Comparison of Cartesian and Dynamically Adaptive Grids for the Parallel Simulation of Shallow Water Waves

Jaclyn Rodrigues Monteiro
Comparison of Cartesian and Dynamically Adaptive Grids for the Parallel Simulation of Shallow Water Waves

Vergleich kartesischer und dynamisch adaptiver Gitter bei der parallelen Simulation von Flachwasserwellen

Author: Jaclyn Rodrigues Monteiro
Supervisor: Univ.-Prof. Dr. Michael Bader
Advisor: Dipl.-Inf. Oliver Meister
Submission Date: February 16, 2015
I confirm that this bachelor’s thesis is my own work and I have documented all sources and material used.

Garching, February 16, 2015
Jaclyn Rodrigues Monteiro
Abstract

The Shallow Water Equations can be used for tsunami simulation. The Chair of Scientific Computing at Technische Universität München offers two codes both able to perform this, sam(oa)$^2$ and SWE. Both use the same approach, but different grids for domain discretisation. In this work we compare these two grids by implementing a benchmark based on a Riemann problem. For this Riemann problem the analytic solution is known, allowing us to quantify the error of both programs when simulating it. It shows, that sam(oa)$^2$ achieves smaller errors when using the same amount of cells as SWE due to its dynamically adaptive grid. While achieving this result we performed an optimisation of sam(oa)$^2$ to improve the grid refinement. This improvement shows its limits though. Finally, we compare a tsunami simulation by sam(oa)$^2$ to real measured data, showing that the simulation is close to reality while such a validation remains difficult at the same time.
1 Introduction

In various sciences computational simulation has become an important aspect. While prior virtual trials can be performed before an actual experiment, it also allows creation of hazardous scenarios which cannot (or should not) be reproduced in field experiments. Examples are the air flow around a sports car or the simulation of tsunamis.

The field of high performance computing covers simulations as it tries to perform those in the least possible time by using thousands of processors in parallel. In the past the number of cores used in supercomputers has been increasing, allowing simulations to use more resources and therefore giving better and/or faster results. For some scenarios this is even necessary, as their computation might be too expensive otherwise.

Simulation of catastrophes like earthquakes or tsunamis has relevance beyond scientific interests; sufficiently fast computation of possible aftermath upon detection of triggering events might allow to avoid casualties.

Any simulation program that intends to forecast or reproduce some behaviour needs to give adequate results. Revision of these unfortunately is not always possible; analysis on how well basic features work might be feasible.

The Chair of Scientific Computing of the Technische Universität München explores the field of simulation and high performance computing. Next to various other projects it offers two simulation softwares for shallow water waves and tsunamis, sam(oa)$^2$ and SWE, which use slightly different approaches. sam(oa)$^2$ uses a dynamically adaptive triangular grid, while SWE uses a non-adaptive Cartesian one. Both build on the Shallow Water Equations and come with certain advantages and disadvantages.

We will focus on a way to measure the accuracy of both programs when simulating a basic scenario, for which the true solution is known. Such a scenario is given by a Riemann problem, which has also been used for error analysis in [3] and [7]. Therefore the two codes received extensions, for sam(oa)$^2$ an optimisation was performed in addition. As the dynamically adaptive grid can be restricted to being a non-adaptive grid, we compare three set-ups in total. In the end, we also investigate how well sam(oa)$^2$ reproduces an actual tsunami scenario.

In section 2, the mathematical foundations SWE and sam(oa)$^2$ are built on is given, as well as the theory needed to perform the evaluation of their simulation adequacies. The following section gives some insight into the realisation of both codes and how to compare the major different component by using a benchmark. It also covers the implementations necessary for this comparison and presents some first results for sam(oa)$^2$, as well as an optimisation due to these first results. Section 4 then compares the benchmark results of SWE and modified sam(oa)$^2$ and analyses the tsunami simulation performed by sam(oa)$^2$. The results are summarised in section 5.
2 Theoretical Foundations

This chapter intends to provide the reader with the theoretical knowledge necessary for the actual target of this work, no recent scientific achievements are presented here though.

2.1 Notation

To avoid misunderstandings, some of the used notation is defined here.

For the partial derivative, there will be two notations used, the classic version using $\partial$ and a subscript one:

$$\frac{\partial}{\partial x} f(x, y) = f_x(x, y).$$

The general derivative of a function $f$ is denoted as

$$f'.$$

Vectors will be denoted by

$$\vec{x}$$

and their scalar product is

$$\langle \vec{x}, \vec{y} \rangle.$$  

2.2 The Shallow Water Equations

The Shallow Water Equations are based on the idea of a fluid with negligible vertical velocities. As the name suggests, we consider shallow water with waves of small amplitude and water depth being of small magnitude compared to the wavelength.

The derivation is based on the conservation laws for mass and momentum and might be easier to understand when first looking only at the one-dimensional case.

For an incompressible fluid (such as water) the density $\bar{\rho}$ is constant, the water height $h(x, t)$ might vary. On a one-dimensional domain the total mass in $[x_1, x_2]$ at time $t$ is

$$\int_{x_1}^{x_2} \bar{\rho} h(x, t) \, dx.$$  

(2.1)

The mass in $[x_1, x_2]$ changes only via in- or outflow through the ends of the domain; so with $+F_1(t)$ and $-F_2(t)$ the fluxes into the domain at $x_1$ resp. $x_2$ at time $t$, we receive the conservation of mass

$$\frac{d}{dt} \int_{x_1}^{x_2} \bar{\rho} h(x, t) \, dx = F_1(t) - F_2(t).$$  

(2.2)

For $u(x, t)$ denoting the velocity at $x$ at time $t$, at each point the density of momentum is given by $\bar{\rho} u(x, t)$, giving the mass flux $\bar{\rho} u(x, t) h(x, t)$. Rewriting the latter using $p(x, t) = h(x, t) u(x, t)$ and cancelling the $\bar{\rho}$ yields

$$\frac{d}{dt} \int_{x_1}^{x_2} h(x, t) \, dx = p(x_1, t) - p(x_2, t) = -p(x, t) \bigg|_{x_1}^{x_2}.\]

(2.3)
2.2 The Shallow Water Equations

Adding some partial differentiation on the right side and rewriting the differentiation on the left gives us

\[ \int_{x_1}^{x_2} \frac{\partial}{\partial t} h(x,t) \, dx = - \int_{x_1}^{x_2} \frac{\partial}{\partial x} p(x,t) \, dx \]  
(2.4)

or

\[ \int_{x_1}^{x_2} \left[ \frac{\partial}{\partial t} h(x,t) \, dx + \frac{\partial}{\partial x} p(x,t) \right] \, dx = 0. \]  
(2.5)

Since this is to hold for all \( x_1 \) and \( x_2 \), the integrand has to be zero, so we receive

\[ \frac{\partial}{\partial t} h(x,t) \, dx + \frac{\partial}{\partial x} p(x,t) = 0 \]  
(2.6)

or in short notation, replacing the \( p \) and denoting partial differentiation with subscript:

\[ h_t + (hu)_x = 0 \]  
(2.7)

Besides mass, momentum is also a conserved quantity. Again, we set up an equation to model change of momentum through fluxes. The momentum is given by \( \rho h (x, t) u(x, t) \).

The momentum flux past any point \( x \) is given by two components, first is the momentum carried by the fluid which flows past, which is momentum times velocity: \( \rho hu \cdot u = \rho hu^2 \). The second component is momentum created by hydrostatic pressure; at distance \( h - y \) from the surface the weight of the fluid above applies the pressure \( \rho g (h - y) \) (with \( g \) gravitational constant). The pressure \( q \) at \( (x, t) \) then is given by vertical integration from \( y = 0 \) to \( y = h(x, t) \): \( q = \frac{1}{2} \rho gh^2 \). Thus, conservation of momentum becomes

\[ \frac{d}{dt} \int_{x_1}^{x_2} \bar{\rho} h(x,t) u(x,t) \, dx = \bar{\rho} h (x_1,t) u^2 (x_1,t) + \frac{1}{2} \bar{\rho} gh^2 (x_1,t) \]

\[ - \bar{\rho} h (x_2,t) u^2 (x_2,t) - \frac{1}{2} \bar{\rho} gh^2 (x_2,t), \]  
(2.8)

repetition of the above procedure yields

\[ (hu)_t + (hu^2 + \frac{1}{2}gh^2)_x = 0. \]  
(2.9)

Together, (2.7) and (2.9) form the system of one-dimensional Shallow Water Equations:

\[ \begin{bmatrix} h \\ hu \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \end{bmatrix}_x = 0 \]  
(2.10)

However, we are mainly interested in the two-dimensional case with \( u \) the velocity in \( x \)-direction and \( v \) the velocity in \( y \)-direction. The conservation of mass in one dimension (2.2) then becomes

\[ \frac{d}{dt} \int_{y_1}^{y_2} \int_{x_1}^{x_2} \bar{\rho} h(x,y,t) \, dx \, dy = - \left( \int_{y_1}^{y_2} \bar{\rho} (hu)(x,y,t) \, dy \right) \bigg|_{x_1}^{x_2} \]

\[ - \left( \int_{x_1}^{x_2} \bar{\rho} (hv)(x,y,t) \, dy \right) \bigg|_{y_1}^{y_2}. \]  
(2.11)
Again we repeat the steps from the beginning of this section, this time also benefiting from switching order of integration, giving us

\[ h_t + (hu)_x + (hv)_y = 0. \] (2.12)

