Delft University of Technology
Faculty Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

The application of differentiable programming frameworks to computational fluid dynamics

De toepassing van differentieerbare programmeer kaders op numerieke stromingsleer

A thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfillment of the requirements

for the degree

BACHELOR OF SCIENCE
in
Applied Mathematics

by

Bart Vos

Heteren, Nederland
February 2021

BSc Thesis Applied Mathematics

“The application of differentiable programming frameworks to computational fluid dynamics”

”De toepassing van differentieerbare programmeer kaders op numerieke stromingsleer”

Bart Vos

Technische Universiteit Delft

Super

Prof. Dr. ir. M. Verlaan Dr. J.D. Nuttall (Deltares)

Further committee members

Dr. N.V. Budko

February, 2021 Heteren

Abstract

In recent years many automatic differentiable programming frameworks have been developed in which numerical programs can be differentiated through automatic differentiation (AD). Examples of these frameworks are Theano, TensorFlow and Pytorch. These frameworks are widely used in Machine Learning. AD also finds applications in the field of computational fluid dynamics (CFD). It is used to develop discrete adjoint CFD code for research concerning for instance sensitivity analysis, data assimilation and design optimization. However, the use of the automatic differentiable programming frameworks in the field of CFD is limited. One can find some examples in the literature on how to find a numerical solution to an initial value problem using a differentiable programming framework. In this work it will be clarified how one can implement an semi-implicit time integration scheme for a staggered grid to simulate the propagation of long waves in water with a free surface in TensorFlow. A main advantage of the automatic differentiable programming frameworks is the user friendly application programming interface (API) for AD. No research has been conducted to use this API in the field of CFD. In this work an example will be given how one can use TensorFlow for research concerning sensitivity analysis. AD requires a significant allocation of memory on a CPU/GPU when working with fine meshes and/or long simulations and since CPU/GPU memory is finite, the method checkpointing is proposed to make it feasible to perform sensitivity analysis when working with fine meshes and/or long simulations. Another main advantage of the differentiable programming framework TensorFlow is the use of compute unified device architecture (CUDA) of a NVIDIA GPU in order to perform computations in parallel, which results in a significant reduction in computation time. A Benchmark will be given that indicates the computational efficiency of TensorFlow compared to a loop over grid implementation in NumPy and a Fortran CPU scalar implementation.

Acknowledgements

I would like to express my special thanks to my supervisors Prof. Dr. ir. Martin Verlaan and Dr. Jonathan D. Nuttall to introduce me to concept behind TensorFlow. Without their help I would not have been able to explore how TensorFlow could be used in the field of Computational Fluid Dynamics. I would especially like to thank them for supporting me through these though times and rough personal circumstances I had to cope with. In addition, I would like to thank Dr. Neil V. Budko for taking the time to read an review the report.

Contents

List of Symbols

Δt	Timestep $[s]$.
Δx	Spacestep in x direction for central differences $[m]$.
Δy	Spacestep in y direction for central differences $[m]$.
$\zeta(\mathbf{x}, t)$	Water level, or the elevation of the free surface $[m]$.
c	Longwave speed, equals $\sqrt{gH}$ $[ms^{-1}]$.
c_f	Friction due to bottom roughness.
$d(\mathbf{x})$	Bottom level $[m]$.
f	Coriolois coefficient, equals about 10^{-4} $[rads^{-1}]$.
g	Gravitational acceleration, equals $9.81[ms^{-2}]$.
H	Mean Height of basin $[m]$.
$h(\mathbf{x})$	Water depth $[m]$.
$J(x, p)$	Cost function
L	Length of basin $[m]$.
$L(x, p)$	Forward model
M	Memory in $[B]$.
$Mtensor_i$	Memory allocated to a tensor $[B]$.
N_x	Number of gridpoint in the x-axis.
N_y	Number of gridpoint in the y-axis.
N_c	Number of checkpoints.
N_{steps}	Number of time steps.
T	Time span of simulation $[s]$.
$u(\mathbf{x}, t)$	The depth-averaged fluid velocity in the x direction, or zonal velocity $[ms^{-1}]$.
$v(\mathbf{x}, t)$	The depth-averaged fluid velocity in the y direction, or meridional velocity $[ms^{-1}]$.
W	Width of basin $[m]$.

1 Introduction

In this chapter, the motivation for the application of the software library TensorFlow [10] to find solutions to initial value problems and perform sensitivity analysis is given, the thesis statement is introduced and the structure of this thesis is outlined.

1.1. Motivation

In recent years, many differentiable programming frameworks have been developed in which numerical programs can be differentiated through automatic differentiation (AD). Examples of these frameworks are TensorFlow, Theano and Pytorch [10][24][22]. These frameworks provide users an accessible and readable syntax for machine learning purposes. Another advantage of these differentiable programming frameworks is the use of Compute Unified Device Architecture (CUDA) of NVIDIA GPUs allowing to perform computations in parallel, which results in a significant acceleration of computing applications [21].

Automatic differentiation also finds applications in the field of computational fluid dynamics (CFD). CFD uses numerical analysis to solve problems that involve fluid flow. These problems can be found in fields of study like aerospace engineering, environmental engineering, weather simulation, etc. Examples of such problems in the field of weather simulation and environmental engineering are the modelling of water flow in rivers, channels and oceans or propagation of a tsunami or flood waves. These flows can mathematically be represented by the shallow water equations (SWE).

In the field of computational fluid dynamics automatic differentiation is applied to develop discrete adjoint CFD Code which is used in research concerning for instance sensitivity analysis, data assimilation and design optimization. In 2009 Souhar and Faure used automatic differentiation in order to asses uncertainties in flood modelling [23]. They used the AD engine TAPENADE that produces discrete adjoint CFD Code, which returns the tangent and adjoint differentiated program of an arbitrary Fortran77, Fotran95 or C code.

