SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation of coupled acoustic-elastic solvers in the ExaHyPE2 hyperbolic PDE Engine

Jingzhou Long

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Implementation of coupled acoustic-elastic solvers in the ExaHyPE2 hyperbolic PDE Engine

Implementierung gekoppelter Akustik- und Elastikmodelle in ExaHyPE2, einer Engine zum Lösen von hyperbolischen partiellen Differentialgleichungen

Author:	Jingzhou Long
Supervisor:	Prof. Dr. Michael Georg Bader
Advisor:	M.Sc. Marc Marot-Lassauzaie
Submission Date:	15.02.2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.02.2024 Jingzhou Long

Acknowledgments

I would like to thank my friends and family for their continuous support throughout my work. They have been a constant anchor in my life.

I would also like to express my thanks to my advisor Marc Marot-Lassauzaie for all his support and helpful advice during difficult phases of this thesis.

Abstract

ExaHype2 is an open-source engine to solve hyperbolic partial differential equations. It is built on top of Peano 4, which provides the storage and traversal of the simulated domain in the form of a space tree.

One of the most essential applications for ExaHype2 is the simulation of waves, which can be used for studying electromagnetism, earthquakes, and tsunamis. As the latter two often go hand in hand, it is useful to simulate them together to achieve a more accurate representation of the event. Since earthquakes are modeled as elastic waves, whereas tsunamis are modeled as acoustic waves, the need for solvers that can correctly model the interaction between the two mediums arises.

In the paper "A high-order discontinuous Galerkin method for wave propagation through coupled elastic–acoustic media"[15] such a solver was derived.

The goal of this paper is to implement the solver in ExaHype2 and demonstrate its correctness with multiple test scenarios.

Contents

1 Related Work

1.1 ExaHype2

The implementation discussed in this paper is in the context of the ExaHype2 project[9], an open-source engine to solve hyperbolic partial differential equations of the form:

$$\frac{\partial Q}{\partial t} + F(Q, \nabla Q) + B(Q) \cdot \nabla Q = S(Q) + \sum_{i=1}^{n_{ps}} \delta_i \tag{1.1}$$

where Q is a vector of conserved quantity, S the source, and F being the flux function, B being the non-conservative flux function, and δ_i the discrete point sources.

The solution is computed numerically by discretizing the problem in space and time.

The space discretization is handled by the Peano4 adaptive mesh refinement framework, which adaptively divides the domain by tri-partitioning, meaning each cell can be subdivided into $3 \cdot dimension$ cells independently from its neighbors[14]. The storage and retrieval of these cells are also handled in Peano4 by linearizing the domain with space-filling Peano trees, seen in Figure 1.1.

The time discretization is handled directly through the solver, here the so-called ADER method is used[1].

When using ExaHype2, most of the configuration is done in a single specification file, here named *Elastic.py*, which defines[9]:

- Project parameters like name, simulation time, and size of the computational domain
- Solver declaration with ADER-DG, DG, Finite Volumes or Finite Difference as the base. On top of these, the length of the Unknown vector Q has to be specified. Key components of the solver like flux function and eigenvalue computation can be chosen here, alongside with the postprocessing of the solution. Solver specific options e.g. polynomial order in ADER-DG can also be set.
- Peano4 optimization options for example distributed or shared memory parallelisation
- Makefile options like compile flags and included files

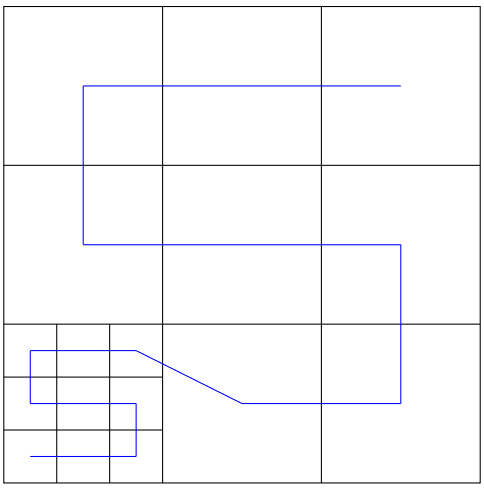


Figure 1.1: Linearization of the domain using Space-filling Peano curves

Executing this configuration file generates several implementation files, including a C++ file, and some C++ template functions, which the user must fill to define the scenario and hyperbolic equation they are simulating.

The implementation is distributed among many files and contains all parts of the numerical solver that are not problem-specific. This includes derivation, integration, updating of the solution, and timestepping of the solver. These files are generated from templates with necessary information about the scenario provided by the specification file.

Any problem-specific handling of the Unknown variables is done in the implementation C++ file, here named *elastic.cpp*, where the flux function, initial state, boundary condition, and source terms are described.

The main C++ file handles the instantiation of the Peano4 space-tree with the domain of specified size and triggers its traversal.

During traversal, the Peano4 sweeps across the entirety of the space-tree, including faces and vertices of all cells, while triggering observers on every move. This is where the generated implementation hooks in, as it is called by these observers, linking Peano4 and the implementation code.

1.2 ADER-DG

As ADER-DG will be used as the base of the solver, the basics of these types of numerical PDE solvers are explained.

Arbitrary high-order DERivatives Discontinuous Galerkin, abbreviated to ADER-DG,

is a method for solving partial differential equations computationally. It combines concepts of the finite elements method and finite volumes method to offer high-order accuracy while being able to model complex geometry[4].

In ADER-DG, each cell is comprised of $(Order + 1)^d$ degrees of freedom, with *Order* being the polynomial order specified in the specification file and $d \in 2/3$(dimensions). Each of these DOFs holds as many values as there are Unknowns Q. These values correspond to the values of the underlying polynomial at pre-defined support points within each cell. They can be interpolated to represent the state at any other point in the cell.

As in classic discontinuous Galerkin methods, the interpolated values at the boundary between two cells do not need to be continuous.

ADER-DG in ExaHype2 follows three core algorithmic steps performed in two Peano4 sweeps over the spacetree[2]:

- Prediction
- Riemann Solve
- Correction

These three steps in computing one timestep are reflected in the Actionsets observers. There are a total of eight of these observers, each attached to the Peano4 traversal and assigned specific jobs necessary for the computation of a new timestep. Relevant for ADER-DG are:

```
observers/TimeStep2exahype2_solvers_aderdg_actionsets_Prediction3.cpp
observers/TimeStep2exahype2_solvers_aderdg_actionsets_Correction6.cpp
```

In the Prediction step, each cell is looked at in isolation from its neighbors. Since the basis functions of the polynomial are known at compile time, mass matrices for derivation and integration can be precomputed. Appropriate support nodes for the polynomial are chosen so that Gaussian quadrature can provide efficient and accurate integrations over the cell. Using these matrices and the user-defined flux function, each cells local solution can be advanced by one timestep.

Since, in the previou step, each cell was solved by itself, the interpolated function is discontinuous when connected at the interface between two cells. To resolve this, a second traversal is triggered. During this sweep, every time a new cell is entered, the shape function is evaluated at the boundary by using a precomputed linear combination of the values stored in the DOF. By solving the Riemann problem for these projected values, the numerical flux between cells is computed.

This numerical flux is used in the third algorithmic step to correct the cell-local solution from step one.

1.3 Parent paper

The implementation done in this thesis is based on results attained in "A high-order discontinuous Galerkin method for wave propagation through coupled elastic–acoustic media"[15].

There, a high-order discontinuous Galerkin scheme for solving wave equations capable of handling material discontinuities, including acoustic-elastic ones, is presented. This scheme was tested using scenarios simulating a variety of waves.

The following chapters detail how these results were reproduced in ExaHype2.

2 Implementation

This chapter details the implementation of the *flux()* and *riemannSolver()* functions in the C++ implementation files mentioned in Chapter 1.1.

In the following, some mathematics needed to understand the later implemented functions and scenarios are explained.

2.1 Prerequisites

The state of the domain is represented in a vector of nine *Unknowns* consisting of:

- $\overline{\sigma} = (\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy}, \sigma_{yz}, \sigma_{xz})^T$ to describe the deformation the wave causes.
- $\overline{v} = (v_x, v_y, v_z)^T$ to describe the speed at which the particles are traveling at.

Augmented by three *Auxiliary Variables*:

- c_p, c_s, ρ to describe the physical property the medium, the wave is traveling on.

The Lamé parameters $\lambda,\ \mu$ together with ρ are an alternative representation to c_p, c_s, ρ for characterizing the medium. They are related by

$$c_p = \sqrt{\frac{\lambda + 2\mu}{\rho}}, \qquad c_s = \sqrt{\frac{\mu}{\rho}} \tag{2.1}$$

The deformation $\overline{\sigma}$ can expressed either as the Cauchy stress tensor $\overline{\overline{S}}$ or the strain tensor $\overline{\overline{E}}$. Due to the inherent symmetry of these deformation tensors, which generally consist of nine elements in 3D, they can be simplified to six elements without loss of information.

They are related by

$$\overline{\overline{S}} = \lambda tr(\overline{\overline{E}})I + 2\mu\overline{\overline{E}} \tag{2.2}$$

This conversion can also be expressed by the fourth-order constitutive tensor C, which will not be further explained here.

$$\overline{\overline{S}} = C\overline{\overline{E}} \tag{2.3}$$

However, in the context of this paper, the conversion (2.3) is equivalent to (2.2) and often easier to write out.

It should be noted that in this thesis and all of the implemented code, the order of iterating over the sigmas $\overline{\sigma}$ from [15] was taken in favor of the default iteration used in other projects of ExaHype2. This change is of a purely cosmetic nature to be consistent with the paper that is reproduced here. As Exahype2 itself does not treat these *Unknowns* differently from each other, changes outside the working directory were not necessary. Problem-specific handling of these *Unknowns* only comes into effect through the user implementation of the *flux()* and *riemannSolver()* functions found in *elastic.cpp*.