In two dimensions momentum is carried along in both directions, so its conservation has to be considered in both directions as well. Therefore, in \( x \)-direction we have not only the term \( huu \), but also the term \( hu \); conservation of momentum is given by

\[ (hu)_t + \left( hu^2 + \frac{1}{2}gh^2 \right)_x + (huv)_y = 0, \]

\[ (hv)_t + (huv)_x + \left( hv^2 + \frac{1}{2}gh^2 \right)_y = 0. \] (2.13)

So the system of two-dimensional Shallow Water Equations is

\[
\begin{bmatrix}
  h \\
  hu \\
  hv
\end{bmatrix}_t + \begin{bmatrix}
  hu \\
  hu^2 + \frac{1}{2}gh^2 \\
  hv
\end{bmatrix}_x + \begin{bmatrix}
  hv \\
  huv \\
  hv^2 + \frac{1}{2}gh^2
\end{bmatrix}_y = 0
\] (2.14)

Up to now we did not consider that any domain filled with water might not have an even floor, e.g. the ocean’s floor, which is also called bathymetry. This gives us source terms on the right-hand side which describe the effect of such a bathymetry \( b \):

\[ h_t + (hu)_x + (hv)_y = 0 \]

\[ (hu)_t + \left( hu^2 + \frac{1}{2}gh^2 \right)_x + (huv)_y = -ghb_x \] (2.15)

\[ (hv)_t + (huv)_x + \left( hv^2 + \frac{1}{2}gh^2 \right)_y = -ghb_y. \]

An irregular floor which does not change form over time does not affect the total mass, since the volume is fixed. For this reason the first equation of (2.15) remains untouched.

\[ [4] \]

2.3 Riemann Problem

For a hyperbolic system, a Riemann problem is given by the system’s equation and a piecewise constant function as initial data. For our purposes, this function will model a single jump discontinuity. For the one-dimensional Shallow Water Equations it looks like follows:

\[ h(x, 0) = \begin{cases} 
  h_l & \text{if } x < 0 \\
  h_r & \text{if } x > 0
\end{cases}, \quad u(x, 0) = 0, \]

(2.16)

with \( h_l > h_r \geq 0 \) and \( u_l = u_r = 0 \). The bathymetry is set to zero, \( b = 0 \). As it models the breaking of a dam separating two different water heights, this is also referred to as dam-break problem. Over time, the water will expand from the dam position; in the centre
it will form an even level of intermediate water height $h_m$, which connects with the higher water level $h_l$ via a so-called rarefaction wave; in this area, the water height will decrease with time and therefore the flow rarefies. Intermediate water height and lower water level $h_r$ will be connected by a shock wave, a travelling jump discontinuity. See figure 1 for an example visualisation.

![Figure 1: Left: Riemann problem at $t_0 = 0$. Right: Riemann problem at some time $t_1 > t_0$; intermediate state, rarefaction and shock wave have formed.](image)

This problem can be solved analytically. In the general case $h_l, h_r > 0$ the solution consists of one rarefaction and one shock wave. Define

$$
\Phi_{\text{rare}} (h_1, h_2) := 2 \left( \sqrt{gh_1} - \sqrt{gh_2} \right),
\Phi_{\text{shock}} (h_1, h_2) := (h_1 - h_2) \sqrt{\frac{g}{2}} \left( \frac{1}{h_1} - \frac{1}{h_2} \right).
$$

(2.17)

And therefore set with $u_l, u_r \in \mathbb{R}$:

$$
\Phi_l (h) := u_l + \begin{cases} 
\Phi_{\text{rare}} (h_l, h) & \text{if } h < h_l \\
\Phi_{\text{shock}} (h_l, h) & \text{otherwise}
\end{cases}
\Phi_r (h) := u_r + \begin{cases} 
\Phi_{\text{shock}} (h, h_r) & \text{if } h > h_r \\
\Phi_{\text{rare}} (h, h_r) & \text{otherwise}
\end{cases}
$$

(2.18)

Then for the water height $h_m$ of the intermediate state must hold: $\Phi_l (h_m) = \Phi_r (h_m)$ with $\Phi_l (h_m) = \Phi_r (h_m) =: u_m$ the intermediate state velocity. [4] This equation can be solved by finding the root of $\Phi := \Phi_r - \Phi_l$, using e.g. the Newton method:

**Newton method** for finding the root of $\Phi (h)$:

$h^{(0)}$ start value

for $k = 1, 2, \ldots$

$$
h^{(k+1)} := h^{(k)} - \frac{\Phi (h^{(k)})}{\Phi'(h^{(k)})}
$$

This Riemann problem thus can be used to simulate a dam-break and compare the resulting behaviour with the behaviour given by the analytic solution.
2.4 The Finite Volume Method

In computer science, simulation of a constant domain is (currently) not possible, a discrete
domain has to be used alternatively. Therefore we subdivide the domain in some fashion
and represent the parts by finite volumes, also called grid cells. Instead of tracking our
quantity of interest \( h \) at every point of the domain, each cell carries the integral of \( h \)
on the respective sub-domain (\( h \) does not need to refer to a water height as in the previous
subsection, but can refer to any other quantity here). In one dimension

\[
C_i = \left( x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}} \right)
\]

shall denote the \( i \)-th grid cell with its left border \( x_{i-\frac{1}{2}} \) and its right border \( x_{i+\frac{1}{2}} \). Then at
time \( t_n \) the value \( H_i^n \) is the approximate average value in the \( i \)-th grid cell:

\[
H_i^n \approx \frac{1}{\Delta x} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} h(x, t_n) \, dx \equiv \frac{1}{\Delta x} \int_{C_i} h(x, t_n) \, dx.
\]

(2.20)

with cell length \( \Delta x = x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}} \) (assume a uniform grid at first, i.e. all cells are of
the same length). Reconsider (2.3): The time derivative of the total amount of \( h \) on any
domain \([x_1, x_2]\) is only changed by fluxes \( f \) through \( x_1 \) and \( x_2 \):

\[
\frac{d}{dt} \int_{x_1}^{x_2} h(x,t) \, dx = f \left( h(x_1, t) \right) - f \left( h(x_2, t) \right).
\]

(2.21)

This is also referred to as integral form of a conservation law. For grid cell \( C_i \) this becomes

\[
\frac{d}{dt} \int_{C_i} h(x,t) \, dx = f \left( h \left( x_{i-\frac{1}{2}}, t \right) \right) - f \left( h \left( x_{i+\frac{1}{2}}, t \right) \right).
\]

(2.22)

So far we set up an equation to describe how \( h \) changes over time in any grid cell. As
already mentioned above, we have to discretise; this also holds for time. A time step of
length \( \Delta t = t_{n+1} - t_n \) then shall give us approximate cell averages \( H_i^{n+1} \) at time \( t_{n+1} \)
based on the cell average \( H_i^n \) at time \( t_n \); integrating (2.22) over one time step yields

\[
\int_{C_i} h(x,t_{n+1}) \, dx - \int_{C_i} h(x,t_n) \, dx = \int_{t_n}^{t_{n+1}} f \left( h \left( x_{i-\frac{1}{2}}, t \right) \right) \, dt \\
- \int_{t_n}^{t_{n+1}} f \left( h \left( x_{i+\frac{1}{2}}, t \right) \right) \, dt,
\]

(2.23)

or with rearranging and division by \( \Delta x \)

\[
\frac{1}{\Delta x} \int_{C_i} h(x,t_{n+1}) \, dx = \frac{1}{\Delta x} \int_{C_i} h(x,t_n) \, dx - \frac{1}{\Delta x} \int_{t_n}^{t_{n+1}} f \left( h (x_1, t) \right) - f \left( h (x_2, t) \right) \, dt.
\]

(2.24)

We have seen the right-hand side before: (2.24) tells us how to perform an update of
(2.20); still, we do not know the time integral in (2.24), since we do not know how
to evaluate \( h(x,t) \) - otherwise the whole discussion of discretisation and approximation
would be pointless. Instead, we approximate the average flux over time \( \Delta t \) at some point
\( x_i \):
2.4 The Finite Volume Method

\[ F^n_i \approx \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} f(h(x_i, t)) \, dt \]  \hspace{1cm} (2.25)

so (2.24) becomes with (2.25) and (2.20) on cell \( C_i \)

\[ H^{n+1}_i = H^n_i - \frac{\Delta t}{\Delta x} \left( F^n_{i+\frac{1}{2}} - F^n_{i-\frac{1}{2}} \right). \]  \hspace{1cm} (2.26)