In order to use the application programming interface (API) GradientTape for AD in TensorFlow for research purposes in the field of weather simulation and environmental engineering it is necessary to develop a PDE solver in TensorFlow that can simulate water flow. Daoust, Lamberta and Abhinavsp produced a small example of a PDE solver on Github in 2019, which simulated the falling of rain droplets on a pond [6]. This shows the possibility to implement a PDE solver in TensorFlow for a CFD model describing the flow below a pressure surface in a fluid.

The API for AD in TensorFlow has been designed for research concerning machine learning. Up to this point no research has been conducted to use the API GradientTape for AD for research in the field of CFD. Where developing adjoint code that is able to perform computations in parallel GPU can become quite extensive process, the API GradientTape makes use of the full processing power of the GPU when performing adjoint calculations without having to perform manual labour. As such this paper will investigate if it is feasible to use the AD tool for sensitivity analysis.

1.2. Thesis statement

In this thesis, a partial differential equation solver is implemented for the shallow water equations, which harnesses the power of the GPU. Moreover, tools from the TensorFlow library will be applied to compute the sensitivity of the initial input parameters to final numerical solution.[10] This leads me to my research question: Is it possible to obtain an efficient partial differential equation solver with the help of the differentiable programming framework TensorFlow and can this software package be used to perform sensitivity analysis to initial value problems?

1.3. Thesis outline

In chapter 2 the shallow water equations in the one- and two dimensional case are introduced, as well as their numerical discretization. Next to that, the linearization of these equations are given. In addition, an initial value problem that represents a gaussian disturbance will be presented for both the one- and two dimensional case. At the end of this chapter, an analytical solution is derived for the one dimensional initial value problem. In chapter 3, the focus lies on how one can obtain a partial differential equation solver. First, a short introduction will be given on the concept of TensorFlow and some key functions. Hereafter, it will be explained how one can apply the finite difference method in TensorFlow through convolution. In addition, two methods on how to deal with boundary conditions are described. Furthermore, it will be treated how one can implement a solver based upon a staggered grid in TensorFlow. Finally two algorithms are given to solve the one- and two dimensional initial value problems. Chapter 4 explains how one can perform sensitivity analysis. A short introduction is given on adjoint equations and on automatic differentiation. Additionally, a checkpointing method is described to run adjoint simulations for larger grids. After all this, in chapter 5 the numerical results are presented and a benchmark with solvers implemented in Fortran and NumPy is displayed. At last, in chapter 6, the conclusions are made and recommendations for further studies are given in chapter 7.

2
Shallow water equations

In this chapter the one- and two dimensional shallow water equations (SWE) are introduced. Furthermore, to check the correctness of the numerical approximation, analytical solutions are required. To obtain these analytical solutions the shallow water equations will be linearized. In addition, an initial value problem is given for both the one- and two dimensional case, for whom a partial differential equations solver will be created in chapter 3. To obtain this, a discretization of the linearized SWE will be derived. To conclude, an analytical solution is derived through d'Alamberts solution.

2.1. Two Dimensional shallow water equations

The shallow water equations are a set of hyperbolic differential equations that describe the propagation of long waves in water with a free surface. The shallow water equations are derived from the Navier-Stokes Equations [25]. The Navier-Stokes equations are derived from mass and momentum conservation and consist of the continuity equation and the momentum equation(s). Since the oceans, lakes and rivers are in general much wider and longer than they are deep, one can make the general assumption that water flow is essentially depth averaged. Under this assumption, conservation of mass implies that the vertical fluid velocity is small. After depth integrating one can remove the vertical velocity from the equation. In addition, from the momentum equation it can be shown that the vertical acceleration is also small, which implies that the vertical pressure gradients are nearly hydrostatic. This means the that pressure is proportional to the depth of the water. Thus the shallow water equations are derived.

The depth averaged zonal- and meridional flow velocities $u(x,y,t)$ and $v(x,y,t)$ and the water level $\zeta(x,y,t)$ can be computed by solving the shallow water equations. For a two dimensional basin the shallow water equations are denoted as follows [25]

$$\frac{\partial \zeta}{\partial t} + \frac{\partial hu}{\partial x} + \frac{\partial hv}{\partial x} = 0, \tag{2.1a}$$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} - fv + g\frac{\partial h}{\partial x} + c_f\frac{u}{h}\sqrt{u^2+v^2} = 0, \tag{2.1b}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + fu + g\frac{\partial h}{\partial y} + c_f\frac{u}{h}\sqrt{u^2+v^2} = 0. \tag{2.1c}$$

Here equation 2.1a denotes the continuity equation, with the water depth $h(x,y,t)$ as sum of the water level and bottom level, $h(x,y,t) = \zeta(x,y,t) + d(x,y)$. Equations 2.1b and 2.1c represent respectively the zonal- and meridional momentum equation. Three constants are present in equation 2.1, the gravitational acceleration $g = 9.81ms^{-2}$, f the Coriolis coefficient associated with the Coriolis force and a dimensionless friction coefficient c_f due to bottom roughness.