In the following, the derivation of these two functions and their implementation is explained.

2.2 Flux

The foundation for the flux function is based on the formulas for linear elasticity that describe the relationship between the strain tensor $\overline{\overline{E}}$ and the velocity $\overline{v}$. [10][15]

$$\frac{\partial \overline{\overline{E}}}{\partial t} = \frac{1}{2}(\nabla \overline{v} + \nabla \overline{v}^T), \qquad \rho \frac{\partial \overline{v}}{\partial t} = \nabla \cdot C\overline{\overline{E}} \tag{2.4}$$

As the choice has been made to represent deformations $\overline{\sigma}$ with the Cauchy stress tensor $\overline{\overline{S}}$ (2.3), the formulation changes to

$$\frac{\partial \overline{\overline{S}}}{\partial t} = C \cdot \frac{1}{2}(\nabla \overline{v} + \nabla \overline{v}^T), \qquad \rho \frac{\partial \overline{v}}{\partial t} = \nabla \cdot \overline{\overline{S}} \tag{2.5}$$

This is done to conform with the later defined *riemannSolver()*.

These equations can then be brought into the the conservative form [11]:

$$\frac{\partial Q}{\partial t} + \nabla \cdot FQ = 0 \tag{2.6}$$

for a conserved quantity Q. In this case these are the *Unknowns* $(\overline{\overline{S}}, \overline{v})^T \in (R^{3\times 3}_{sym}, R^3)^T$ and F being the *flux()* function.

The resulting flux function F can be written as

$$(FQ)_i = \begin{pmatrix} -\frac{1}{2} \cdot C \cdot (\overline{v} \otimes \overline{n}_i + \overline{n}_i \otimes \overline{v}) \\ -\frac{\overline{\overline{S}} \cdot \overline{n}_i}{\rho} \end{pmatrix} for\ i = 1,2,3 \tag{2.7}$$

or if *Unknows* are simplified to a vector

$$(\overline{\overline{S}}, \overline{v})^T = (\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy}, \sigma_{yz}, \sigma_{xz}, v_x, v_y, v_z)^T$$

$$F = - \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & (\lambda + 2\mu) \cdot n_1 & \lambda \cdot n_2 & \lambda \cdot n_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & \lambda \cdot n_1 & (\lambda + 2\mu) \cdot n_2 & \lambda \cdot n_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & \lambda \cdot n_1 & \lambda \cdot n_2 & (\lambda + 2\mu) \cdot n_3 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mu \cdot n_2 & \mu \cdot n_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mu \cdot n_3 & \mu \cdot n_2 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mu \cdot n_3 & 0 & \mu \cdot n_1 \\ \frac{n_1}{\rho} & 0 & 0 & \frac{n_2}{\rho} & 0 & \frac{n_2}{\rho} & 0 & 0 & 0 \\ 0 & \frac{n_2}{\rho} & 0 & \frac{n_1}{\rho} & \frac{n_3}{\rho} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{n_3}{\rho} & 0 & \frac{n_2}{\rho} & \frac{n_3}{\rho} & 0 & 0 & 0 \end{pmatrix} \tag{2.8}$$

with n_1, n_2, n_3 denoting the three dimensions the flux can be computed for.

$$n_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad n_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \qquad n_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Although this looks very different from Equation (7) and (10) from [15] respectively, most of these discrepancies stem from the choice to convert $\overline{\overline{E}}$ to $\overline{\overline{S}}$ when computing the deformations $\overline{\sigma}$ instead of the velocities $\overline{v}$.

Notably different from the flux function derived [15], on the other hand, is the factor of $\frac{1}{\rho}$ in the calculation of the velocity flux. This is because the parent paper[15] does not account for ρ in (2.5) when deriving their flux function.

It is demonstrated later in chapter 3.1.1 that this modified flux yields correct numerical results in ExaHype2.

2.3 Riemann Solver

The Riemann solver derived in the parent paper[15] computes the upwind numerical flux using the exact solution of the Riemann problem. In contrast to other Riemann solvers[12][5], this Riemann solver takes material parameters from both sides of the discontinuity into consideration.

As this paper focuses on the implementation in ExaHype2, the lengthy derivation of the Riemann solver will be omitted. The techniques used can be found[15][4].

The unified numerical flux in direction n for all material interfaces written in the velocity stress formulation is

$$\begin{aligned}\overline{n}\cdot\left(\left(F\begin{pmatrix}\overline{\overline{S}}\\ \overline{v}\end{pmatrix}\right)^{*}-F^{-}\begin{pmatrix}\overline{\overline{S}}^{-}\\ \overline{v}^{-}\end{pmatrix}\right)=&k_0(\overline{n}\cdot[\![\overline{\overline{S}}]\!]+\rho^{+}c_p^{+}[\![\overline{v}]\!])\begin{pmatrix}\lambda^{-}\overline{\overline{I}}+2\mu^{-}\overline{n}\otimes\overline{n}\\ \rho^{-}c_p^{-}\overline{n}\end{pmatrix}\\ &-k_1\begin{pmatrix}2\mu^{-}sym(\overline{n}\otimes(\overline{n}\times(\overline{n}\times[\![\overline{\overline{S}}]\!])))\\ \rho^{-}c_s^{-}\overline{n}\times(\overline{n}\times[\![\overline{\overline{S}}]\!])\end{pmatrix}\\ &-k_1\rho^{+}c_s^{+}\begin{pmatrix}2\mu^{-}sym(\overline{n}\otimes(\overline{n}\times(\overline{n}\times[\overline{v}])))\\ \rho^{-}c_s^{-}\overline{n}\times(\overline{n}\times[\overline{v}])\end{pmatrix}\end{aligned}\tag{2.9}$$

With k_0, k_1 defined as

$$k_0=\frac{1}{\rho^{-}c_p^{-}+\rho^{+}c_p^{+}},\qquad k_1=\begin{cases}\frac{1}{\rho^{-}c_s^{-}+\rho^{+}c_s^{+}} & \mu^{-}\neq 0\\ 0 & \mu^{-}=0\end{cases}\tag{2.10}$$

and

$$[\![\overline{\overline{S}}]\!]=\overline{\overline{S}}^{-}\cdot\overline{n}^{-}+\overline{\overline{S}}^{+}\cdot\overline{n}^{+},\qquad [\![\overline{v}]\!]=\overline{v}^{-}\cdot\overline{n}^{+}+\overline{v}^{+}\cdot\overline{n}^{+},\qquad [\overline{v}]=\overline{v}^{-}-\overline{v}^{+}$$

n denotes the direction of the interface that is being processed and can be one of

$$\begin{pmatrix}1\\0\\0\end{pmatrix},\qquad\begin{pmatrix}0\\1\\0\end{pmatrix},\qquad\begin{pmatrix}0\\0\\1\end{pmatrix}$$

while $sym()$ is defined as

$$sym(\overline{x})=\frac{1}{2}(\overline{x}+\overline{x}^{T}),\qquad sym(\overline{\overline{x}})=\frac{1}{2}(\overline{\overline{x}}+\overline{\overline{x}}^{T})\tag{2.11}$$

The original flux function was designed to compute ρv, which was modified to calculate ρ instead. For this reason, it has been assumed that the Riemann solver also outputs ρv as the formulation of the numerical flux is derived from the flux function.

Therefore every term that makes up the particle speed in the resulting numerical flux has the ρ^{-} factor removed compared to the Riemann solver presented in [15].

$$\begin{aligned}\overline{n}\cdot\left(\left(F\begin{pmatrix}\overline{\overline{S}}\\ \overline{v}\end{pmatrix}\right)^{*}-F^{-}\begin{pmatrix}\overline{\overline{S}}^{-}\\ \overline{v}^{-}\end{pmatrix}\right)=&k_0(\overline{n}\cdot[\![\overline{\overline{S}}]\!]+\rho^{+}c_p^{+}[\![\overline{v}]\!])\begin{pmatrix}\lambda^{-}\overline{\overline{I}}+2\mu^{-}\overline{n}\otimes\overline{n}\\ c_p^{-}\overline{n}\end{pmatrix}\\ &-k_1\begin{pmatrix}2\mu^{-}sym(\overline{n}\otimes(\overline{n}\times(\overline{n}\times[\![\overline{\overline{S}}]\!])))\\ c_s^{-}\overline{n}\times(\overline{n}\times[\![\overline{\overline{S}}]\!])\end{pmatrix}\\ &-k_1\rho^{+}c_s^{+}\begin{pmatrix}2\mu^{-}sym(\overline{n}\otimes(\overline{n}\times(\overline{n}\times[\overline{v}])))\\ c_s^{-}\overline{n}\times(\overline{n}\times[\overline{v}])\end{pmatrix}\end{aligned}\tag{2.12}$$

It should be noted that this has not been verified with a formal rederivation. The correctness of this modification is discussed in chapter 2.6 and demonstrated in chapter 3.4.1.

2.4 Split Riemann solver

When using acoustic equations to model the domain, the medium is characterized by $\mu = 0$, which is equivalent to $c_s = 0$. This enables the Riemann solver (2.12) to be simplified in cases where the interface borders an acoustic cell since many terms evaluate to zero.