Any problem given by a system of differential equations

\[ \frac{\partial}{\partial t} q(x,t) + A \cdot \frac{\partial}{\partial x} q(x,t) = 0 \]  \hspace{1cm} (2.27)

with \( q: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}^m \) a quantity vector and \( A \) a constant real \( m \times m \)-Matrix is called

hyperbolic, if \( A \) has real eigenvalues and \( m \) eigenvectors which are linearly independent; the Shallow Water Equations pose such a problem. The properties of such problems allow
us to use

\[ F^n_{i-\frac{1}{2}} = F(H^n_{i-1}, H^n_i), \]  \hspace{1cm} (2.28)

where \( F \) is a numerical flux function, which approximates the flow from one cell to another. For a hyperbolic problem information travels at limited speed (i.e. nothing happens in one cell that immediately affects all other cells), so we suppose we can obtain the flux \( F^n_{i-1/2} \) from \( H^n_{i-1} \) to \( H^n_i \) based on just those two values. (2.26) becomes

\[ H^{n+1}_i = H^n_i - \frac{\Delta t}{\Delta x} \left[ F(H^n_i, H^n_{i+1}) - F(H^n_{i-1}, H^n_i) \right], \]  \hspace{1cm} (2.29)

which means that the update of \( H^n_i \) is based on its own and the two neighbouring values. So far we have discretised our domain to cells \( C_i \) and determined to compute the quantity average there over time via (2.29). This leaves us with the task to find the numerical flux function \( F \). [4]

2.4.1 Fluxes

Any used flux function is useless if it does not converge towards the true solution. Convergence is given, when a method is stable and consistent. It is consistent if it approximates the differential equation well locally. It is stable if small errors do not grow too fast while time-stepping.

Imagine \( h(x, t) \equiv \bar{h} \) is constant in \( x \); it will not change with time and the integral over \( f(h(x_i, t)) \) in (2.25) reduces to \( f(\bar{h}) \). \( \bar{h} \) is constant in \( x \), so \( H^n_{i-1} = H^n_i = \bar{h} \) and therefore we receive from (2.28):

\[ f(\bar{h}) = F(\bar{h}, \bar{h}). \]  \hspace{1cm} (2.30)

So if \( H_{i-1}, H_i \to \bar{h} \), we expect that \( F(H_{i-1}, H_i) \to f(\bar{h}) \); this is usually wrapped up in a Lipschitz condition:

\[ \exists L : |F(H_{i-1}, H_i) - f(\bar{h})| \leq L \cdot \max(|H_i - \bar{h}|, |H_{i-1} - \bar{h}|) \]  \hspace{1cm} (2.31)

A necessary condition which has to be satisfied by any finite volume method is the CFL condition. The idea is easiest to understand for a constant flux in one direction; this applies to (2.7) with constant velocity \( \bar{u} \):
This is also called *advection equation* and describes a single wave: \( h \) simply translates with speed \( \bar{u} \), and covers a distance \( \bar{u}\Delta t \) in one time step. Suppose this distance is larger than one grid cell, or expressed in a formula, \( \bar{u}\Delta t > \Delta x \) - suppose it is actually one and a half grid cells. Basing the flux at \( x_{i-1/2} \) only on the two neighbouring cell averages \( H^n_{i-1} \) and \( H^n_i \) as (2.28) suggests, would be insufficient. At time \( t_{n+1} \) the value at \( x_{i-1/2} \) is actually the value of \( x_{i-2} \) at time \( t_n \), but the cell average covering this region, \( H^n_{i-2} \), is not included in our flux function. Therefore we have to demand that

\[
\nu = \frac{|\bar{u}\Delta t|}{\Delta x} \leq 1. \tag{2.33}
\]

Where \( \nu \) is the so-called *Courant number*. We are treating hyperbolic systems of differential equations though, where we have instead of a single constant velocity \( \bar{u} \) a matrix \( A \) as described above. This system can actually be reduced to a set of \( m \) propagating waves with respective speeds \( \lambda_1, \ldots, \lambda_m \) (the eigenvalues of \( A \)) and the Courant number becomes

\[
\nu = \frac{\Delta t}{\Delta x} \max_p |\lambda_p|. \tag{2.34}
\]

For a flux based on the direct neighbours, this again has to be \( \leq 1 \). We will not go further into stability here; the CFL condition already helps to keep things in order.

So how do we define the flux at \( x_{i-1/2} \), based on \( H^n_{i-1} \) and \( H^n_i \)? A simple approach is using the arithmetic average

\[
\mathcal{F}(H^n_{i-1}, H^n_i) = \frac{1}{2} \left[ f\left( H^n_{i-1} \right) + f\left( H^n_i \right) \right], \tag{2.35}
\]

so the cell update becomes

\[
H^{n+1}_i = H^n_i - \frac{\Delta t}{2\Delta x} \left[ f\left( H^{n+1}_{i+1} \right) - f\left( H^{n}_{i-1} \right) \right]. \tag{2.36}
\]

Unfortunately this turns out to be unstable for hyperbolic problems in general. An improvement is posed by the *Lax-Friedrichs method*:

\[
H^{n+1}_i = \frac{1}{2} \left( H^n_{i-1} + H^n_{i+1} \right) - \frac{\Delta t}{t\Delta x} \left[ f\left( H^{n+1}_{i+1} \right) - f\left( H^{n}_{i-1} \right) \right] \tag{2.37}
\]

which is stable if \( \nu \leq 1 \). However, this method was found to deliver smeared results due to too much diffusion.

There is another possibility to look at the problem of neighbouring cells with different averages, see figure 2; this should look familiar, as it depicts that the neighbouring cells can be seen as a Riemann problem as shown in figure 1.

As discussed in section 2.3, an analytic solution exists for this. Therefore, instead of computing an equalising flow from one cell to another, one might actually solve the local Riemann problem. [4]
2.4 The Finite Volume Method

\[ H_{i-1} \]
\[ H_i \]
\[ x_{i-3/2} \quad x_{i-1/2} \quad x_{i+1/2} \]
\[ C_{i-1} \quad C_i \]

Figure 2: The different cell averages in two neighbouring cells can be seen as a Riemann problem.

2.4.2 Solvers

In [2], the authors introduced a method for spatially varying flux functions, of which we will only look at the basic concept. Instead of treating a one dimensional hyperbolic equation of the form

\[ h_t + f (h) \cdot x = 0 \quad (2.38) \]

(which corresponds to (2.7) and (2.27), and is also referred to as autonomous system), with \( h(x,t) \in \mathbb{R}^m \) again a vector of some conserved quantity and flux function \( f: \mathbb{R}^m \rightarrow \mathbb{R}^m \), the flux function is allowed to be spatially dependent as well:

\[ h_t + f (h, x) \cdot x = 0. \quad (2.39) \]

One can approximate the fluxes on the cell edge using an approximate Jacobian matrix \( A_{i-1/2} \) and replace (2.38) with

\[ h_t + A_{i-1/2} h_x = 0. \quad (2.40) \]

For the autonomous case of \( m \) equations the solution of the Riemann system in general consists of \( m \) superposing waves \( \mathcal{W}_{i-1/2}^p \) \( (p = 1, \ldots, m) \) with respective propagation speeds \( s_{i-1/2}^p \). These speeds are the eigenvalues of \( A_{i-1/2} \), their respective eigenvectors are \( r_{i-1/2}^p \). We assume that the difference between cells - now seen as a Riemann problem shock discontinuity - is given by such superposing waves:

\[ H_i - H_{i-1} = \sum_{p=1}^{m} \mathcal{W}_{i-1/2}^p = \sum_{p=1}^{m} \alpha_{i-1/2}^p r_{i-1/2}^p. \quad (2.41) \]

where

\[ \alpha_{i-1/2}^p = R_{i-1/2}^{-1} (H_i - H_{i-1}) \quad (2.42) \]

with \( R_{i-1/2} \) the matrix of right eigenvectors. Key feature of the introduced method is, instead of using a decomposition in the style of (2.41), to decompose the wave via the flux:

\[ f_i (H_i) - f_{i-1} (H_{i-1}) = \sum_{p=1}^{m} \beta_{i-1/2}^p r_{i-1/2}^p = \sum_{p=1}^{m} \beta_{i-1/2}^p \quad (2.43) \]
with a new coefficient

\[ \beta_{i-1/2} = R_{i-1/2}^{-1} (f_i (H_i) - f_{i-1} (H_{i-1})) . \]  (2.44)

The \( \mathcal{Z}^p = \beta^p r^p \) were named \textit{f-waves}. For balance laws with source term

\[ h_t + f(h, x)_x = \psi(h, x) \]  (2.45)

(the Shallow Water Equations (2.15) are of this form) (2.43) and (2.44) take the form

\[ f_i (H_i) - f_{i-1} (H_{i-1}) - \Delta x \Psi_{i-1/2} = \sum_{p=1}^{m} \beta_{i-1/2}^p r_{i-1/2}^p \equiv \sum_{p=1}^{m} \mathcal{Z}_{i-1/2}^p \]  (2.46)

and

\[ \beta_{i-1/2} = R_{i-1/2}^{-1} \left( f_i (H_i) - f_{i-1} (H_{i-1}) - \Delta x \Psi_{i-1/2} \right) \]  (2.47)

respectively.