2.1.1. Linearization of the two dimensional shallow water equations

In this work only linear equations will be treated, as such equation 2.1 must be linearized. In most cases, the terms which represent bulk advection that are quadratic in u and v are small in comparison to other terms. Since the friction term is still an approximation of the actual friction, the friction term $c_f \frac{u}{h}\sqrt{u^2+v^2}$ is replaced by $c_f u$ and $c_f \frac{u}{h}\sqrt{u^2+v^2}$ is replaced by $c_f v$. In addition, the Coriolis force is neglected. Moreover, assuming that wave height is significantly smaller than the mean height of the basin, the equations that are considered in this paper are obtained [25]

$$\frac{\partial \zeta}{\partial t} + H(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial x}) = 0, \tag{2.2a}$$

$$\frac{\partial u}{\partial t} + g\frac{\partial \zeta}{\partial x} + c_f u = 0, \tag{2.2b}$$

$$\frac{\partial v}{\partial t} + g\frac{\partial \zeta}{\partial y} + c_f v = 0. \tag{2.2c}$$

2.2. One dimensional shallow water equations

The one dimensional shallow water equations were derived by Saint-Venant in 1871 and describe the propagation of a long wave along a certain characteristic [7]. These equations are also referred to as the one dimensional Saint-Venant equations and can be seen as a contraction of the two dimensional shallow water equations 2.3. The derivation is analogous to the derivation of the two dimensional shallow water equations. This paper focuses on a one dimensional channel of length L in the x-space. The depth averaged flow velocity $u(x,t)$ and the water level $\zeta(x,t)$ in this channel can be computed by solving the shallow water equations, also referred to as the Saint-Venant Equations [7]. The set of partial differential equations is written in equation 2.3.

$$\frac{\partial \zeta}{\partial t} + \frac{\partial hu}{\partial x} = 0, \tag{2.3a}$$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + g\frac{\partial \zeta}{\partial x} + c_f\frac{u|u|}{h} = 0. \tag{2.3b}$$

The shallow water equations consist of two partial differential equations, where equation 2.3a denotes the continuity equation. The water depth $h(x,t)$ is again the sum of the water level $\zeta(x,t)$ and the bottom level $d(x)$. Equation 2.3b denotes the momentum equation, where g is again the gravitational acceleration and equals $9.81 ms^{-2}$ and c_f is the dimensionless friction coefficient due to bottom roughness.

2.2.1. Linearization of the one dimensional shallow water equations

In order to analyse the accuracy of the numerical results, the numerical results can be compared to the analytical solution to the shallow water equations. The analytical solution can be obtained through the linearized form of equation 2.3. To linearize equation 2.3, it is assumed that the fluid velocity $u(x,t)$ is small. Furthermore, constant bottom level is assumed, $d(x) = H$. Contrary to the two-dimensional case, the assumption is made that the bottom is friction less, that is $c_f\frac{u|u|}{h} = 0$. This is done compare the numerical solution to the analytical solution, which will be derived in section 2.5, without having to take the effect of friction into account. Finally the product of two small variables is neglected. This leads to the linearized shallow water equations

$$\frac{\partial \zeta}{\partial t} + H\frac{\partial u}{\partial x} = 0, \tag{2.4a}$$

$$\frac{\partial u}{\partial t} + g\frac{\partial \zeta}{\partial x} = 0. \tag{2.4b}$$

2.3. Initial value problems

For both the one- and two dimensional case a similar initial value problem (IVP) will be considered. The IVPs will describe a gaussian disturbance on a free surface.

2.3.1. One dimensional case

A half-closed basin of length L is considered over time T, where the water level is zero at $x = 0$ and there is zero flow at $x = L$. In combination with the linearized shallow water equations 2.4, a problem describing a gaussian disturbance on a free surface satisfies the following initial conditions

$$\zeta(x,0) = f(x) = \frac{5}{\sqrt{\pi}} \exp\left(\frac{(x-\frac{L}{2})^2}{150L}\right), \qquad 0 \le x \le L, \tag{2.5a}$$

$$u(x,0) = g(x) = 0, \qquad 0 \le x \le L. \tag{2.5b}$$

The boundary conditions are described as follows

$$\begin{aligned} \zeta(0,t) &= 0,\ 0 \le t \le T, \\ u(L,t) &= 0,\ 0 \le t \le T. \end{aligned} \tag{2.6}$$

2.3.2. Two dimensional case

For the two dimensional case the focus lies on a basin of length L and width W, in the (x, y)-space, over time T. The initial value problem describing a gaussian disturbance on a surface satisfies the shallow water equations described in equation 2.2 in combination with the following initial conditions

$$\begin{aligned} \zeta(x,y,0) &= \exp\left(\frac{(x-\frac{L}{2})^2}{50L} + \frac{(y-\frac{W}{2})^2}{50W}\right), && 0 \le x \le L, 0 \le y \le W, \\ v(x,y,0) &= 0, && 0 \le x \le L, 0 \le y \le W, \\ u(x,y,0) &= 0, && 0 \le x \le L, 0 \le y \le W. \end{aligned} \tag{2.7}$$

Furthermore, for the boundary conditions zero flow is considered

$$\begin{aligned} u(0,y,t) = u(L,y,t) &= 0, 0 \le y \le W,\ 0 \le t \le T, \\ v(x,0,t) = v(x,W,t) &= 0, 0 \le x \le L,\ 0 \le t \le T. \end{aligned} \tag{2.8}$$

2.4. Discretized shallow water equations

To implement the PDE solver, the discretized form of equations 2.2 and 2.4 must be obtained. In this work the finite difference method is used for the spatial discretization and the semi-implicit Euler method is considered as time integration scheme.

2.4.1. One dimensional Discretization

First, the spatial discretization of equation 2.4 is considered. Since the linear shallow water equations are a coupled system of partial differential equations, one can run into the problem of odd-even decoupling between the fluid velocity and height. This is a discretization error, which can lead to a checkerboard pattern instead of the wave propagating evenly across the area, as mentioned by Mesinger (1973) [20]. To overcome this problem a staggered grid can be used. Models for oceanography and meteorology are frequently based on the staggered Arakawa C-grid [2].