In the case of an acoustic-acoustic or acoustic-elastic interface, (2.12) becomes

$$\overline{n} \cdot \left(\left(F \begin{pmatrix} \overline{\overline{S}} \\ \overline{v} \end{pmatrix} \right)^* - F^- \begin{pmatrix} \overline{\overline{S}}^- \\ \overline{v}^- \end{pmatrix} \right) = k_0 (\overline{n} \cdot [\![\overline{\overline{S}}]\!] + \rho^+ c_p^+ [\![\overline{v}]\!]) \begin{pmatrix} \lambda^- \overline{\overline{I}} + 2\mu^- \overline{n} \otimes \overline{n} \\ c_p^- \overline{n} \end{pmatrix} \tag{2.13}$$

And for elastic-acoustic interfaces (2.12) becomes

$$\begin{aligned} \overline{n} \cdot \left(\left(F \begin{pmatrix} \overline{\overline{S}} \\ \overline{v} \end{pmatrix} \right)^* - F^- \begin{pmatrix} \overline{\overline{S}}^- \\ \overline{v}^- \end{pmatrix} \right) = & k_0 (\overline{n} \cdot [\![\overline{\overline{S}}]\!] + \rho^+ c_p^+ [\![\overline{v}]\!]) \begin{pmatrix} \lambda^- \overline{\overline{I}} + 2\mu^- \overline{n} \otimes \overline{n} \\ c_p^- \overline{n} \end{pmatrix} \\ & - \frac{1}{\rho^- c_s^-} \begin{pmatrix} 2\mu^- sym(\overline{n} \otimes (\overline{n} \times (\overline{n} \times [\![\overline{\overline{S}}]\!]))) \\ c_s^- \overline{n} \times (\overline{n} \times [\![\overline{\overline{S}}]\!]) \end{pmatrix} \end{aligned} \tag{2.14}$$

2.5 Implementation in Code

The implemented Riemann Solver and flux function can be found in *riemann.cpp* and *flux.cpp* respectively and are designed to be called from implementation C++ file *elastic.cpp*.

- *riemann.cpp*:

```
riemann_unified(Flux,QL,QR,direction,n);
riemann_split(Flux,QL,QR,direction,n);
riemann_paper(Flux,QL,QR,direction,n);
```

- *flux.cpp*

```
flux(Q,direction,Flux);
flux_paper(Q,direction,Flux);
```

```
#if Dimensions == 2
    const int end = (Order+1);
#elif Dimensions == 3
    const int end = (Order+1)*(Order+1);
#endif
    for(int i = 0; i < end; i++){
        riemann_unified(FL+9*i, QL+12*i, QR+12*i, direction, 1);
        riemann_unified(FR+9*i, QR+12*i, QL+12*i, direction, -1);
    }
```

Figure 2.1: Code in *elastic.cpp* to call the Riemann solver

The functions with *_paper* suffix use the equations derived in the parent paper[15, (7)(22)], while all the others use the modified flux and Riemann solver mentioned here (2.7)(2.12).

Two methods of coupling the acoustic and elastic domains are suggested in ExaHype2. The first is to create two different solvers that traverse separate domains overlapping in a boundary domain that couples the two domains together.

The other method is to create a single unified solver for the entire domain and store all the *Unknowns*, needed for both the acoustic and elastic domain, in every cell.

Although some of the data stored in the second approach might be unnecessary, it is generally faster since only one mesh traversal is needed[6].

Coupling of acoustic-elastic domains using a unified solver is especially efficient since the *Unknowns* needed for the acoustic equation are a subset of the *Unknowns* of the elastic equation.

The implemented *riemann_unified()* uses the unified equation applicable to any material interface(2.12), while *riemann_split()* chooses the most suitable equation out of (2.12)(2.13)(2.14), depending on the material parameters at the interface.

The implemented Riemann solvers compute the numerical *Flux* for two states *QL*, *QR* in $direction \cdot n$. To get the corresponding *Flux* in the reverse direction, the function has to be called again with *QL* and *QR* swapped and $direction \cdot 1$ reversed to $direction \cdot -1$. Due to the nature of ExaHype2, this procedure has to be repeated along the entire interface that is being processed. This means for any polynomial order N, the computation of the numerical flux has to be repeated for $N+1$ pairs of *QL* and *QR* if using 2D. When simulating in 3D, the number of pairs grows to $(N+1)^2$.

2.6 Correctness of the modified Riemann Solver

Having removed the ρ^- factor in the Riemann solver without formal justification, new questions arise:

Does the modified Riemann solver (2.12) still correctly account for material parameters on both sides of the discontinuity? What if the correct change would have been to leave the ρ^- factor in and instead to divide by ρ^+?

The thought process behind dividing by ρ^- instead of ρ^+ is the following:

Under the assumption that the original Riemann solver (2.9) computed ρv^- in the flux affecting Q^-, in order to calculate v^- the solver should divide by ρ^-, the density value of Q^- itself.

When computing the flux in the reverse direction for Q^+ the same is done for v^+ because of the interchanging of parameters during the call, seen in Figure 2.1. Therefore, both densities ρ^-, ρ^+ from both sides have an effect on the resulting flux for their respective cells.

A test supporting this decision can be found in Chapter 3.4.1.

2.6.1 Boundary Conditions

For the well posedness of the system, boundary conditions are needed to define the behavior at the edge of the computational domain. There are three types of boundaries which will be used in this work:

- The periodic boundary is a feature of ExaHype2 and is set in the Python specification file *Elastic.py*. It works by connecting the two ends of the computational domain together to create a circular domain.
- The analytical boundary only works for scenarios for which the exact solution is known, such as the scenarios implemented in Chapter 3. It sets the value of the boundary to the correct analytical solution of the wave at that position and time. This simulates a domain that extends infinitely in the direction of the boundary. However, even small inaccuracies or deviations of the simulation from the analytical solution can lead to reflections and instability at the boundary.
- The traction-free boundary simulates an interface with a vacuum on the other side. Since vacuum is incapable of transferring stresses $\overline{\sigma}$, these have to be zero at the boundary. To achieve this, the stresses $\overline{\sigma}$ of the boundary are set to be the negative of the $\overline{\sigma}$ of the neighboring cell.

3 Scenarios

To demonstrate the correctness of the implementation, scenarios with well-known analytical solutions were implemented. These are chosen to cover all parts of the solver while only featuring displacements along two of the three axes in order to work in both 2D and 3D.

All scenarios share a common interface in *scenario.h*:

```
boundary(Qinside,Qoutside,position,t,normal);
initial(Q,position,with_params,t);
```

This interface is called from *elastic.cpp* and currently has four working implementations in *plane_scenario.cpp*, *rayleigh_scenario.cpp*, *lamb_scenario.cpp* and *scholte_scenario.cpp*. The *boundary()* function is called by *elastic::boundaryConditions()*, while *initial()* returns the analytically computed state at any position $\overline{x}$ and time t in the Q parameter and is called by *elastic::initialCondition()* and *elastic::analyticalSolution()*. The *with_params* parameter controls whether the material parameters $c_p,\ c_s,\ \rho$ should be included in Q as these do not change over time. Therefore *elastic::analyticalSolution()* does not need them and also has no memory allocated to store these parameters.

The scenario can be set in the *Elastic.py* file by changing the *Scenario* Enum. This handles the inclusion of the correct required file and sets the offset and boundaries to the values needed for the scenarios to work.

Scenarios are defined in terms of their analytically calculated displacement in time t and space $\overline{x} = \{x, y, z\}^T$: $u_x(\overline{x}, t)$, $u_y(\overline{x}, t)$ and material parameters $\rho(\overline{x}),\ c_p(\overline{x}),\ c_s(\overline{x})$ which are constant within a cell and do not change in time. Derivating $u_x,\ u_y$ in time gets us the velocities v_x, v_y.

The strain tensor $\overline{\overline{E}}$ is defined by

$$\overline{\overline{E}} = \frac{1}{2}(\nabla\overline{u} + \nabla\overline{u}^T)$$

This can then be converted into the Cauchy Stress Tensor $\overline{\overline{S}}$ using Lamé parameters λ, μ

$$\overline{\overline{S}} = 2\mu\overline{\overline{E}} + \lambda tr(\overline{\overline{E}})\overline{\overline{I}}$$

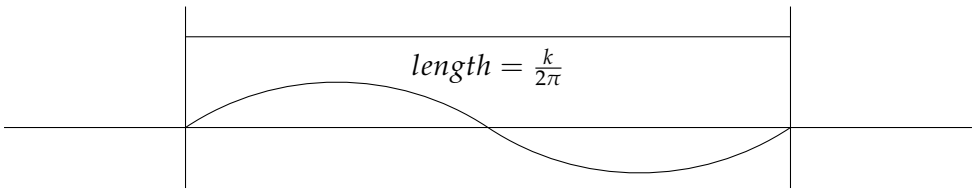


Figure 3.1: Wavelength in relation to the wavenumber k

The state vector $\overline{\overline{Q}}$ can then be expressed as

$$\sigma_{xx}(\overline{x},t) = (\lambda + 2\mu)\frac{\partial u_x}{\partial x} + \lambda\frac{\partial u_y}{\partial y}, \qquad \sigma_{yy}(\overline{x},t) = (\lambda + 2\mu)\frac{\partial u_y}{\partial y} + \lambda\frac{\partial u_x}{\partial x}$$

$$\sigma_{zz}(\overline{x},t) = \lambda(\frac{\partial u_y}{\partial y} + \frac{\partial u_x}{\partial x}), \qquad \sigma_{xy}(\overline{x},t) = \mu(\frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x})$$

$$\sigma_{yz}(\overline{x},t) = 0, \qquad \sigma_{xz}(\overline{x},t) = 0$$

$$v_x(\overline{x},t) = \frac{\partial u_x}{\partial t}, \qquad v_y(\overline{x},t) = \frac{\partial u_y}{\partial t}, \qquad v_z(\overline{x},t) = 0$$

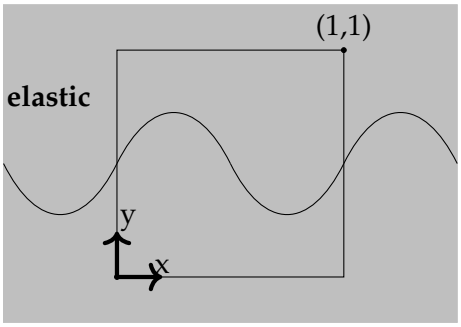
In all scenarios, the wavenumber $k = 2\pi$ is chosen. This way, the wavelength, the distance after the wave returns to its starting state, is exactly one, see Figure 3.1. When periodic boundaries are applied in the direction the wave travels on a domain of $length = 1$ in that direction, a periodic wave is achieved. This wave behaves as if the same wave is repeated on an infinite domain.