This approach tackles the flux problem in a different way and is applicable for the shallow water equations with or without source term. Overall, there exist many solvers. Besides the F-Wave solver only the augmented Riemann solver plays a role in this work, which is introduced in [5].

### 2.4.3 Grids

We discussed cells \( C_i \) and looked at computation of fluxes and solving Riemann problems between them. Now we shall look at how to arrange our cells. We will do this for two dimensions only.

We are given a rectangular or quadratic domain and need to discretise it by splitting it into cells. One idea jumps right to mind: to arrange the cells in the fashion of a chess board. A 2D \textit{Cartesian grid} consists of rectangles or squares and is defined by the number of cells in each dimension \( n_x, n_y \) and the cell length in each dimension \( d_x, d_y \). An example is given in figure 3.

![Figure 3](image_url)

Figure 3: A Cartesian grid with \( n_x \cdot n_y = 4 \cdot 4 = 16 \) cells with equal width and length \( (d_x = d_y) \).

While this grid is something everybody will have seen in one way or another, there are countless ways to design a grid, especially when the discretised domain has an odd shape.
that needs to be nicely reproduced by the discretisation. For our purposes, we will limit this section to what we actually need and therefore introduce just one other form of grid, a triangular grid. Such a grid might consist of triangles of any shape; we restrict ourselves to right-angled triangles with legs of equal length, one possibility is given in figure 4a.

Regardless of the cell shape, it might be useful to represent the given domain more accurately in a certain area by using smaller cells there. A grid using cells of different sizes is called adaptive; in case it changes its form in between time steps and uses the smaller cells in another area, it is called dynamically adaptive (an illustrating application for dynamic adaptation is given in section 2.5). Whenever we refer to the “triangular grid” in this text, we actually mean a grid consisting of the above defined triangles, which is adaptive and therefore has different cell sizes.

Furthermore, we decide how to construct such a grid: Start with a square domain and connect the opposing corners with edges. To receive smaller cells, refine a cell by splitting it in two via adding an edge connecting the hypotenuse midpoint and the opposing triangle corner. We do not allow hanging nodes, i.e. when two neighbouring cells share their hypotenuses and one gets refined, this would introduce an additional node to the non-refined cell at its hypotenuse midpoint - we do not allow this. An example for a hanging node is given in figure 4c.

One more definition is needed: the cell depth. On our initial grid consisting of four triangles, the cell depth is set to 1 for every cell/triangle. If a cell $C_i$ with depth $d_i$ is refined, the resulting two cells $C_{i1}$ and $C_{i2}$ have depth $d_i + 1$, so $d_{i1} = d_{i2} = d_i + 1$. An example for a valid partially refined grid which also shows the cell depths can be found in figure 4b.

### 2.5 Tsunami Simulation

As promised, we want to give an application for the previously introduced theory: tsunami simulation. Disastrous events in the recent past, as the 2004 tsunami in the Indian Ocean or the 2011 Tohoku tsunami, demonstrate its relevance while possibilities for scientific study of this phenomenon are rare.
To capture a large-scale object like a propagating tsunami sufficiently, a huge domain has to be used which stretches over thousands of kilometres. Such a domain may be discretised using both possibilities mentioned in the preceding section; the advantages of an adaptive grid are easy to see here. While areas of ocean which are not yet affected may be represented by large cells, interesting features like coastlines or the actual tsunami wave front can be represented at high resolution. The over-time adaptivity allows to keep the high resolution at the moving tsunami. The essential physics modelling part is covered by the Shallow Water Equations (2.15). One might wonder if shallow applies to the Pacific Ocean - in this context yes, since the ocean depth is small compared to domain size in both x- and y-direction. [5]

2.6 Error Measurement

For some problems - such as the Riemann problem - one might be interested in how far off an approximated solution provided by above methods is from the actual analytic solution. Therefore consider the $L^1$-norm $|\cdot|_1$ for a vector $\vec{x} = (x_1, \ldots, x_n)^T$:

$$||\vec{x}||_1 = \sum_{i=1}^{n} |x_i|. \quad (2.48)$$

In the context of the one dimensional Riemann problem and a two dimensional grid consisting of any number of cells, this definition needs some modification. Let the Riemann problem evolve in $x$-direction, while the grid expands in $x$- and $y$-direction, then $h_R(x,t)$ is the water height at $x$ at time $t$ given by the Riemann solution. The initial setup then shall be

$$h_R(x,0) = \begin{cases} h_l & \text{if } x < 0.5, \\ h_r & \text{if } x > 0.5, \end{cases} \quad u(x,0) = 0. \quad (2.49)$$

$C_{x,y}$ shall denote any cell from the grid, $x$ and $y$ the coordinates of the cell centre; the coordinates shall be scaled to the unit interval: $x, y \in (0, 1) \subset \mathbb{R}$. The grid is not supposed to consist of infinitely small grid cells, it has only a finite number of cells and therefore only a finite set $I$ of discrete coordinates: $(x, y) \in I \subset (0, 1)^2$. $h_S(x,y,t)$ the water height in cell $C_{x,y}$ at time $t$. Then the grid related to (2.49) will be initialised with the values

$$h_S(x,y,0) = \begin{cases} h_l & \text{if } 0 \leq x < 0.5, \\ h_r & \text{if } 0.5 \leq x \leq 1. \end{cases} \quad (2.50)$$

So at time $t$ at some point $x,y$ the analytic solution is given by $h_R(x,t)$, whereas the computed solution is given by $h_S(x,y,t)$. The error in any cell $C_{x,y}$ at time $t$ then is given by

$$|h_R(x,t) - h_S(x,y,t)|.$$

On an adaptive grid, the volume of any two cells (i.e., the areas they represent) might differ, therefore their errors may not be weighted in the same way; define $0 < \omega_{x,y} \leq 1$ the weight or cell volume of cell $C_{x,y}$. Then the $L^1$ error $e_{L^1}$ of the computed solution on the whole grid is
2.6 Error Measurement

\[ e_{L^1} = \sum_{(x,y) \in I} |h_R(x,t) - h_S(x,y,t)| \cdot \omega_{x,y}. \]  

(2.51)

Other norms are the \( L^2 \)-norm

\[ \| \vec{x} \|_2 = \left( \sum_{i=1}^{n} |x_i|^2 \right)^{\frac{1}{2}} \]  

(2.52)

and the \( L^\infty \)-norm

\[ \| \vec{x} \|_\infty = \max (|x_1|, \ldots, |x_n|) \]  

(2.53)

giving us \( e_{L^2} \) and \( e_{L^{\infty}} \):

\[ e_{L^2} = \sqrt{\sum_{(x,y) \in I} \left| h_R(x,t) - h_S(x,y,t) \right|^2 \cdot \omega_{x,y}} \]  

(2.54)

\[ e_{L^{\infty}} = \max_{(x,y) \in I} \left( |h_R(x,t) - h_S(x,y,t)| \right). \]  

(2.55)
3 Setup and Parameter Optimisation

3.1 A Riemann Problem Benchmark

For the key part of this work, a simulation scenario was needed to compare the two types of grids previously introduced. In section 2.3 the Riemann problem was discussed and an analytic solution was given. This is not only relevant for solving the local Riemann problem between two neighbouring grid cells; we can simulate the problem itself using an entire grid and then investigate how well the true solution is reproduced.

The two types of grids are represented by two different programs, which will be introduced more thoroughly in upcoming sections. The Cartesian grid is used in SWE, while sam(oa)$^2$ (see sections 3.2 and 3.3 resp.) works with the dynamically adaptive grid discussed further above.

Since both of these programs are not designed for 1D purposes, the Riemann problem needed to be mirrored on their 2D domains. It was set-up as defined by (2.50), with initial water height $h_l = 3$ on the left half of the domain and $h_r = 1$ on the right, while using the simple even bathymetry $b = 0$ in the entire area. The domain has size $100 \text{m} \times 100 \text{m}$, and the comparison of simulated solution and true solution was made at time $t = 4s$, when shock and rarefaction wave have formed and are separated from an intermediate state area; see figure 5 for this scenario as computed by sam(oa)$^2$.

![Figure 5: The Riemann problem in x-direction, at $t = 4s$, as computed by sam(oa)$^2$. The water height is plotted in z-direction.](image-url)
The actual solution for the water height \( h \) for \( h_l = 3, h_r = 1 \) was found to be

\[
h(x, y, t) = h(x, t) = \begin{cases} 
3 & \text{if } x < s_l \cdot t \\
\alpha \cdot (\beta - \frac{z}{t})^2 & \text{if } s_l \cdot t < x < s_r \cdot t \\
h_m & \text{if } s_r \cdot t < x < s_m \cdot t \\
1 & \text{otherwise}
\end{cases}
\]  

(3.1)

with \( h_m = 1.84857660309676, \alpha = 0.0113301801441992, \beta = 10.8480320795986, s_l = -5.4201603979929, s_m = 5.08133721790905 \) and \( s_r = -1.92518574923489 \). [5]

To measure the difference between analytic solution and the solutions produced by both sam(oa)\(^2\) and SWE, the error definitions from section 2.6 were used. Both programs had to be slightly modified to enable the error computation.