The one dimensional Arakawa C-grid evaluates the fluid velocity and the water level at alternating grid points. Divide the interval $(0, L)$ into N_x subintervals of length $\Delta x = \frac{L}{N_x+0.5}$ and one subinterval of length $\frac{\Delta x}{2}$. For the grid points the notation $x_i = i\Delta x$ is used, where $i \in \{0, 1, \dots, N_x - 1\}$. At the grid points x_i the water level is stored and denoted by $\zeta(x_i) = \zeta_i$. The fluid velocity is stored at the grid points $x_{i+\frac{1}{2}}$. For the fluid velocity grid points the notation $u(x_{i+\frac{1}{2}}) = u_{i+\frac{1}{2}}$ is used. A visual representation of the one dimensional Arakawa C-grid is given in figure 2.1.

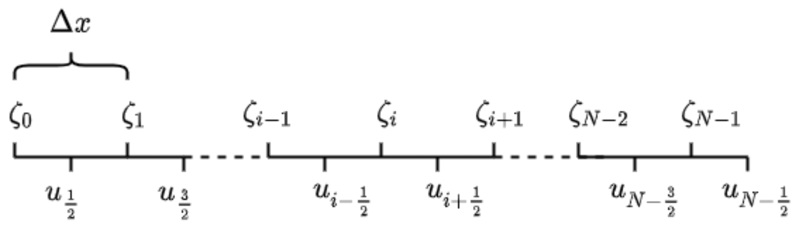

Figure 2.1: Visual Representation of the one dimensional Arakawa C-grid

To obtain the discretization of the spatial derivatives in the linearized shallow water equations 2.4, central finite difference is used [5]. The following semi-discretization is obtained

$$\frac{\partial u_{i-\frac{1}{2}}}{\partial t} = g\frac{\zeta_{i+1} - \zeta_i}{\Delta x}, \tag{2.9a}$$

$$\frac{\partial \zeta_i}{\partial t} = H\frac{u_{i+\frac{1}{2}} - u_{i-\frac{1}{2}}}{\Delta x}. \tag{2.9b}$$

The next step is the time integration scheme. For this, the semi-implicit Euler method is used, as used by E. Haier et al. (2003) [8]. This results in the following discretization

$$\frac{u^{n+1}_{i+\frac{1}{2}} - u^n_{i+\frac{1}{2}}}{\Delta t} = g\frac{\zeta^n_{i+1} - \zeta^n_i}{\Delta x}, \tag{2.10a}$$

$$\frac{\zeta^{n+1}_i - \zeta^n_i}{\Delta t} = H\frac{u^{n+1}_{i+\frac{1}{2}} - u^{n+1}_{i-\frac{1}{2}}}{\Delta x}, \tag{2.10b}$$

in which ζ^{n+1}_i, ζ^n_i, $u^{n+1}_{i+\frac{1}{2}}$ and $u^n_{i+\frac{1}{2}}$ represent the water level and fluid velocity at grid points x_i and $x_{i+\frac{1}{2}}$ at time t_{n+1} and t_n respectively, with $t_n = n\Delta t$. The number of time steps N_{steps} depends on the time span T of the simulation and equals $\frac{T}{\Delta t}$, which is rounded upwards. The index is defined by n and $n \in \{0, 1, \dots, N_{steps} - 1\}$. To obtain the next iteration of the fluid velocity and water level grid points, the discretization of the linearized shallow water equations 2.10 can be rewritten into

$$u^{n+1}_{i+\frac{1}{2}} = u^n_{i+\frac{1}{2}} + g\Delta t\frac{\zeta^n_{i+1} - \zeta^n_i}{\Delta x}, \tag{2.11a}$$

$$\zeta^{n+1}_i = \zeta^n_i + H\Delta t\frac{u^{n+1}_{i+\frac{1}{2}} - u^{n+1}_{i-\frac{1}{2}}}{\Delta x}. \tag{2.11b}$$

2.4.2. Two dimensional discretization

A similar approach as used for the one dimensional case is considered. Similar to the one dimensional case the staggered Arakawa C-grid is used [2]. In the two dimensional rectangular basin of length L and width W set N_x nodes to store ζ in the x-space and N_y nodes in the y-space. Then the grid consist of $(N_x \times N_y)$ cells, which are of length $\Delta x = \frac{L}{N_x}$ and width $\Delta y = \frac{W}{N_y}$. For the grid points the notation $x_{i,j} = (i\Delta x, j\Delta y)$ is introduced, where $i \in \{0, 1, \dots, N_x - 1\}$ and $j \in \{0, 1, \dots, N_y - 1\}$. The water level is stored at the center of each cell and is denoted by $\zeta(x_i, y_j) = \zeta_{i,j}$. The zonal fluid velocity points are stored at the centers of the left/right faces of cell and are denoted by $u(x_{i+\frac{1}{2}}, y_j) = u_{i+\frac{1}{2},j}$. In addition, additional nodes are added to store the zonal fluid velocity along the left boundary, denoted as $u(x_{i-\frac{1}{2}}, y_j) = u_{-\frac{1}{2},j}$, for $j \in \{0, 1, \dots, N_y - 1\}$. The meridional fluid velocity points are at the centers of the upper/lower faces of the grid cell, and are denoted by $v(x_i, y_{j+\frac{1}{2}}) = v_{i,j+\frac{1}{2}}$. At the lower boundary additional nodes are placed to store the meridional fluid velocity, denoted as $v(x_i, y_{-\frac{1}{2}}) = v_{i,-\frac{1}{2}}$, where $i \in \{0, 1, \cdots, N_x - 1\}$. In figure 2.2 a part of the 2D Arakawa C-grid is displayed.