In scenarios where the domain is divided into two areas of different material parameters, it is split at $y = \frac{2}{3}$, as this is a cell boundary shared by all refinement levels, which guarantees that material parameters remain constant within each cell. This is necessary because only the Riemann solver is capable of solving at material discontinuities.

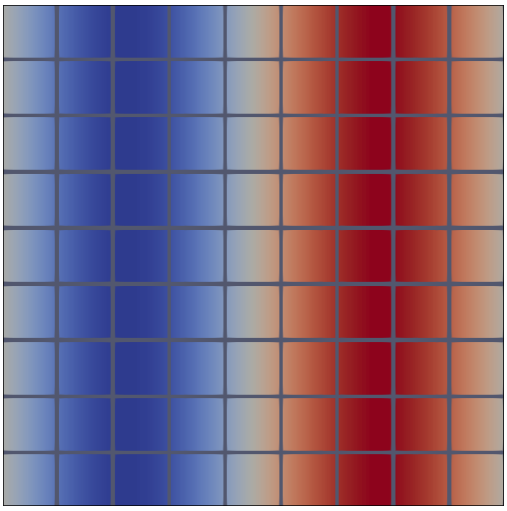
3.1 Plane Wave

Plane waves consist of longitudinal and transverse waves that travel in a fully periodic elastic domain. A sketch of the scenario can be seen in Figure 3.2a. This scenario has been implemented to test the correctness of the flux function and, later, the Riemann solver. It was chosen because it is described with very simple displacements:

$$u_x(\overline{x},t) = cos(k \cdot (x_0 - c_p t)) \qquad u_y(\overline{x},t) = cos(k \cdot (x_0 - c_s t))$$



(a) Sketch of Plane wave scenario



(b) Initial condition of σ_{yy} visualized in Paraview

Figure 3.2: Plane wave scenario

The material parameters used are $\lambda = 2.2$, $\mu = 1.3, \rho = 1.2$. The domain spans $[0, 1] \times [0, 1]$ in $[x] \times [y]$ dimension split in 9×9 cells with periodic boundaries in all dimension.

3.1.1 L2-Errors of different solvers

Using this exact basic scenario as a test, the differences between the solvers of the parent paper[15] and their modified versions implemented here(2.12)(2.7) are shown.

The two solver components, the Riemann solver and the flux function, are tested independently. The *riemann_split()* function has been left out since it is functionally the same as *riemann_unified()*. The remaining components have been combined to create four solvers:

- *flux_paper()* + *riemann_paper()*
- *flux_paper()* + *riemann_unified()*
- *flux()* + *riemann_paper()*
- *flux()* + *riemann_unified()*

In Figure 3.3 the L2-Error is plotted as the polynomial order of the simulation is increased. It can be seen that the L2-Error in this test case is mainly influenced by the

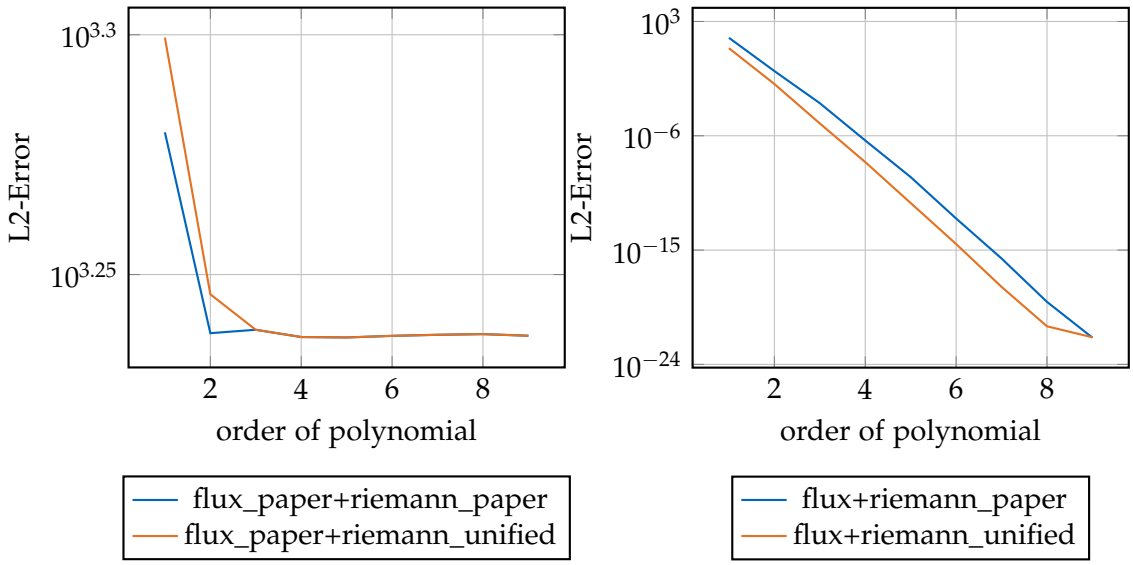


Figure 3.3: L2 Error integrated over the domain of different solvers as the order of the polynomial is increased, Plane wave scenario at t=1, 9×9 Domain

choice of the flux function. Between the two flux functions, the errors of the solvers using the flux from the parent paper do not converge toward the correct solution.

To determine which Riemann solver implementation is correct, the level of refinement is increased to create domains of 27×27 cells, to increase the influence of the Riemann solvers. Since the difference between the two solvers is how the density ρ is treated, instead of plotting the L2-Error against polynomial order, the error as the density increases is visualized in Figure 3.4

As can be seen, the L2-Error of the solver using the modified Riemann solver stays constant as the density is increased, whereas the error of the other solver increases steadily until it spikes. At $\rho = 1.7$ it already increases to $1.55112e + 298$. The sudden jump is due to the error introduced into each cell from the Riemann solver through the numerical flux, reaching a magnitude that significantly affects the interface on the opposite side of the cell. This then gets amplified in the next timestep, resulting in escalation of the error.

In the following, the solver used will always be the modified *flux()+riemann_unified()*.

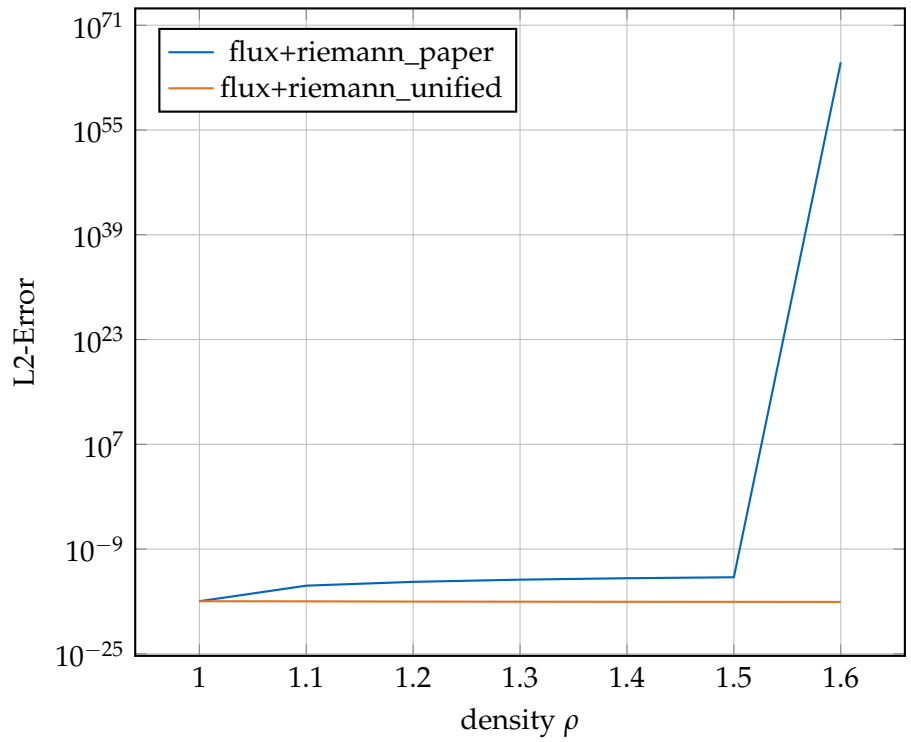


Figure 3.4: L2-Error integrated over the domain of the Plane wave scenario as density ρ is increased at $t = 1$ with Order=5, 27×27 cells comparing *riemann_unified* and *riemann_paper* 2.5

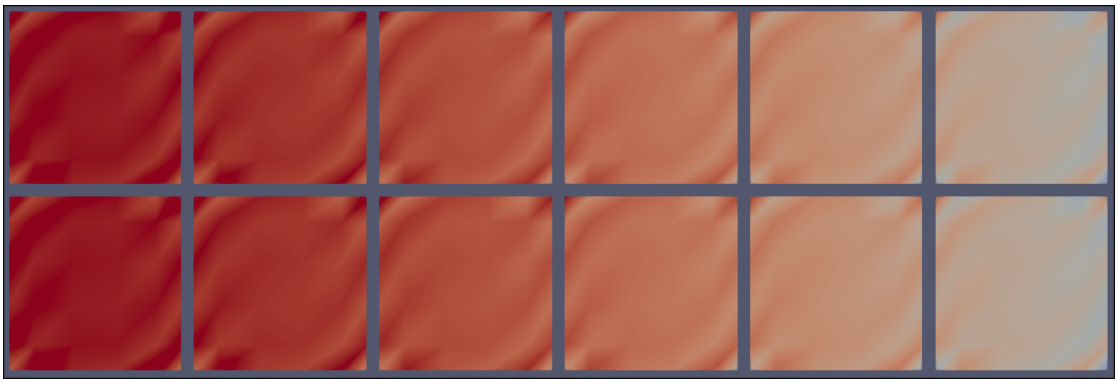


Figure 3.5: Error spreading from the cell interfaces in Plane wave scenario Order=5, 27×27 cells starting at density $\rho = 1.6$ when using *riemann_paper* 2.5