### 3.2 SWE

SWE [1] is a teaching code developed at TU München. Written in C++, it is intended to allow learning of parallel concepts and programming languages by e.g. implementation of a CUDA kernel for memory-efficient computation of cell updates (or optimising such) into a given framework. It uses a 2D Cartesian grid, which allows easy parallelisation of grid patches; this can be performed by using MPI, OpenMP, CUDA and hybrid approaches (e.g. MPI + OpenMP). The name actually refers to the Shallow Water Equations. To avoid confusion of both, the term “SWE” will correspond to the program, while when the shallow water equations are meant, their name will be spelt out in full.

In contrast to sam(oa)\(^2\), SWE has only a limited choice of parameters. While the domain size is fixed in the scenario implementation, one can determine the number of cells in both dimensions and therefore create grids with rectangular or square grid cells in different resolutions (which stays fixed during the whole run, so SWE is non-adaptive). Other available parameters available are not relevant to the simulation result. While there are a number of grid solvers available in theory, only the F-Wave solver could actually be used during the composition of this work.

Two major extensions of SWE were performed for this work. The grid is organised in a data structure called \texttt{SWE\_Block}. This class covers a patch of a grid, holds the unknowns for every cell and offers functions necessary for parallelised computation. This class received a new function \texttt{SWE\_Block::computeError} which computes for the cells of this block the analytic solution and the difference to the simulated solution. Those results are stored as member variables of the block. Furthermore, the gravitational constant had to be set to a more accurate value from \( g = 9.81 \frac{m}{s^2} \) to \( g = 9.80665 \frac{m}{s^2} \).

As it seemed best fit for SWE, the error is simply computed after all time steps are done. Since the blocks are handled in parallel by different MPI ranks, their respective error results have to be collected, summed up and wrapped in an output file globally.

Furthermore, a scenario needed to be implemented to define the domain and its initial values.

Code snippets which cover these two parts are given in appendix A.1.
3.3 \textit{sam(oa)}$^2$

\textit{sam(oa)}$^2$ [5], Space-Filling Curves and Adaptive Meshes for Oceanic And Other Applications, was also developed at TU München. It is designed for memory-efficient solving of two dimensional partial differential equations using triangular grids and hybrid parallelisation. It is not limited to simulation of the Shallow Water Equations but is able to compute other scenarios, e.g. porous media flow. It uses Sierpinski traversals, which provide a sequential order to iterate over all cells of the triangular grid; this ordering also gives partitions of the grid for load balancing and parallelisation. The latter is realised with MPI and, if desired, OpenMP in addition.

\textit{sam(oa)}$^2$ is written in Fortran and has a wide range of parameters to be set for every execution. Minimal and maximal refinement depths are just two of them, though those are the most relevant for this work. Further parameters allow to trigger different kinds of output or specify parallelisation aspects. Since this grid parallelisation is not performed deterministically, the resulting grid partitions might vary in different runs. This means that the grid does not always look the same for runs with same parameters, which also might have minimal effects on the result, though these are negligibly small. This circumstance and the depth parameters result in the fact that we can aim only very roughly at a certain number of cells - which also varies over time due to the performed adaptions. Default solver is an augmented Riemann (“AugRie”) solver, and luckily the F-Wave solver is available amongst the others.

Like SWE, \textit{sam(oa)}$^2$ did not initially provide the necessary functions for the Dam-Break Benchmark. It uses a kernel approach to perform the grid traversal, with the kernels operating on the whole grid, grid sections or element-wise. Therefore a module was created whose kernels are executed after every simulation phase, i.e. in our case at the end of the simulation. It is realised in mostly the same way as for SWE (initialisation, parallel local error computation for every grid section, summation of all partial errors, output in .txt file), with respect to the kernel concept. The corresponding code snippets are given in appendix A.1.

The set-up of the scenario could be realised easily, as \textit{sam(oa)}$^2$ works with a domain with coordinates in $[0,1]^2$, where simply an offset and a scaling factor had to be set to receive the 100m $\times$ 100m domain. Setting the initial water height and bathymetry was also reduced to a few lines of code; besides that, we restricted \textit{sam(oa)}$^2$ to refine only at the shock initially to avoid unnecessary initial refinement which had been observed before. The relevant parts of code for the latter two are also given in A.1.

3.3.1 Parameter Optimisation for \textit{sam(oa)}$^2$

An initial run of \textit{sam(oa)}$^2$ resulted in L1 errors visualised in figure 6. The maximal and minimal refinement depths $d_{\text{max}}$ and $d_{\text{min}}$ were chosen to vary between 10 and 28 and 6 and 28 respectively. Note that a continuous plane was created from the discrete points via a Delaunay triangulation and that some combinations of high refinement depths are missing (computation too time-intensive).

We note that for small $d_{\text{min}}$ the error seems to decrease when $d_{\text{max}}$ increases, though this seems to be limited. To receive an even smaller error, we apparently have to increase $d_{\text{min}}$ as well. Having a look at the number of used cells gives further insight into this matter, see figure 7. Here we can see that the number of cells barely increases when increasing
3.3 $\text{sam(oa)}^2$

Figure 6: Error for a selection of different refinement depths, marked by the black dots. To enhance interpretability, the discrete domain was interpolated to a continuous one.

Figure 7: The number of cells on the respective domain of figure 6, using a logarithmic scale.
$d_{\text{max}}$ for small $d_{\text{min}}$. The plot even suggests that for $d_{\text{min}} \gtrsim 14$ the number of cells is even no longer related to $d_{\text{max}}$ at all. Reminding ourselves that higher $d_{\text{max}}$ should lead to creation of smaller and therefore more cells, we are led to the observation that this does not seem to happen. Instead, we have to “enforce” higher resolutions by prohibiting larger cells via increasing $d_{\text{min}}$.

Obviously, this is not the desired behaviour, which is that regardless of the chosen $d_{\text{min}}$, when increasing $d_{\text{max}}$ the number of cells should increase and the result should be more accurate i.e. the error should decrease. A closer look at a scenario plot for $d_{\text{min}} = 6$ and $d_{\text{max}} = 28$ yielded that the desired $d_{\text{max}}$ was not present, but the highest refinement depth was 23. sam(oa)$^2$ did not refine sufficiently, which leads us to a closer look at how refinement is actually performed.

Currently, refinement of a cell is issued if

$$\langle \vec{p}_t, \vec{p}_t \rangle > (c \cdot r)^2,$$  \hspace{1cm} (3.2)

with $c$ being a factor that scales $[0, 1]^2$ to domain space, $p$ the momentum and $r = 0.002$. If this condition is met and the current grid depth is smaller than the maximal depth, the grid will be refined in the next time step. The respective coarsening condition is

$$\langle \vec{p}_t, \vec{p}_t \rangle < \left( \frac{c \cdot r}{2} \right)^2$$  \hspace{1cm} (3.3)

with taking the permitted $d_{\text{min}}$ into account. For the benchmark scenario, $r = 0.002$ seems to be too big to allow refinement above 23.
In a second approach, to get better behaviour for a maximum refinement depth larger than 18, the factor $r$ was chosen to depend on $d_{\text{max}}$ as follows:

$$
 r (d_{\text{max}}) = \begin{cases} 
0.002 \cdot \frac{1}{d_{\text{max}}-18} & \text{if } d_{\text{max}} > 18 \\
0.002 & \text{otherwise}
\end{cases}
$$

The idea behind this was to aim for better behaviour while keeping computation times at a bearable level. At the same time, a valid value for all $d_{\text{max}}$ is given. A comparison of both refinement criteria is given in figure 8 (since the new criterion does not apply for $d_{\text{max}} \leq 18$, we compare the results on a smaller domain).

By just looking at the plots, we can see that an improvement is made, though further optimisation is still necessary. For $d_{\text{min}} \lesssim 12$ the number of cells keeps growing with $d_{\text{max}}$, while the error is decreased. For $d_{\text{min}} = 16$, $d_{\text{max}} = 26$ the error is worse than for $d_{\text{min}} = 14$, $d_{\text{max}} = 26$, but it decreases again afterwards. This shows that the refinement rule has to be further improved to allow the behaviour to become more predictive. The smallest errors are still observed for the highest $d_{\text{min}}$, though in this case we are dealing with an rather “unadaptive” grid. The cell depths range from 26 to 28, which is also very expensive in both computational resources and computation time due to the high number of cells. So if we do not go for the minimal error, but a small one at reasonable cost, we find that e.g. for $d_{\text{max}} = 26$ we receive approximately the same error for both $d_{\text{min}} = 6$ and $d_{\text{min}} = 20$, while the latter comes at a larger number of cells. A first indicator that adaptivity pays out.
4 Comparison of SWE and sam(oa)$^2$

It was originally planned to compare sam(oa)$^2$ and SWE for the dam-break problem and to cross-check their results of simulating the 2011 Tohoku tsunami against real measured data. Unfortunately, there were ongoing problems getting SWE to run the tsunami simulation, from giving interpretable output data to producing valid data at all, an alternative approach to get the relevant data extracted came back with invalid water heights. Therefore the latter comparison could not be made and only sam(oa)$^2$’s tsunami results are compared to the real measurements.