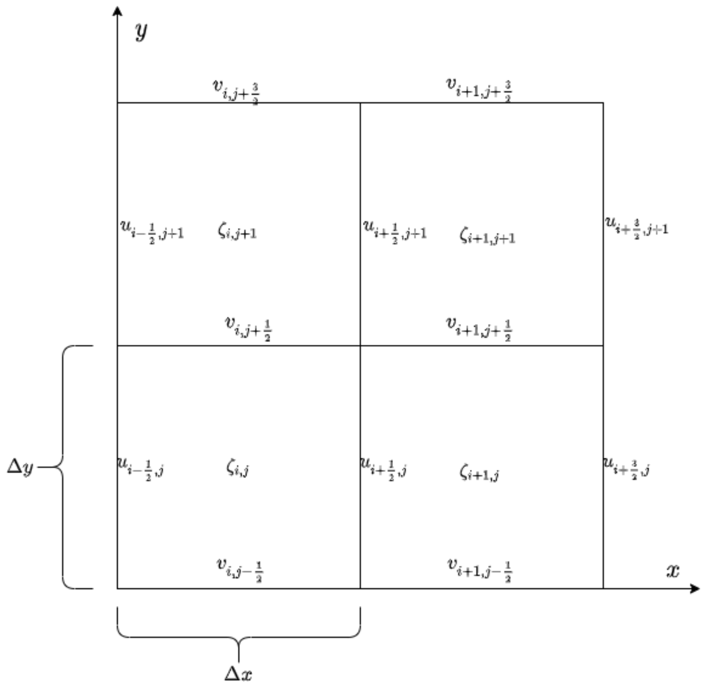

Figure 2.2: Visual representation of a part of the 2D Arakawa C-grid

As for the one dimensional case, the central finite difference method [5] is applied to discretize the spatial derivatives. Applying this method results in the following semi-discretized equations

$$
\begin{aligned}
\frac{\partial u_{i+\frac{1}{2},j}}{\partial t} &= -g\frac{\zeta_{i+1,j} - \zeta_{i,j}}{\Delta x} - bu_{i+\frac{1}{2},j}, \\
\frac{\partial v_{i,j+\frac{1}{2}}}{\partial t} &= -g\frac{\zeta_{i,j+1} - \zeta_{i,j}}{\Delta y} - bv_{i,j+\frac{1}{2}}, \\
\frac{\partial \zeta_{i,j}}{\partial t} &= -H\left(\frac{u_{i+\frac{1}{2},j} - u_{i-\frac{1}{2},j}}{\Delta x} + \frac{v_{i,j+\frac{1}{2}} - v_{i,j-\frac{1}{2}}}{\Delta y}\right).
\end{aligned}
\tag{2.12}
$$

Moving forward, the time integration scheme is looked at. An extension of the semi-implicit Euler method is used [8]. Similar notation concerning the time discretization is used as for one-dimensional discretization. This method gives the following results

$$\begin{aligned}
\frac{u^{n+1}_{i+\frac{1}{2},j} - u^{n}_{i+\frac{1}{2},j}}{\Delta t} &= -g\frac{\zeta^{n}_{i+1,j} - \zeta^{n}_{i,j}}{\Delta x} - bu^{n}_{i+\frac{1}{2},j}, \\
\frac{v^{n+1}_{i,j+\frac{1}{2}} - v^{n}_{i,j+\frac{1}{2}}}{\Delta t} &= -g\frac{\zeta^{n}_{i,j+1} - \zeta^{n}_{i,j}}{\Delta y} - bv^{n}_{i,j+\frac{1}{2}}, \\
\frac{\zeta^{n+1}_{i,j} - \zeta^{n}_{i,j}}{\Delta t} &= -H\left(\frac{u^{n+1}_{i+\frac{1}{2},j} - u^{n+1}_{i-\frac{1}{2},j}}{\Delta x} + \frac{v^{n+1}_{i,j+\frac{1}{2}} - v^{n+1}_{i,j-\frac{1}{2}}}{\Delta y}\right).
\end{aligned} \tag{2.13}$$

Rewriting the set of equations 2.13, one can obtain the value of the fluid velocity points and the water level points at the next iteration in the following way

$$u^{n+1}_{i+\frac{1}{2},j} = (1-\Delta tb)u^{n}_{i+\frac{1}{2},j} - g\Delta t\frac{\zeta^{n}_{i+1,j} - \zeta^{n}_{i,j}}{\Delta x}, \tag{2.14a}$$

$$v^{n+1}_{i,j+\frac{1}{2}} = (1-\Delta tb)v^{n}_{i,j+\frac{1}{2}} - g\Delta t\frac{\zeta^{n}_{i,j+1} - \zeta^{n}_{i,j}}{\Delta y}, \tag{2.14b}$$

$$\zeta^{n+1}_{i,j} = \zeta^{n}_{i,j} - H\Delta t\left(\frac{u^{n+1}_{i+\frac{1}{2},j} - u^{n+1}_{i-\frac{1}{2},j}}{\Delta x} + \frac{v^{n+1}_{i,j+\frac{1}{2}} - v^{n+1}_{i,j-\frac{1}{2}}}{\Delta y}\right). \tag{2.14c}$$

2.4.3. CFL-criterion

In general, the parameters of a numerical simulation cannot be chosen arbitrarily. Δx ($,\Delta y$) and Δt need to satisfy the CFL-criterion, see (Courant, R.; Friedrichs, K.; Lewy, H., 1928)[18]. This criterion ensures that the numerical solution has access to all the point sources that physically influence this solution. The condition is described by the following equation in a one dimensional grid

$$\frac{c\Delta t}{\Delta x} \leq 1, \tag{2.15}$$

where c denotes the long wave speed. For a two dimensional grid, the CFL-criterion is defined as follows

$$\frac{c\Delta t}{\Delta x} + \frac{c\Delta t}{\Delta y} \leq 1. \tag{2.16}$$

2.5. Analytical Solution

The linearized shallow water equations 2.4 can be rewritten into a wave equation, of which the general solution is known as d'Alembert's solution [15]. D'Alembert's solution consists of a right and left traveling wave. For a general wave equation

$$\frac{\partial^2 \omega}{\partial t^2} = c^2\frac{\partial^2 \omega}{\partial x^2},$$

subject to the initial conditions

$$\begin{aligned}
\omega(x,0) &= k(x), \\
\frac{\partial \omega(x,0)}{\partial t} &= l(x),
\end{aligned}$$

where $k(x)$ and $k(x)$ are arbitrary functions, then d'Alemberts solution is

$$\omega(x,t) = \frac{1}{2}(k(x+ct) + k(x-ct)) + \frac{1}{2c}\int_{x-ct}^{x+ct} l(s)ds. \tag{2.17}$$