3.2 Rayleigh Wave

The Rayleigh waves are surface waves traveling along the boundary between an elastic medium and a vacuum. They find wide application in seismology in modeling earthquakes. This scenario is included to show the correctness of the traction-free boundary alongside the elastic-elastic Riemann solver. Similar scenarios have been implemented in [3, Section 4.4.1][15, Section 6.3]. The displacement of the wave is described as

$$u_x(\overline{x},t) = [A_1 e^{b_1 x_3} + A_2 e^{b_2 x_3}] cos(k(x_1 - c_r t))$$
$$u_y(\overline{x},t) = [\frac{b_1}{k} A_1 e^{b_1 x_3} + \frac{k}{b_2} A_2 e^{b_2 x_3}] sin(k(x_1 - c_r t))$$

with $\rho = \lambda = \mu = 1$
c depends on the chosen material parameters and satisfies

$$(2 - \frac{c^2}{c_s^2})^2 - 4(1 - \frac{c^2}{c_p^2})^{\frac{1}{2}}(1 - \frac{c^2}{c_s^2})^{\frac{1}{2}}$$

It was computed to be $c = \sqrt{2 - \frac{2}{\sqrt{3}}}$.
The relation between A_1 and A_2 is described by

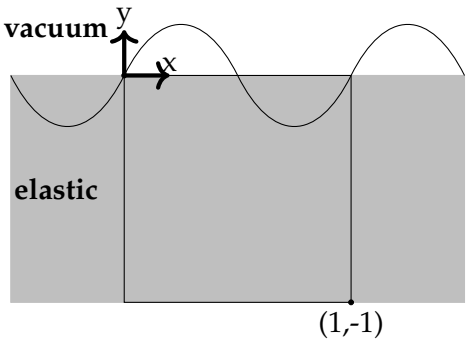
$$(2 - \frac{c^2}{c_s^2})A_1 + 2A_2 = 0$$

$A_2 = 1$ sets $A_1 = -\sqrt{3}$
b_1, b_2 are driven by

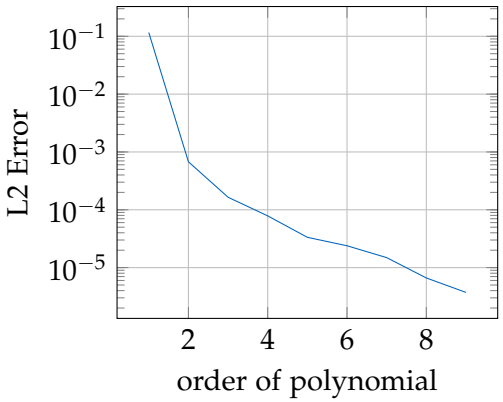
$$b_1 = k(1 - \frac{c_r^2}{c_p^2})^{\frac{1}{2}}, \qquad b_2 = k(1 - \frac{c_r^2}{c_s^2})^{\frac{1}{2}}$$

The computational domain is $[0,1] \times [-1,0]$ in $x \times y$ divided into 9×9 cells of equal size. A sketch of it can be seen in Figure 3.6a. The top boundary at $y = 0$ is implemented as a traction-free boundary to simulate the vacuum, while the bottom boundary at $y = -1$ is set as an analytical boundary. This does not noticeably impact the accuracy of the simulation because the magnitude of the wave decays away from $y = 0$.

The L2-Error of the simulation at $t = 1$ is measured. As can be seen in Figure 3.6b the error decreases exponentially as the poynomial order of the solver is increased.



(a) Sketch of Rayleigh wave scenario



(b) L2 Error integrated over the domain of Rayleigh wave scenario as the order of the polynomial is increased at t=1, 9×9 Domain

Figure 3.6: Rayleigh wave scenario

3.3 Lamb Wave

Lamb waves propagate along an infinite plane of elastic medium of finite thickness $2d$ surrounded by a vacuum. This scenario is similar to the Rayleigh one and demonstrates the vacuum boundary together with the elastic-elastic Riemann solver in the absence of a boundary that enforces the analytical solution upon the computational domain. A sketch can be seen in Figure 3.7a

The Lamb Wave is described in [15, Section 6.4]

$$u_x(\bar{x},t) = (-kAcos(py) - qBcos(qy))sin(kx - \omega t)$$
$$u_y(\bar{x},t) = (-kAsin(py) + qBsin(qy))cos(kx - \omega t)$$

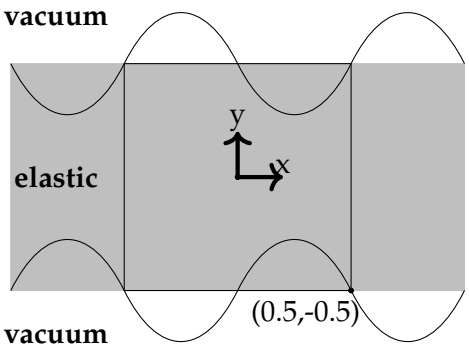
with ω driven by the chosen material parameters and wavenumber k

$$\frac{tan(qd)}{tan(pd)} = \frac{4k^2pq}{(q^2-k^2)^2}$$

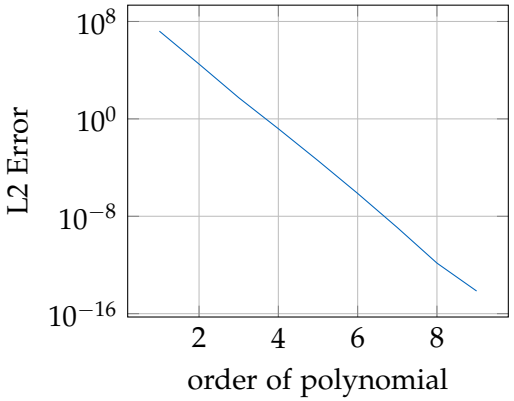
and

$$p^2 = \frac{\omega^2}{c_p^2} - k^2, \qquad p^2 = \frac{\omega^2}{c_s^2} - k^2$$

$\omega = 13.13706319723$ has been taken from [15]



(a) Sketch of Lamb wave scenario



(b) L2 Error integrated over the domain of Lamb wave as the order of the polynomials is increased scenario at t=1, 9×9 Domain

Figure 3.7: Lamb wave scenario

and A and B can be computed with

$$A = 2\mu kq \cdot cos(qd), \qquad B = (\lambda k^2 + (\lambda + w\mu)p^2)cos(pd)$$

here $\lambda = 2$, $\mu = \rho = 1$, $\omega = 13.13706319723$, $A = 126.1992721468$, $B = 53.88807700007$ were used.

The domain spans $[-0.5, 0.5] \times [-0.5, 0.5]$ and is refined two times into a 9×9 cell grid.

Both the top and bottom are implemented as traction-free boundaries, while all the other sides are periodic boundaries to act like an infinite repeating plane.

As can be seen in Figure 3.7b, the Lamb scenario starts with a very high L2 error compared to the Rayleigh one. This can be attributed to the high range of values that need to be represented within one cell. This error decreases exponentially as the order is increased.

3.4 Scholte Wave

Scholte waves propagate along the boundary between acoustic and elastic domains. They are especially interesting as they occur in underwater seismology at the boundary between water and seabed[13]. This scenario is implemented to highlight the Riemann solver's capabilities to handle acoustic-elastic boundaries.

The displacements for the upper acoustic domain $y > 0$ are:

$$u_x(\overline{x},t) = Re((ikB_1e^{-kb_{2p}x_3} + kb_{2s}B_2e^{-kb_{2s}x_3})e^{i(kx_1-\omega t)})$$
$$u_y(\overline{x},t) = Re((-kb_{2p}B_1e^{-kb_{2p}x_3} + ikB_2e^{-kb_{2s}x_3})e^{i(kx_1-\omega t)})$$

and for the lower elastic half $y < 0$:

$$u_x(\overline{x},t) = Re((ikB_3e^{kb_{2p}x_3} - kb_{2s}B4e^{kb_{2s}x_3})e^{i(kx_1-\omega t)})$$
$$u_y(\overline{x},t) = Re((kb_{2p}B_3e^{kb_{2p}x_3} + ikB_4e^{kb_{2s}x_3})e^{i(kx_1-\omega t)})$$

with $\lambda_1 = \rho_1 = 1,\ \mu_1 = 0,$ and $\lambda_2 = \mu_2 = \rho_2 = 1$.
c depends on the chosen material parameters

$$(\frac{\rho_1}{\rho_2}b_{2p} + b_{1p})r^4 - 4b_{1p}r^2 - 4b_{1p}(b_{2p}b_{2s} - 1) = 0$$

with

$$b_{1p} = (1 - \frac{c^2}{c_{1p}^2})^{\frac{1}{2}}, \quad b_{1s} = (1 - \frac{c^2}{c_{1s}^2})^{\frac{1}{2}}, \quad b_{2p} = (1 - \frac{c^2}{c_{2p}^2})^{\frac{1}{2}},\ b_{2s} = (1 - \frac{c^2}{c_{2s}^2})^{\frac{1}{2}}$$

$$c_{1p} = \sqrt{\frac{\lambda_1 + 2\mu_1}{\rho_1}}, \quad c_{1s} = \sqrt{\frac{\mu_1}{\rho_1}}, \quad c_{1p} = \sqrt{\frac{\lambda_2 + 2\mu_2}{\rho_2}}, \quad c_{2s} = \sqrt{\frac{\mu_2}{\rho_2}}$$

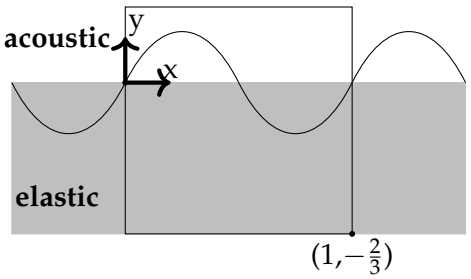
Here $c = 0.7110017230197$ has been determined.
A derivation for $B_1,\ B_2,\ B_3,\ B_4$ can be found in [15, Section 6.5]
In this case $B_1 = -i0.3594499773037,\ B_2 = -i0.8194642725978,\ B_3 = i0.5220044931212,\ B_4 = -0.9339639688697$ were used in the simulation.