4.1 Dam-Break Benchmark

The first results of sam(oa)$^2$ for the dam-break problem benchmark has already been presented. To compare these results with SWE, we decided to investigate the error received for a certain number of cells. This quantity is closely related to memory usage - the more cells there are, the more values have to be stored. While we can aim directly for a certain number of cells at SWE, we cannot do so for sam(oa)$^2$. There the number of cells results from the set minimal and maximal refinement depths, the performed adaptions and the grid partitioning, in addition it varies over time. We therefore simply ran sam(oa)$^2$ first, noted amounts of cells for the various input parameters and then used the corresponding parameters on SWE.

So far, we did not consider that sam(oa)$^2$ can be seen as “more adaptive” or “less adaptive” depending on minimal and maximum refinement depths. The closer those two are, the smaller is the range of cell sizes. Is $d_{\min} = d_{\max}$, there is no adaptivity at all and we receive a nonadaptive grid like shown in figure 4a. This poses another possible comparison: triangular and Cartesian grid, both non-adaptive.

Usage of the new refinement criterion gave us its best behaviour for small $d_{\min}$, but since we aim for “high adaptivity” anyway, this is fine. We therefore fix $d_{\min}$ to 6 and vary $d_{\max}$. Furthermore are for sam(oa)$^2$ various solvers available. The AugRie solver is default, the F-Wave solver is also available; we will try both. Unlike sam(oa)$^2$, SWE is not able to do its computations in double precision; however, no result difference could be found, so single precision seems precise enough for this problem.

The actual L1 errors for SWE and sam(oa)$^2$ both using the F-Wave solver are plotted in figure 9. Both non-adaptive grids scale logarithmic for the given amounts of up to ten to twenty million cells, though sam(oa)$^2$ gives a smaller error. Therefore, we could assume that further increasing of the cells would continue to give better errors. It also suggests that sam(oa)$^2$ keeps its “advantage” and will continue to provide a better result than SWE.

The adaptive case however behaves quite differently. The jump at 20k cells results from the refinement criterion which only takes effect for $d_{\max} > 18$; the result is given for $d_{\max} \leq 32$. In this area, the error keeps improving while the number of cells grows, though we can already guess that this is limited, since the differences between neighbouring sampling points grows smaller. This suggests that while the new criterion led to an improvement at first, it might be inconvenient to use it for $d_{\max} > 32$. As discussed before, there might still be room for optimisation concerning refinement.

However, the errors using an adaptive grid are significantly smaller than those for the non-
4.1 Dam-Break Benchmark

Figure 9: Result error for numbers of cells on logarithmic scale. Black: SWE. Blue: sam(oa)$^2$, non-adaptive. Red: sam(oa)$^2$, adaptive.

adaptive grids when using the approximate same number of cells. For a efficient simulation though, memory usage is not the only relevant quantity. And while sam(oa)$^2$ might give the better error for less memory, the time intensity might be quite different: while adaptive sam(oa)$^2$ gives a much smaller error than SWE for approximately 470k cells, the computations lasted 187.2 seconds on 256 cores, while SWE finishes computations on a grid of 536k cells in 6 seconds using only 32 cores of the same kind. We will not take a more detailed look into these performance matters, but want to point out that the comparison based on amount of cells does not give the full picture.

Also interesting is the comparison of sam(oa)$^2$ using the two solvers AugRie and F-Wave. We discover that the default solver of sam(oa)$^2$ proves to actually be slightly worse than the F-Wave. The difference is small though, considering the gap between adaptive and non-adaptive results, and also decreases when increasing the number of cells. There is more to consider, like which solver might work faster; we want to emphasise that the result cannot only be improved by increasing $d_{\text{max}}$ but also by usage of another solver or some other optimisation. On the other hand, while another solver might give a slightly worse result, it might work faster; we have already decided to not look at performance results.

In figure 11 we visualise how SWE and sam(oa)$^2$ reproduce the analytic solution for approximately 44k cells - here the errors are already quite apart. We can see how the graphs of those two are step functions where the water height varies from cell to cell, and
4.2 Tsunamis

As section 2.5 already addressed, tsunami simulation benefits from usage of an adaptive grid. Large amounts of small cells allow to invest high resolution at interesting areas without spending too much resources in uneventful regions. This means that sam(oa)$^2$ should give us more accurate results for the same number of cells compared to SWE. We could also formulate this the other way round, that a run of SWE which aims at the same resolution as sam(oa)$^2$’s maximal resolution should be way more expensive. Unfortunately, without results from SWE, we cannot investigate this and have to confine ourselves to analysis of sam(oa)$^2$ results.

In this work we already focused on how well sam(oa)$^2$ and SWE reproduce a certain scenario. For the 2011 Tohoku tsunami there is some data available we can compare with; in the Pacific Ocean there are a number of buoys which measure the ocean’s water height. Figure 12 shows the positions of buoys in the north-eastern Pacific as well as the tsunami’s point of origin and its over-time propagation. The three nearest buoys

while the step length (= cell length) is constant for SWE, it varies for sam(oa)$^2$. It is also obvious that sam(oa)$^2$ is much closer to the analytic solution than SWE, especially at the shock, where SWE most likely produces its largest errors. The shock keeps captured by small cells in sam(oa)$^2$, in SWE the coarse resolution there skews it. This confirms the theory that sam(oa)$^2$ captures the relevant areas better with its adaptive grid.

Figure 12: Map of the north-eastern Pacific, showing the positions of NOAA buoys. The red star marks the epicentre of the undersea earthquake that caused the 2011 Tohoku tsunami; yellow and red squares correspond to buoys. The coloured ocean refers to the propagation of the tsunami.[6]
Figure 13: The measured water height over time at the coordinates of buoy 21418. The time is given in days since year’s beginning. The actual buoy data is given in red, the other three lines correspond to results of \text{sam(oa)}^2 for different $d_{\text{max}}$. 

Figure 14: Comparison of \text{sam(oa)}^2 and buoy 21419. For further explanation of line colours and axis arrangement see figure 13.
with the numbers 21418, 21413 and 21419 were chosen to compare their measured data with sam(oa)$^2$; as every simulation run starts with the undersea earthquake, buoys with less proximity would need longer simulation runs for the tsunami to reach their position. sam(oa)$^2$ is able to track the behaviour of unknowns at any pair of coordinates.\footnote{Implementation of this feature and past analysis of its results were performed when the author was employed as student assistant, not during the composition of this thesis. However, the results given here were newly computed to represent the current version of sam(oa)$^2$.}

We investigate how the result of sam(oa)$^2$ varies for different maximal refinement depths $d_{\text{max}}$; therefore we compare the data given by the three buoys with simulations where $d_{\text{max}} = 20, 24$ or $26$ and $d_{\text{min}} = 6$. Plots of these four for every buoy are given in figures 13, 14 and 15.

Before we start, we should remind ourselves that a simulation like sam(oa)$^2$ has limits. The discretised Pacific gives only an approximation of the actual one, and the physics are not modelled completely (e.g., sam(oa)$^2$ does not yet consider hydrostatic pressure). Furthermore, the water heights measured by the buoys do not only refer to the tsunami, but also to surface waves caused by wind and weather or passing by ships. The buoys measure the water height not constantly, and the time between two measurement points might be 15 seconds, 60 seconds or even 15 minutes as can be seen from their raw data. There might be measurement errors as well. Overall, this comparison will give just an idea if the computation results are somewhere close to reality.

From all three figs. 13 to 15 we can see how the increasing maximal depth affects the simulation result. For $d_{\text{max}} = 20$ the resulting graphs are rather smooth, comparing this

Figure 15: Comparison of sam(oa)$^2$ and buoy 21413. For further explanation of line colours and axis arrangement see figure 13.
to $d_{\text{max}} = 24$ and comparing that again with the graph of $d_{\text{max}} = 28$, we can assume that increasing the refinement depths will give higher amplitudes of wave peaks and troughs. It also looks like for a higher $d_{\text{max}}$ a single wave peak or trough might become two compared to the smaller $d_{\text{max}}$, compare the graphs of $d_{\text{max}} = 24$ and $d_{\text{max}} = 20$ in the area from 70.35 to 70.38 in figure 15 or $d_{\text{max}} = 28$ vs. the other two around 70.35 in figure 14. Therefore sam(oa)$^2$ seems to deliver a more smeared result for a lower $d_{\text{max}}$. The growth in the amplitudes when increasing $d_{\text{max}}$ might be limited though, for all three buoys it was less when going from $d_{\text{max}} = 24$ to 28 than when increasing 20 to 24. However, as the runs with $d_{\text{max}} = 28$ were already time-intensive (which is why the corresponding graphs end before the other - the time limit was exceeded), we did not go for $d_{\text{max}} = 32$. 