To obtain a wave like equation of the linearized continuity equation 2.4a take the partial derivative with respect to t, which results in

$$\frac{\partial}{\partial t}(\frac{\partial \zeta}{\partial t}) + \frac{\partial}{\partial x}(H\frac{\partial u}{\partial t}) = 0. \tag{2.18}$$

Rewrite the linearized momentum equation 2.4b to obtain the identity

$$\frac{\partial u}{\partial t} = -g\frac{\partial \zeta}{\partial x}.$$

Consequently, substituting into equation 2.18 induces

$$\frac{\partial^2 \zeta}{\partial t^2} - gH\frac{\partial^2 \zeta}{\partial x^2} = 0. \tag{2.19}$$

Equation 2.19 is indeed a wave equation and the long wave speed c equals $\sqrt{gH}$. Since in the initial conditions of the initial value problem 2.5 don't describe an initial condition for the rate of change of water level over time, we can find $\frac{\partial \zeta}{\partial t}(x,0)$ through the linearized continuity equation 2.4a. Rewriting gives

$$\frac{\partial \zeta}{\partial t} = -H\frac{\partial u}{\partial x}.$$

Differentiating the initial condition 2.5b with respect to x, gives

$$\frac{\partial u}{\partial x} = 0,$$

so

$$\frac{\partial \zeta}{\partial t} = 0.$$

Using this result and initial condition 2.5a one finds that the solution for the water level, according to d'Alembert's solution, satisfies

$$\zeta(x,t) = \frac{1}{2}\left[f(x+ct) + f(x-ct)\right] \tag{2.20}$$

Analogous to the analytical solution for ζ, the analytical solution for u to the IVP can be found, only take the time derivative of the linearized momentum equation 2.4b and rewrite the linearized momentum equation 2.4b. For the initial conditions, use

$$\frac{\partial u}{\partial t} = -g\frac{\partial \zeta}{\partial x}.$$

The partial derivative of ζ with respect to x can be found by taking the derivative of equation 2.20. Combing the latter and the initial velocity condition 2.5b, inline with d'Alembert's solution an analytical solution for the fluid velocity is

$$u(x,t) = \frac{\sqrt{g}}{2\sqrt{H}}\left[f(x-ct) - f(x+ct)\right]. \tag{2.21}$$

3
PDE simulation in TensorFlow

The aim of this chapter is to explain how one can implement an initial value problem solver in TensorFlow. Since TensorFlow has been designed for machine learning, the concept of TensorFlow and some key functions will be explained. Afterwards, a description will be given how these concepts can be used to develop a PDE solver in the TensorFlow framework. Finally, an algorithm to implement a PDE solver is produced for both the one dimensional and two dimensional case

3.1. TensorFlow

TensorFlow is an open-source library for machine learning developed by Google Brain, the deep learning artificial intelligence research team at Google [10]. It has been designed to provide users with an accessible and readable syntax for machine learning. It is based upon dataflow and differentiable programming, the latter will be described in chapter 4. Dataflow programming allows users to build computational graphs, that show how information moves through a directed graph, which contains a set of nodes connected through edges. Each node represents a mathematical operation and each edge carries a tensor, which is an multidimensional array, and represents the data dependencies between the nodes. Some more information on tensors will be given in section 3.1.2. A relatively simple example of a computational graph is shown in figure 3.1, where arithmetic operations are performed on three inputs X,Y and 10 and produces one output C.

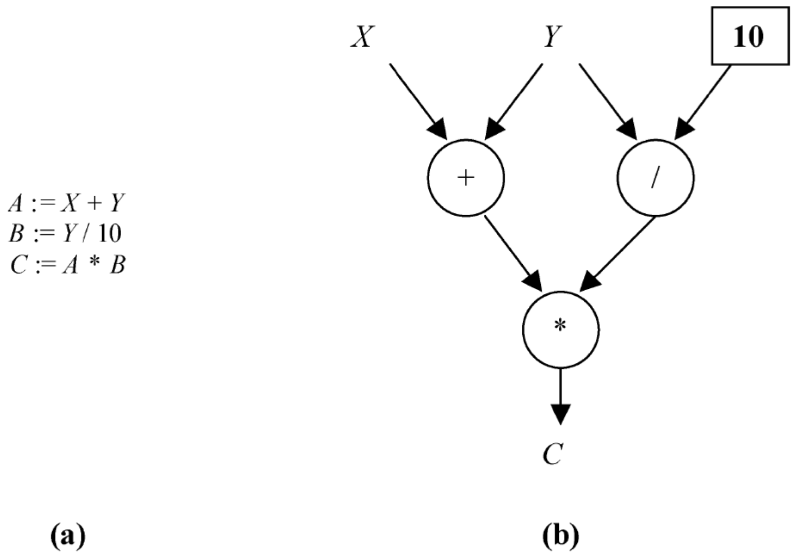

Figure 3.1: A simple implementation and the dataflow representation. Source Johnston et al. 2004, fig 1. [17]