3.4.1 Material discontinuities with modified Riemann Solver

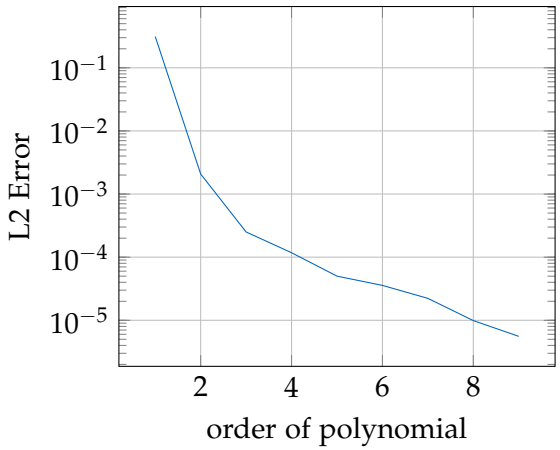
Although the Scholte scenario features an acoustic-elastic interface, both domains have identical density $\rho = 1$. This leaves one crucial part of the modified Riemann solver untested: the handling of material interfaces with different densities ρ.

To verify the correctness of the solver for material discontinuities with different densities ρ, a scenario simulating a Stoneley wave is implemented. This wave is a generalization of the Scholte wave and represents a wave traveling along the interface between two domains of arbitrary material parameters.

Unfortunately, as of now, this scenario does not run correctly, as the L2-Error does not converge to zero for increasing polynomial orders. Nontheless running the scenario simulates a wave with the aforementioned material interface at $y = \frac{2}{3}$.



(a) Sketch of Scholte wave scenario



(b) L2 Error integrated over the domain of Scholte wave scenario as order of the polynomial is increased at t=1, 9×9 Domain as the order of the polynomial is inreased

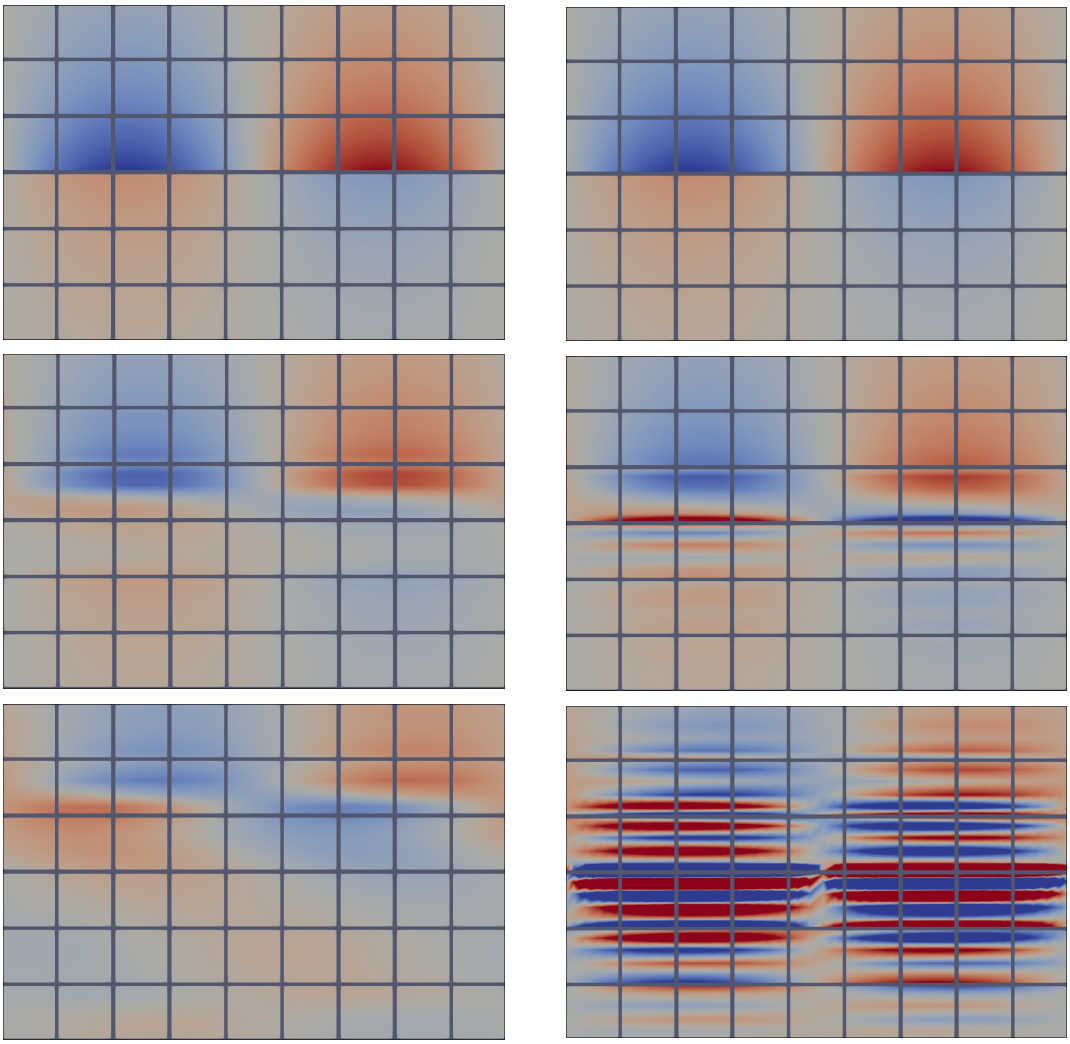
Figure 3.8: Scholte wave scenario

Being a generalization of the Scholte scenario, the computational domain is the same spanning $[0,1] \times [-\frac{2}{3}, \frac{1}{3}]$ divided into 9×9 cells with polynomial order 5, seen in Figure 3.8a.

Since the scenario is not correctly implemented, a detailed description of the scenarios is omitted here. A complete definition can be found in [15].

To answer the question posed in chapter 2.6 the Stoneley scenario is run on two different Riemann solvers. One using the formulation detailed in (2.12), here called ρ^--solver. The other one is modified by dividing with ρ^+ instead of ρ^- as discussed in chapter 2.6, here called ρ^+-solver.

Figure 3.9 shows the visual results of these simulations cropped to only show the top $\frac{2}{3}$ portion. As can be seen, both start with the same initial condition. However, the solution computed with ρ^+-solver has strong instabilities expanding alternatingly from the material discontinuity. This indicates that the ρ value from the wrong side is being taken when computing the numerical flux, causing an error, which gets repeated in the next timestep, causing the alternating pattern. The ρ^--solver, on the other hand, produces sensible output.



(a) ρ^- Riemann solver

(b) ρ^+ Riemann solver

Figure 3.9: Behavior at at elastic-elastic interface between domains of different densities ρ comparing ρ^- and ρ^+ Riemann solvers. Visualized using Paraview at t = 0.0, 0.1, 0.3 using Stoneley wave scenario

3.5 Results

To determine the correctness of the flux and Riemann solver four scenarios representing various waves were implemented. With tests using different densities on *plane_scenario* and *stoneley_scenario*, the modifications to the solver of the parent paper were justified. L2-Error convergence towards zero has been observed in four scenarios. Especially proper handling of the acoustic-elastic boundary is demonstrated in the Scholte Scenario.

One noticeable result that can be seen in the L2-Errors, is different types of convergence towards zero. While the L2-Error in the Lamb and Plane scenarios approach zero almost exactly exponentially, the other scenarios converge more linearly. These two scenarios with no analytical boundaries also approaches zero a lot faster. It is possible that the analytical boundary, by being slightly more accurate, causes reflections that interfere with the stability of the solution.

4 Evaluation

4.1 Performance

In this chapter, the performance of the implemented Riemann solvers is measured and compared against the default Riemann solver of the ExaHype2 project, the Rusanov Riemann solver. This approximate solver computes the numerical flux at the discontinuity between two cells by taking averages of both sides and applying the *flux()* function onto them.

The performance has been measured with *perf stat -r10 ./EXECUTABLE* on an Intel i7-14700KF, 32GB RAM, with the executable being compiled with *gcc -O3*. The writing of patch files and measurement of errors have been disabled. The performance was only measured in single-core mode since the Riemann Solver does not transmit any data nor influences the parallelization of the code.

As can be seen in Figure 4.1, the execution time grows exponentially as the order of the polynomials increases. The computational effort of most actions in the pipeline, mentioned in chapter 1.2, grows linearly with the number of nodes. Since $NumberOfNodes = Order^2$ in 2D, this growth in execution time makes sense. The plotting of the performance of the Rusanov and Split solver have been omitted, as they were within 1% of the result of the unified solver.

4.1.1 Profiling

Despite vast differences between the Rusanov solver and the two other solvers, the execution time of all three Riemann solvers are practically identical. To study this in more detail, the code has been profiled with

```
valgrind --tool=callgrind --dump-instr=yes --simulate-cache=yes
    --collect-jumps=yes --collect-atstart=no --instr-atstart=no ./EXECUTABLE
```

and with a set of Macros inserted into the main C++ file as described[8] to only collect calls made during the timestepping phase of the program execution, see Figure 4.2.