Now for the buoy data: looking at figures figs. 13 to 15, we can see the previously mentioned not-modelled aspects of sam(oa)$^2$. Before anything happening is detected at all, sam(oa)$^2$ gives a straight water height of zero. The buoys first measure a short shock of high amplitude, which might correspond to an initial shock wave which is sent out by any earthquake and travels faster than actual ruptures, along with some noise which is probably surface waves. All three buoys detect the first wave peak of the tsunami some time later than sam(oa)$^2$ computes them, this difference was found to be about three minutes. This peak, whose height is met differently well for the buoys and depths, is followed by a larger trough. In between those buoy 21418 measures another trough and peak; this is also reported by buoy 21419 though far less strong and buoy 21413 does not show it at all. The larger trough is again time-displaced, though sam(oa)$^2$ seems to give too high amplitude here. The further behaviour in all three cases seems to fit more or less, as when the major wave has passed we cannot tell any more if the buoys are measuring actual tsunami effects or just noise; the sam(oa)$^2$ graphs do not differ too much from those values as well.

The final evaluation whether this is a good or bad reproduction of the real proceedings remains open, as both the buoy data and the simulation are not giving the full picture. However, we might decide that sam(oa)$^2$ captures the time and form of the tsunami well enough.

We also examined, whether usage of the new refinement criterion had any effect on the results for the tsunami scenario. We could not confirm that; apparently the old criterion is sufficient in this case.
5 Conclusion

We have introduced two simulation programs sam(oa)$^2$ and SWE and the theory they are based on. To investigate how well they reproduce a scenario whose exact behaviour is known, we extended the two programs with the initial setup of the dam-break scenario and the required functionalities to measure the difference between simulation solution and exact solution. The theory of the exact solution and how to perform the measurement was also given.

In the progress of error analysis we discovered that sam(oa)$^2$ does not refine its grid optimally for the dam-break scenario. We introduced and implemented an alternative refinement criterion and found it to deliver better, yet not optimal results. Nevertheless, the improvement was sufficient enough to continue the analysis.

We found that even when sam(oa)$^2$ is non-adaptive, it gives better results than SWE for the approximate same amount of cells. Comparison of SWE with adaptive sam(oa)$^2$ yielded that adaptivity provides significantly smaller errors for the same number of cells as when using a non-adaptive grid. The reason for this was visualised: the adaptive grid used in sam(oa)$^2$ invests the resources in the important regions and therefore delivers a more accurate approximation of the solution. SWE captures the distinctive shock wave rather badly and illustrates the disadvantage of a Cartesian grid compared to dynamically adaptive one: small errors are achieved only for high resolution, which comes at higher cost.

We also mentioned that comparison based on the used amount of cells does not give a complete picture. While the number of cells is related to the memory usage, other important magnitudes of performant simulation were not reviewed, like parallel efficiency or computation time.

Although the new refinement criterion was an improvement, its limits also became visible already. This leaves the matter open; while it might not be relevant for the tsunami scenario, other scenarios could benefit from a proper refinement criterion.

Comparison of the 2011 Tohoku tsunami as computed by sam(oa)$^2$ with actual data from measuring buoys showed that using more cells and higher resolution changes the details of the result, while its basic structure of where and when remains the same. The validity of this comparison though may be challenged as both simulation and measured data are limited.

Overall, sam(oa)$^2$ has proven the advantages of dynamically adaptive grids.
A Appendix

A.1 SWE Code Snippets

The class \texttt{SWE\_Block} was provided with a function \texttt{computeError} which measures the local difference between analytic and simulation solution. The local error variables are initialised with zero, the cell volume is set and then the true solution is computed. The differences from the local solution is then added to the block’s local total error.

```cpp
void SWE\_Block::computeError(float sim\_time) {

    float state\_left, state\_rarefaction, state\_intermediate;
    float domain\_first, domain\_length, shock\_position;
    float h\_sol, u\_sol, h\_diff, h\_rarefaction, h\_intermediate;

    state\_left = -5.42401603979929;
    state\_rarefaction = -1.92518574923489;
    state\_intermediate = 5.08133721790905;

    domain\_first = 0.0;
    domain\_length = 100.0;
    shock\_position = 50.0;

    comp\_error[0][0] = 0.0;
    comp\_error[0][1] = 0.0;
    comp\_error[0][2] = 0.0;
    comp\_error[1][0] = 0.0;
    comp\_error[1][1] = 0.0;
    comp\_error[1][2] = 0.0;

    cell\_volume = float(dx * dy) / (100.0 * 100.0);

    for(int j=1; j<=ny; j++) {
        for(int i=1; i<=nx; i++) {

            float x = offsetX + (i-0.5f)*dx;
            float y = offsetY + (j-0.5f)*dy;
            float switched\_x = x - shock\_position;

            if (switched\_x < state\_left \* sim\_time) {
                h\_sol = 3.0;
                u\_sol = 0.0;
            } else if (switched\_x < state\_rarefaction \* sim\_time) {
                h\_sol = 0.0113301801441992 * pow(10.8480320795986 - switched\_x / sim\_time, 2);
                u\_sol = -(2.0/3.0) * (10.8480320795986 - switched\_x / sim\_time) + 10.8480320795986;
            } else if (switched\_x < state\_intermediate \* sim\_time) {
                h\_sol = 1.8457660309676;
            }
        }
    }
}
```
The computations above take place in parallel. So afterwards all the blocks’ errors have to be gathered to compute the overall errors. This is performed using functions provided by MPI. Afterwards, all data is wrapped up in a simple .txt file. Except from swe mpi.cpp:

```cpp
1_1waveBlock.computeError(l_t);

float whole_errors[2][3];

MPI_Barrier(MPI_COMM_WORLD);

for (int i=0; i<2; i++) {
    for (int j=0; j<2; j++) {
        MPI_Reduce(&l_1waveBlock.comp_error[i][j], &whole_errors[i][j], 1,
                    MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
    }
    whole_errors[i][1] = sqrt(whole_errors[i][1]);
    MPI_Reduce(&l_1waveBlock.comp_error[i][2], &whole_errors[i][2], 1, MPI_FLOAT,
                MPI_MAX, 0, MPI_COMM_WORLD);
}

float computation_time=0; //cpu & communication time for all processors in total
float my_time = tools::Logger::logger.getTime("CpuCommunication");

MPI_Reduce(&my_time, &computation_time, 1, MPI_FLOAT, MPI_MAX, 0, MPI_COMM_WORLD);

if (l_mpiRank == 0) {
    std::stringstream filename2;
```
In addition, SWE\_simple\_scenarios.hh was extended to hold a further scenario, which creates the domain needed:

```cpp
class SWE\_DamBreakScenario : public SWE\_Scenario {

public:

  float getBathymetry(float x, float y) {
    // initialise with even bathymetry
    return 0.f;
  };

  float getWaterHeight(float x, float y) {
    // h\_l = 3, h\_r=1
    return (x < 50.f) ? 3.f : 1.f;
  };

  virtual float endSimulation() { return (float) 4; }; // determines the time
  // covered by the simulation

  virtual BoundaryType getBoundaryType(BoundaryEdge edge) { return OUTFLOW;
  };

  float getBoundaryPos(BoundaryEdge i\_edge) { // 100m x 100m domain
    if (i\_edge == BND\_LEFT)
      return (float)0;
    else if (i\_edge == BND\_RIGHT)
      return (float)100;
    else if (i\_edge == BND\_BOTTOM)
      return (float)0;
    else
      return (float)100;
  };
```

A.2 \textit{sam(oa)}$^2$ Code Snippets

To realise the error computation in \textit{sam(oa)}$^2$, a module \texttt{SWE\_compute\_error.f90} was written; the relevant parts are given here.

Before the actual grid traversal starts, all values are initialised to zero.

```fortran
subroutine pre_traversal_op(traversal, section)
  type(t_swe_compute_error_traversal), intent(inout) :: traversal
  type(t_grid_section), intent(inout) :: section

  traversal%norm_data_h%error_l1 = 0.0_GRID_SR
  traversal%norm_data_h%error_l2 = 0.0_GRID_SR
  traversal%norm_data_h%error_max = 0.0_GRID_SR
  traversal%norm_data_u%error_l1 = 0.0_GRID_SR
  traversal%norm_data_u%error_l2 = 0.0_GRID_SR
  traversal%norm_data_u%error_max = 0.0_GRID_SR

end subroutine
```

Then for every element the analytic solution at the cell coordinates is computed and the difference to the simulated values is saved.

```fortran
subroutine element_op(traversal, section, element)
  type(t_swe_compute_error_traversal), intent(inout) :: traversal
  type(t_grid_section), intent(inout) :: section
  type(t_element_base), intent(inout) :: element

  !local variables
  type(t_state), dimension(_SWE_CELL_SIZE) :: Q
  type(t_state), dimension(6) :: Q_test
  real (kind = GRID_SR) :: g, hl, hr, h, u, h_diff, u_diff, time, cell_volume
  real (kind = GRID_SR) :: state_left, state_rarefaction, state_intermediate
  real (kind = GRID_SR) :: domain_length !length of domain
  real (kind = GRID_SR) :: domain_first !first coordinate of domain
  real (kind = GRID_SR) :: shock_position !shock_position in domain coordinates
  real (kind = GRID_SR) :: coords(2) !cell center coordinates
  call gv_Q%read(element, Q)

  time = section%r_time

  !compute analytic solution
  lg = 9.80665_GRID_SR
  hl = 3.0_GRID_SR
```
hr = 1.0_GRID_SR
state_left = -5.42401603979929_GRID_SR
state_rarefaction = -1.92518574923489_GRID_SR
state_intermediate = 5.0813721790905_GRID_SR

domain_length = cfg%scaling
domain_first = cfg%offset(1)
shock_position = 0.0_GRID_SR

coords = samoa_barycentric_to_world_point(element%transform_data, [1.0_GRID_SR/3.0_GRID_SR, 1.0_GRID_SR/3.0_GRID_SR]) * cfg%scaling + cfg%offset