In figure 3.1 only arithmetic operations are considered at the nodes inside the computational graph. However, these arithmetic operations don't suffice to implement complicated machine learning models. To implement machine learning models one might need several functions e.g. convolution functions and activation functions. TensorFlow allows the users to call the required functions for implementing machine learning models. As a consequence, the user can focus on the overall logic of the model instead of focusing on the details implementing the model. An example of this is given in listings 3.1 and 3.2, where one convolution operation requires a for loop in listing 3.1, TensorFlow in listing 3.2 requires just one function call.

```
x = np.ones(N_x)
y = np.zeros(N_x-2)
k = np.array([1.,2.,3.])
for i in range(1,len(x)-2):
    y[i] = np.dot([x[i-1],x[i],x[i
+1]],k)
```

Listing 3.1: Convolution NumPy

```
x = tf.ones([1,N_x,1])
k = tf.Variable([[[1.],[2.],[3.]]])
y = tf.nn.conv1d(x,k,stride = 1,
padding = "VALID")
```

Listing 3.2: Convolution TensorFlow

A huge upside of TensorFlow is that TensorFlow provides a user-friendly front-end application programming interface (API) through Python for building these computational graphs, while executing the mathematical operations at the nodes inside the computational graph in highly-optimized C++ code [10]. In addition to performing these mathematical operations in C++, the mathematical operations also make use of the compute unified architecture (CUDA) of a NVIDIA graphics processing unit (GPU) without having to perform manual labour in order to run processes in parallel, which will be explained in the next section, in order to speed up computations.

3.1.1. Parallel computing

One of the advantages of the implementation of a PDE solver in TensorFlow [10] is the ability to run computations in parallel on the GPU without having to manually activate the GPU by adapting the code. Parallel computing breaks up a particular computation into much smaller independent computations, which can be processed simultaneously, speeding up computations. The resulting computations are then recombined in an overall output. Where a central processing unit (CPU) consists of few strong processing cores clocked at 2 to 3 GHz, a GPU is built up of up to potentially thousands of weak processing cores with a much lower clock speed [9][19]. Due to the strong processing cores a CPU is ideal for performing tasks sequentially[19]. Having said that, being a multiple processing core system, a GPU is ideal to run large scale computations in parallel.[9]

3.1.2. Tensors

As mentioned before, at each edge inside the graph a tensor is stored. A tensor is a multidimensional array. An important property of tensors is that they lack set-index operators, as a consequence of the dataflow programming structure. The structure of a tensor can be explained on the basis of a few definitions. The *rank* of a tensor defines the number of dimensions. An array in a certain rank, is commonly defined as *axis*. The number of indices that are present in an axis, is defined as the *length* of the axis. Combining all the previous, this leads to the *shape* of a tensor. The shape of a tensor defines the length of each axis [12]. For example, let $\boldsymbol{y}$ be the tensor that contains the grid points. In the one dimensional case $\boldsymbol{y}$ is a vector and a tensor of rank 1 and of shape (N_x). In the two dimensional case $\boldsymbol{y}$ is a $(N_x \times N_y)$ matrix and a tensor of rank 2 and of shape (N_x, N_y).

The shape of a tensor is of great importance in TensorFlow, since functions in TensorFlow require input with specific shapes. If a tensor is used as argument in a function with an unsupported shape, an invalid argument error will arise. An invalid argument error can be avoided by *reshaping* the input tensors. Reshaping alters the shape of the tensors, without modifying the order of the indices. TensorFlow offers the function *tf.reshape()*, which takes as input the tensor to be reshaped and the desired shaped.

3.1.3. Convolution

Convolution lays at the foundation of convolutional neural networks in machine learning. Mathematically, convolution is defined as the operation on two functions resulting in a third function. In terms of machine learning convolution is widely used to extract information from an image. The pixel values of an image are stored in a tensor. Then the tensor is modified by a filter, also referred to as kernel. This filter slides to every position of the tensor and computes the dot product of the filter and the indices it floats over, resulting in a new tensor [10]. In TensorFlow one can perform convolution through the function *tf.nn.conv1d()* in the one dimensional case and *tf.nn.conv2d()* in the two dimensional case. The process of convolution is visualized in figure 3.2.

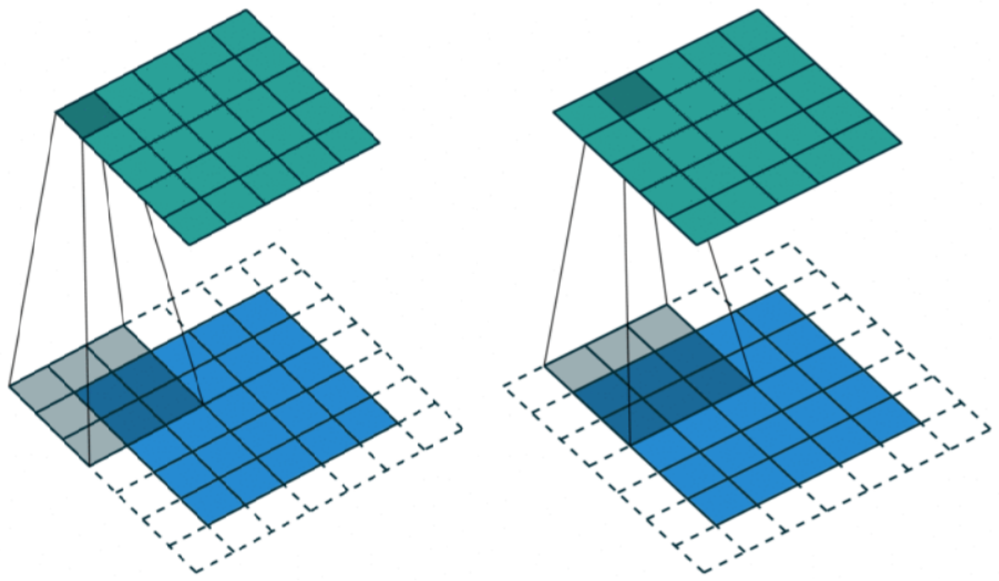

Figure 3.2: Example of convolution. The blue two dimensional plane denotes the input, and the shaded part on top of the blue represents a 3x3 convolution filter. The green two dimensional plane is the output channel. Source: Deeplizard [1] .

