-) + (\boldsymbol{n}^+ \Diamond u^+ + \boldsymbol{n}^- \Diamond u^-)\frac{1}{2}(v^+ + v^-) + \sum_{e \in \mathcal{E}_\partial} \int_e \Diamond u \cdot \boldsymbol{n} = \\ &\frac{1}{2} \sum_{e \in \mathcal{E}_I} \int_e \big[\Diamond u^+ \boldsymbol{n}^+ v^+ + \Diamond u^- \boldsymbol{n}^- v^- + \Diamond u^+ \boldsymbol{n}^+ v^+ + \Diamond u^- \boldsymbol{n}^- v^- + \\ &\qquad \Diamond u^+ \boldsymbol{n}^- v^- + \Diamond u^- \boldsymbol{n}^+ v^+ + \Diamond u^+ \boldsymbol{n}^+ v^- + \Diamond u^- \boldsymbol{n}^- v^+\big] + \sum_{e \in \mathcal{E}_\partial} \int_e \Diamond u \cdot \boldsymbol{n}\end{aligned}$$

Now because $\boldsymbol{n}^- = -\boldsymbol{n}^+$, the four terms in the second line in the [ ]-brackets cancel, and we are left with

$$\begin{aligned}\int_{\mathcal{E}} \{\Diamond u\}[v]_{\boldsymbol{n}} + \int_{\mathcal{E}_I} [\Diamond u]_{\boldsymbol{n}}\{v\} &= \sum_{e \in \mathcal{E}_I} \int_e \Diamond u^+ \boldsymbol{n}^+ v^+ + \Diamond u^- \boldsymbol{n}^- v^- + \sum_{e \in \mathcal{E}_\partial} \int_e \Diamond u \cdot \boldsymbol{n} \\ &= 2 \sum_{e \in \mathcal{E}_I} \int_e \Diamond u \cdot \boldsymbol{n} v + \sum_{e \in \mathcal{E}_\partial} \int_e \Diamond u \cdot \boldsymbol{n} \\ &= \sum_{K \in \mathcal{M}} \sum_{e \in K} \Diamond u \cdot \boldsymbol{n} v = \sum_{K \in \mathcal{M}} \int_{\partial K} \Diamond u \cdot \boldsymbol{n} v\end{aligned}$$

which is exactly what we wanted to show.