!compute h and u:coords(1)
if (coords(1) < state_left * time) then
    h = 3.0_GRID_SR
    u = 0.0_GRID_SR
else if (coords(1) < state_rarefaction * time) then
    h = 0.0113301801441992_GRID_SR * (10.8480320795986_GRID_SR - coords(1) / time)**2
    u = -(2.0_GRID_SR/3.0_GRID_SR) * sqrt((10.8480320795986_GRID_SR - coords(1) / time)**2) + 10.8480320795986_GRID_SR
else if (coords(1) < state_intermediate * time) then
    h = 1.84857660309676_GRID_SR
    u = 2.33255352704294_GRID_SR
else
    h = 1.0_GRID_SR
    u = 0.0_GRID_SR
end if

!compute l1 and l2 local l1 and l2 norm (adding everything up is done by mpi, sqrt is done afterwards)
h_diff = abs(h - t_basis_Q_eval([1.0_GRID_SR/3.0_GRID_SR, 1.0_GRID_SR/3.0_GRID_SR], Q%h))
u_diff = abs(u - t_basis_Q_eval([1.0_GRID_SR/3.0_GRID_SR, 1.0_GRID_SR/3.0_GRID_SR], Q%p(1)) / t_basis_Q_eval([1.0_GRID_SR/3.0_GRID_SR, 1.0_GRID_SR/3.0_GRID_SR], Q%h))
cell_volume = element%cell%geometry%get_volume()

traversal%norm_data_h%error_l1 = traversal%norm_data_h%error_l1 + h_diff * cell_volume
traversal%norm_data_h%error_l2 = traversal%norm_data_h%error_l2 + h_diff * h_diff * cell_volume
traversal%norm_data_u%error_l1 = traversal%norm_data_u%error_l1 + u_diff * cell_volume
traversal%norm_data_u%error_l2 = traversal%norm_data_u%error_l2 + u_diff * u_diff * cell_volume
traversal%norm_data_h%error_max = max(h_diff,
    traversal%norm_data_h%error_max)
traversal%norm_data_u%error_max = max(u_diff,
    traversal%norm_data_u%error_max)
end subroutine

Afterwards summation of the values is performed, again using MPI functionalities, fol-
lowed by wrap-up to A .txt file.

subroutine post_traversal_grid_op(traversal, grid)
  type(t_swe_compute_error_traversal), intent(inout) :: traversal
  type(t_grid), intent(inout) :: grid

!local variables
  type(t_grid_info) :: grid_info
  integer :: i_error, i, j, k, alloc_err, nr_of_edges,
             nr_of_cells
  integer(4) :: i_rank, i_section, e_io
  logical :: l_exists
  integer :: counter = 1

# if defined(_MPI)
  call mpi_barrier(MPI_COMM_WORLD, i_error); assert_eq(i_error, 0)
  call reduce(traversal%norm_data_h%error_l1,
      traversal%children%norm_data_h%error_l1, MPI_SUM, .true.)
  call reduce(traversal%norm_data_h%error_l2,
      traversal%children%norm_data_h%error_l2, MPI_SUM, .true.)
  call reduce(traversal%norm_data_h%error_max,
      traversal%children%norm_data_h%error_max, MPI_MAX, .true.)
  call reduce(traversal%norm_data_u%error_l1,
      traversal%children%norm_data_u%error_l1, MPI_SUM, .true.)
  call reduce(traversal%norm_data_u%error_l2,
      traversal%children%norm_data_u%error_l2, MPI_SUM, .true.)
  call reduce(traversal%norm_data_u%error_max,
      traversal%children%norm_data_u%error_max, MPI_MAX, .true.)
  grid_info = grid%get_info(MPI_SUM, .true.)
  nr_of_edges = grid_info%i_crossed_edges + grid_info%i_color_edges +
                 grid_info%i_boundary_edges(RED)+ grid_info%i_boundary_edges(GREEN)
  nr_of_cells = grid%get_cells(MPI_SUM, .true.)
# else
  grid_info = grid%get_info(MPI_SUM, .false.)
  nr_of_edges = grid_info%i_crossed_edges + grid_info%i_color_edges +
                 grid_info%i_boundary_edges(RED)+ grid_info%i_boundary_edges(GREEN)
  nr_of_cells = grid%get_cells(MPI_SUM, .false.)
# endif
traversal\%norm_data_h\%error_l2 = sqrt(traversal\%norm_data_h\%error_l2)
traversal\%norm_data_u\%error_l2 = sqrt(traversal\%norm_data_u\%error_l2)
traversal\%i_output_iteration = traversal\%i_output_iteration + 1

if (rank_MPI == 0) then
  if (cfg\%i_phase_nr > 0) then
    ! write file
    # if defined(_SINGLE_PRECISION)
    write(pout_file_name, "(A, I0, A, I0, A, F6.5, A, F6.5, A, I0, A, I0, A)"
      "erroroutput_dmin", cfg\%i_min_depth, "_dmax",
      cfg\%i_max_depth, "_cou", cfg\%courant_number, "_dry",
      cfg\%dry_tolerance, "_p", size_MPI, "_phase", cfg\%i_phase_nr
      ,"_sp.txt"
    # elif defined(_DOUBLE_PRECISION)
    write(pout_file_name, "(A, I0, A, I0, A, F6.5, A, F6.5, A, I0, A, I0, A)"
      "erroroutput_dmin", cfg\%i_min_depth, "_dmax",
      cfg\%i_max_depth, "_cou", cfg\%courant_number, "_dry",
      cfg\%dry_tolerance, "_p", size_MPI, "_phase", cfg\%i_phase_nr
      ,"_dp.txt"
    # elif defined(_QUAD_PRECISION)
    write(pout_file_name, "(A, I0, A, I0, A, F6.5, A, F6.5, A, I0, A, I0, A)"
      "erroroutput_dmin", cfg\%i_min_depth, "_dmax",
      cfg\%i_max_depth, "_cou", cfg\%courant_number, "_dry",
      cfg\%dry_tolerance, "_p", size_MPI, "_phase", cfg\%i_phase_nr
      ,"_qp.txt"
    # endif
    open(unit=out_unit, file=pout_file_name, action="write",
      status="replace")
    write(out_unit, "(A)" "dmin, dmax, cells, edges, cou, dry_tol,
      processes, sim_time, h_error_l1, h_error_l2, h_error_max,
      u_error_l1, u_error_l2, u_error_max"
    write(out_unit, "(4(I0, A), 2(F6.5, A), I0, A, F12.4, 6(A, ES14.7))"
      cfg\%i_min_depth, ", " cfg\%i_max_depth, ", " nr_of_cells, ", ",
      nr_of_edges, ", ", cfg\%courant_number, ", ", cfg\%dry_tolerance,
      ", ", size_MPI, ", ", cfg\%t_phase, ", ",
      traversal\%norm_data_h\%error_l1, ", ",
      traversal\%norm_data_h\%error_l2, ", ",
      traversal\%norm_data_h\%error_max, ", ",
      traversal\%norm_data_u\%error_l1, ", ",
      traversal\%norm_data_u\%error_l2, ", ",
      traversal\%norm_data_u\%error_max
    close(out_unit)
  end if
end if

end subroutine
The following two excerpts from `SWE_initialize.f90` initialise the grid with the desired water heights and issue initial refinement at the initial shock only. Of the respective functions only the relevant lines are included.

```fortran
function get_initial_dof_state(section, x, lod) result(Q)
!
if (x(1) < 0.5 GRID_SR) then
  Q%h = 3.0 GRID_SR
else
  Q%h = 1.0 GRID_SR
endif
!
end function

subroutine alpha_volume_op(traversal, section, element, Q)
!
  pos1 = samoa_barycentric_to_world_point(element%transform_data,
      [0.0 GRID_SR, 0.0 GRID_SR]) * cfg%scaling + cfg%offset
  pos2 = samoa_barycentric_to_world_point(element%transform_data,
      [1.0 GRID_SR, 0.0 GRID_SR]) * cfg%scaling + cfg%offset
  pos3 = samoa_barycentric_to_world_point(element%transform_data,
      [0.0 GRID_SR, 1.0 GRID_SR]) * cfg%scaling + cfg%offset
  mincoord = min(pos1(1), pos2(1))
  mincoord = min(mincoord, pos3(1))
  maxcoord = max(pos1(1), pos2(1))
  maxcoord = max(maxcoord, pos3(1))
!
  if (mincoord <= 0.0 GRID_SR .and. maxcoord >= 0.0 GRID_SR) then
    traversal%i_refinements_issued = traversal%i_refinements_issued + 1
  end if
!
end subroutine
```
B References