When compiled with $Order = 5$ on 9×9 domain, Figure 4.3 shows where most of the time is spent. All "Time spent" values have been taken from *KCachegrind* "Self" value

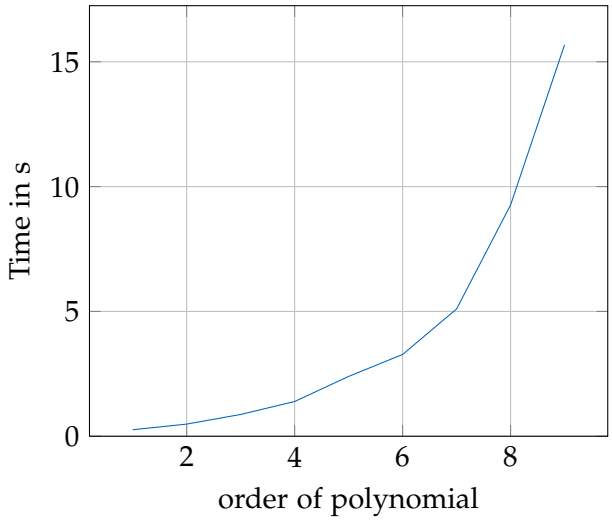


Figure 4.1: Execution time running the Scholte wave scenario as order of the polynomial is increased, end time = 1, 9×9 cells

```
#include <valgrind/callgrind.h>
int main()
{ CALLGRIND_START_INSTRUMENTATION;
  CALLGRIND_TOGGLE_COLLECT;

  //Code for computing a TimeStep

  CALLGRIND_TOGGLE_COLLECT;
  CALLGRIND_STOP_INSTRUMENTATION;}
```

Figure 4.2: Code used in the main C++ function to capture calls for profiling

with the sole exception of *riemannSolver()*, where the "Incl." value is taken, since the solver only calls a single helper function. This way, "Time spent" accurately reflects the time it took to compute the numerical flux by combining the time of *riemannSolver()* and its helper function.

Label	Time spent %	Function
A	22.01	generated::kernels::AderDG::linear:: fusedSpaceTimePredictorVolumeIntegral()
B	9.86	peano4::datamanagment::FaceMarker::FaceMarker()
C	8.87	_memset_avx2_unaligned_erms
D	3.64	examples::exahype2::elastic::observers::TimeStep::enterCell()
E	3.63	examples::exahype2::elastic::observers::TimeStep::leaveCell()
F	3.08	examples::exahype2::elastic::elastic::flux()
...	...	...
G	0.57	examples::exahype2::elastic::elastic::riemannSolver()

Figure 4.3: Profiling Order 5 Depth 2

Most of the time is spent in **A**, which makes sense since this function computes the per-cell prediction solution, mentioned in chapter 1.2. It is also the only caller of **C**, which is used to set large memory areas to zero.

B is called exclusively out of the TimeStep observer and handles the storage and retrieval of cell values.

Although **D** and **E** themselves do not compute much, they are responsible for managing and calling the whole "Actionset" pipeline. This is reflected in a high "Incl." value of 51.07% of **D**, which means a total of 51.07% of the entire runtime is spent in functions **D** calls, including itself.

A surprisingly high amount of time is spent in the flux function **F**. Despite the computation happening in there being trivially simple, it is being called by **A** repeatedly numerous times to compute the flux for all nodes in a cell. It is possible that the *USE_IPO* variable could improve the performance here by inlining the call to the *flux()* function, which then could be vectorized by the compiler. This, however has not been profiled.

As expected, when decreasing the order of the polynomials to one, the time spent in **A** and its related functions **C** and **F** are drastically decreased. Although, as a result, the other functions gain in percentage, the *riemannSolver()* function remains very low in usage time.

Label	Time spent %	Function
B	16.95	peano4::datamanagment::FaceMarker::FaceMarker()
D	5.80	examples::exahype2::elastic::observers::TimeStep::enterCell()
E	5.79	examples::exahype2::elastic::observers::TimeStep::leaveCell()
...	...	...
A	1.08	generated::kernels::AderDG::linear:: fusedSpaceTimePredictorVolumeIntegral()
...	...	...
G	0.30	examples::exahype2::elastic::elastic::riemannSolver()
...	...	...
F	0.24	examples::exahype2::elastic::elastic::flux()
...	...	...
C	0.06	_memset_avx2_unaligned_erms

Figure 4.4: Profiling Order 1 Depth 2

4.1.2 Riemann Solver Performance

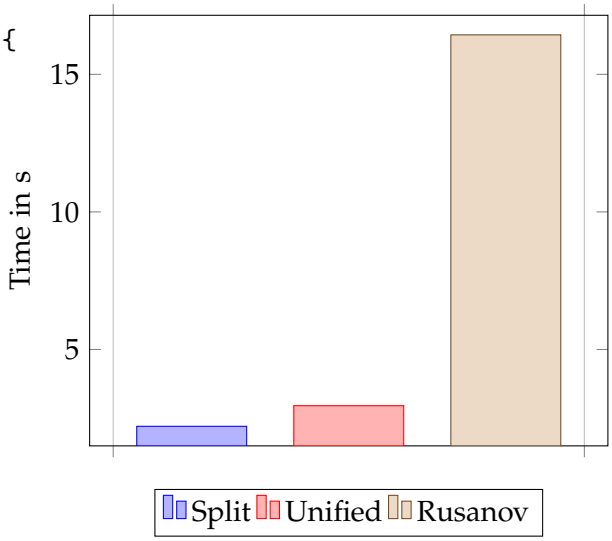
To properly measure the performance of the three Riemann solvers, a simple test function that can be injected into the main function has been written, Figure 4.5a. It tries to emulate the material parameters found in an actual scenario with an acoustic-elastic interface at $y = \frac{2}{3}$ with acoustic being on top and elastic on the bottom. The probability at which three types of Riemann discontinuities are chosen to match their occurencce rate of the aforementioned scenario. Since the type of discontinuity only depends on $QL[cs]$ and $QR[cs]$, all other values of QL and QR are randomized.

As can be seen in Figure 4.5b, the implemented Riemann solvers outperform the default Rusanov solver by quite a bit. This is due to the computation in the new solvers being a lot simpler because they are specially derived for this very Riemann problem. Additionally, the Split solver is noticeably faster than the Unified solver because it can save a lot of time when calculating numerical fluxes for acoustic-acoustic Riemann problems.

Although these improvements in performance are barely measurable when used in ADER-DG, they are very significant in a finite volume approach. However, it should be noted that, in contrast to the new Riemann solvers, the Rusanov solver in the state it is used in ADER-DG can not be used in finite volumes.

```
QL = rand();
QR = rand();
for(int i = 0; i < 100000000; i++){
    long j = rand();
    if(j\%100 < 33){
        QL[11] = 0;
        QR[11] = 0;
    }else if(j\%100 < 40){
        QL[11] = 1;
        QR[11] = 0;
    }else{
        QL[11] = 1;
        QR[11] = 1;
    }
    riemannSolver(Ql,QR);
}
```

(a) Pseudocode used to test the performance of the Riemann Solvers



(b) Performance of different Riemann solvers compared, 100000000 repeated calls

Figure 4.5: Performance of Riemann Solvers

4.2 Consistency

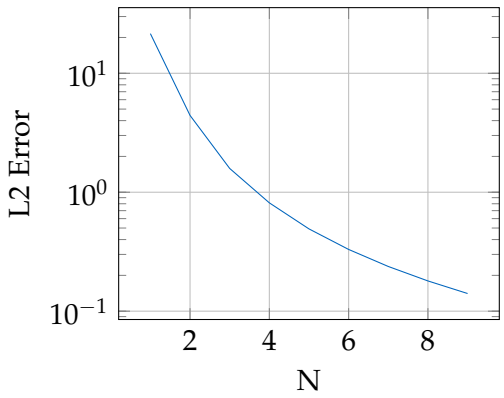
A noticeable property of the implemented Riemann solver is that it takes material properties from both sides of the discontinuity when computing the numerical flux. The correctness of this solver has been proven in [15, Section 4] and demonstrated with the scenarios in Chapter 3.

Although the Rusanov solver is able to reproduce all scenarios in a meaningful way, the L2-Error decreases magnitudes slower compared to the other two solvers, see Figure 4.6b. Apart from that, even at $Order = 9$, there is a "gap" that forms at boundaries and material interfaces, see Figure 4.6c.

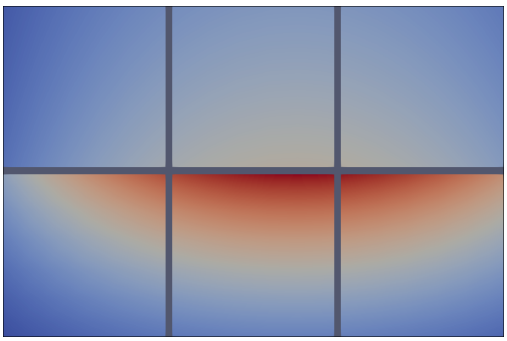
According to [15, Section 4.1], the consistency of a Riemann solver is given, if the computed numerical flux is zero when applied to a continuous solution. This is the case if the projected values of the cells are the same on both sides of the discontinuity. However, different material interfaces put different constraints on which values have to be continuous.

```
for(int i = 0; i < 1000; i++){
    QL = rand();
    QR = QL;
    QL[v+1] = 1; QR[v+2] = 2;
    QL[v+1] = 1; QR[v+2] = 2
    QL[cs] = 1; QL[cs] = 0;
        riemannSolver(Ql,QR);
    }
```

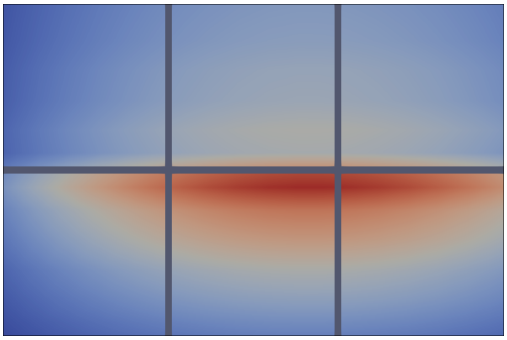
(a) Pseudocode used to test the Correctness of the Riemann Solvers



(b) L2 Error integrated over the domian of Scholte wave scenario at t=1, 9×9 Domain with Rusanov Solver



Expected result generated using implemented Riemann solver



Result generated by Rusanov Riemann solver

(c) Differences at acoustic-elastic interface of Scholte wave scenario

Figure 4.6: Differences between Riemann Solvers

For elastic-elastic interface in direction n:

$$\overline{v}^+ = \overline{v}^- \qquad \overline{\overline{S}}^+ \cdot \overline{n} = \overline{\overline{S}}^- \cdot \overline{n}$$

For acoustic-elastic and acoustic-acoustic interfaces in direction n:

$$\overline{n} \cdot \overline{v}^+ = \overline{n} \cdot \overline{v}^- \qquad \overline{\overline{S}}^+ \cdot \overline{n} = \overline{\overline{S}}^- \cdot \overline{n}$$

Noticeably, it is not required for the tangential component of v to be continuous, when there is an acoustic cell at the interface.

A test function has been implemented to test this property of the Riemann solvers, see Figure 4.6a, by calling the Riemann solvers with $QL = QR$ that have different

tangential $v[1]$, $v[2]$ values. An acoustic-elastic interface is simulated by setting one of the $Q[cs]$ values to zero.

The test verifies the property of consistency for the implemented Riemann solver, while showing that the Rusanov solver does not have that property.

Strangely, this "gap" appears not only at acoustic-elastic interfaces but also at the vacuum boundary at the top of the Rayleigh wave scenario, where no differences in tangential velocities occur. Although the reason for this gap has not been found, it is suspected that it appears due to the Rusanov solver taking the average of the cell states when calculating the numerical flux.

5 Future Work

5.1 Additional scenarios

Out of the six scenarios presented in the parent paper[15], only four are working correctly in ExaHype2. The missing Stoneley and Snells law scenarios are both very interesting as they feature an acoustic-elastic boundary with different densities on each side, which the correctly implemented scenarios lack. This is also very likely the reason why they do not work, as the modifications to the flux function and Riemann solver were not made until well into the work. Currently, the two missing scenarios are implemented but do not work correctly.

5.2 Correctness

As it stands, the modified Riemann solver has only been tested for discontinuities at mediums with different densities. This could be improved with a formal rederivation of the Riemann solver from the modified flux function. Additionally, the stability of the modified Riemann solver has not been proven.

5.3 Comparison

A performance comparison with the same acoustic-elastic solver running on separate domains would have been interesting. This way, the performance overhead of coupling two separate solvers and their respective domains could have been analyzed.

6 Conclusion

During this work on coupled acoustic-elastic solvers, the architecture and workflow of ExaHype2 and the underlying Peano4 were familiarized.

In order to implement a solver, the ADER-DG method and its implementation within ExaHype2 was studied.

Using equations derived in the paper "A high-order discontinuous Galerkin method for wave propagation through coupled elastic–acoustic media"[15] an ADER-DG solver, capable of handling acoustic-elastic boundaries, was implemented in ExaHype2.

Following the steps for deriving a flux function, the flux function was modified to achieve correct numerical results in ExaHype2. Since the Riemann solver is derived from the flux functions according changes were made to how the numerical flux is calculated. This Riemann solver was split up to handle all interface types, acoustic-acoustic, acoustic-elastic, and elastic-elastic efficiently.

By implementing scenarios modeling Plane, Rayleigh, Lamb, and Scholte waves the correctness of the modified acoustic-elastic solver was demonstrated for all material interfaces.

Using the Plane scenarios with different densities ρ, the changes to the flux were justified by comparing different combinations of flux functions and Riemann solvers.

Numerical results exponentially converging towards the exact analytical solution were observed for these four scenarios.

The performance of the entire execution with different parameters was profiled using *valgrind* and analyzed.

The performance of the optimized solver was measured and compared against the standard solver and the default Rusanov Riemann solver.

The consistency of the implemented Riemann solver at material interfaces was demonstrated and contrasted against the Rusanov solver.

7 Notes

All the implementation is done on the DG branch of Peano4[7] on commit 17b8979f. Slight modification to *python/exahype2/solvers/aderdg/kernels.py* were made on that commit to fix a bug, see Figure 7.1. The implementation is on the basis of the Elastic example found in *applications/exahyp2/aderdg/Elastic*.

```
+++ b/python/exahype2/solvers/aderdg/kernels.py
@@ -1011,8 +1011,8 @@ def add_final_action_for_error_measurement():
   repositories::{{SOLVER_INSTANCE}}.Order,
   repositories::{{SOLVER_INSTANCE}}.QuadratureWeights,
   repositories::{{SOLVER_INSTANCE}}.QuadraturePoints1d,
-  repositories::{{SOLVER_INSTANCE}}.NumberOfVariables,
-  repositories::{{SOLVER_INSTANCE}}.NumberOfParameters,
+  repositories::{{SOLVER_INSTANCE}}.NumberOfUnknowns,
+  repositories::{{SOLVER_INSTANCE}}.NumberOfAuxiliaryVariables,
   fineGridCell{{UNKNOWN_IDENTIFIER}}.value,
   errors
   );
```

Figure 7.1: Bugfix in *kernels.py*

List of Figures

Bibliography

[1] "ADER Schemes for Nonlinear Systems of Stiff Advection–Diffusion–Reaction Equations." In: (). DOI: https://doi.org/10.1007/s10915-010-9426-6.

[2] *ExaHype2 - ADER-DG solvers*. https://hpcsoftware.pages.gitlab.lrz.de/Peano/html/db/dae/page_exahype_solvers_aderdg.html. accessed 1.02.2024.

[3] K.-A. Feng, C.-H. Teng, and M.-H. Chen. "A Pseudospectral Penalty Scheme for 2D Isotropic Elastic Wave Computations." In: *Journal of Scientific Computing* 33.3 (Dec. 2007), pp. 313–348. ISSN: 1573-7691. DOI: 10.1007/s10915-007-9154-8.

[4] T. W. Jan S. Hesthaven. *Nodal Discontinuous Galerkin Methods*. 1st ed. Springer New York, NY. ISBN: 978-0-387-72065-4. DOI: https://doi.org/10.1007/978-0-387-72067-8.

[5] M. Käser and M. Dumbser. "A highly accurate discontinuous Galerkin method for complex interfaces between solids and moving fluids." In: *Geophysics* 73.3 (Mar. 2008), T23–T35. ISSN: 0016-8033. DOI: 10.1190/1.2870081. eprint: https://pubs.geoscienceworld.org/geophysics/article-pdf/73/3/T23/3214094/gsgpy_73_3_T23.pdf.

[6] *Peano4 - Coupling of various solvers and additional mesh traversals*. https://hpcsoftware.pages.gitlab.lrz.de/Peano/html/d4/de8/page_exahype_coupling.html. accessed 1.02.2024.

[7] *Peano4 Git Repository*. https://gitlab.lrz.de/hpcsoftware/Peano. accessed 01.02.2024.

[8] *Profiling with Valgrind*. https://developer.mantidproject.org/ProfilingWithValgrind.html. accessed 01.02.2024.

[9] A. Reinarz, D. E. Charrier, M. Bader, L. Bovard, M. Dumbser, K. Duru, F. Fambri, A.-A. Gabriel, J.-M. Gallard, S. Köppel, L. Krenz, L. Rannabauer, L. Rezzolla, P. Samfass, M. Tavelli, and T. Weinzierl. "ExaHyPE: An engine for parallel dynamically adaptive simulations of wave problems." In: *Computer Physics Communications* 254 (2020), p. 107251. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2020.107251.

[10] W. S. Slaughter. *The Linearized Theory of Elasticity*. 1st ed. Birkhäuser Boston, MA. ISBN: 978-0-8176-4117-7. DOI: https://doi.org/10.1007/978-1-4612-0093-2.

[11] E. F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Springer Berlin, Heidelberg. DOI: https://doi.org/10.1007/978-3-662-03490-3.

[12] H. Wang, M. Cosnefroy, and M. Hornikx. "An arbitrary high-order discontinuous Galerkin method with local time-stepping for linear acoustic wave propagation." In: *The Journal of the Acoustical Society of America* 149.1 (Jan. 2021), pp. 569–580. ISSN: 0001-4966. DOI: 10.1121/10.0003340. eprint: https://pubs.aip.org/asa/jasa/article-pdf/149/1/569/14139845/569_1_online.pdf.

[13] S. Wege, C. Legendre, W.-C. Chi, T. Wang, P. Kunath, and C.-S. Liu. "Field and Synthetic Waveform Tests on Using Large-Offset Seismic Streamer Data to Derive Shallow Seabed Shear-Wave Velocity and Geotechnical Properties." In: *Earth and Space Science* 9 (June 2022). DOI: 10.1029/2021EA002196.

[14] T. Weinzierl and M. Mehl. "Peano—A Traversal and Storage Scheme for Octree-Like Adaptive Cartesian Multiscale Grids." In: *SIAM Journal on Scientific Computing* 33.5 (2011), pp. 2732–2760. DOI: 10.1137/100799071. eprint: https://doi.org/10.1137/100799071.

[15] L. C. Wilcox, G. Stadler, C. Burstedde, and O. Ghattas. "A high-order discontinuous Galerkin method for wave propagation through coupled elastic–acoustic media." In: *Journal of Computational Physics* 229.24 (2010), pp. 9373–9396. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2010.09.008.