Note, in figure 3.2 it can be seen that the convolution operation is an independent operation. Consequently, all these computations can be performed in parallel and an overall output channel can be produced on a GPU. This results in an acceleration of computation speed. Furthermore, one can see that the filter moves a single index forward in figure 3.2. The number of indices the filters moves is defined by the argument *strides*. Next to strides, one can see a column and a row are added to the sides. This is known as *padding*, which will be explained in the following section.

3.1.4. Padding

Padding is a synonym for appending and prepending your input tensor with some value. This can be achieved in TensorFlow with the function *tf.pad()*. The convolution functions have a padding argument, which can either be set to *"VALID"* or *"SAME"*. *"VALID"* means no padding and *"SAME"* will append zeros to either end of the tensor prior to performing convolution, in order to return a tensor that is of similar shape. A user can manually append values with the help of the function *tf.pad()*. This function has multiple modes, however only the mode *"CONSTANT"* is used. With this mode the user can append one or multiple arbitrary value(s) to either end of a tensor.

3.2. Finite difference method

To obtain the value of the grid point at the next time step in a time integration scheme, the spatial derivative needs to be computed, this can be done through the finite difference method. To fit this into the TensorFlow framework, one can use convolution in order to evaluate derivatives. Instead of using tensors to store images, tensors are used to store the initial values of the grid points. In order to evaluate the derivative through convolution, a convolution filter that represents the finite difference method must be defined. For first order derivatives the finite differences operators are [16]

- Forward difference : $dy/dx \approx \frac{y(x+\Delta x)-y(x)}{\Delta x}$.
- Backward difference : $dy/dx \approx \frac{y(x)-y(x-\Delta x)}{\Delta x}$.
- Central difference : $dy/dx \approx \frac{y(x+\Delta x)-y(x-\Delta x)}{2\Delta x}$.

Storing the values in a tensor $y(i\Delta x)$, where $i \in \{0, 1, ..., N_x - 1\}$, then the finite differences are written as:

- Forward difference : $dydx[i] \approx \frac{y[i+1]-y[i]}{\Delta x}$.
- Backward difference : $dydx[i] \approx \frac{y[i]-y[i-1]}{\Delta x}$.
- Central difference : $dydx[i] \approx \frac{y[i+1]-y[i-1]}{2\Delta x}$.

This can be written as a convolution operation with the following convolution filters:

- Forward difference : $\begin{bmatrix} 0 & \frac{-1}{\Delta x} & \frac{1}{\Delta x} \end{bmatrix}$.
- Backward difference : $\begin{bmatrix} \frac{-1}{\Delta x} & \frac{1}{\Delta x} & 0 \end{bmatrix}$.
- Central difference : $\begin{bmatrix} \frac{-1}{2\Delta x} & 0 & \frac{1}{2\Delta x} \end{bmatrix}$.

Suppose we want to compute the backward difference using the mentioned filter of the tensor $\boldsymbol{y}$ with shape (N_x). A visual representation of this convolution is given in figure 3.3. Note, in figure 3.3 there are two zeros appended to the tensor $\boldsymbol{y}$. This is a result of the argument padding in the convolution function being set to *"SAME"*, as mentioned in section 3.1.4. Next to padding, it can be observed that the filter moves a single grid point forward in figure 3.3. In case of finite differences, we choose a stride of 1. Furthermore, note that it's not possible to calculate the backward difference of the first grid point, since the cell left of $y(0)$ doesn't exist. This is similar for the forward case, only then for the last grid point. If the central difference is contemplated, it's not feasible to obtain the central differences of both boundary values. The next section explains how the boundary indices can be set correct.

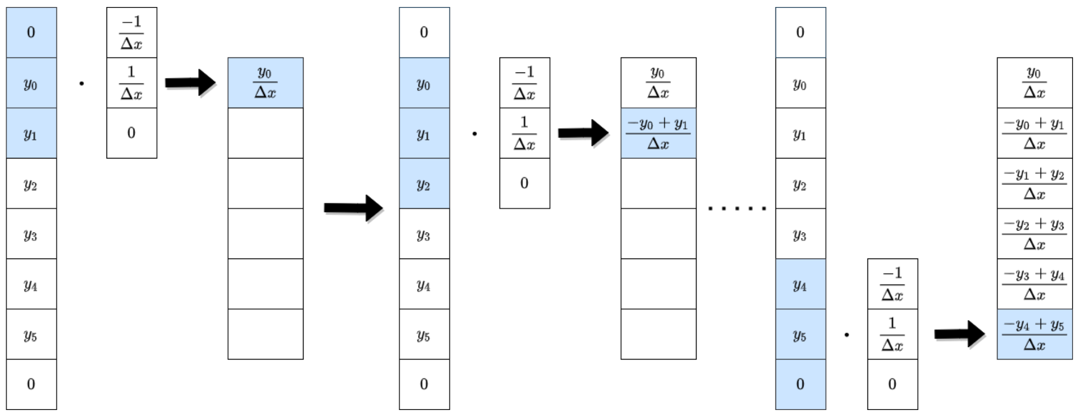
0
y_0
y_1
y_2
y_3
y_4
y_5
0
$\frac{-1}{\Delta x}$
$\frac{1}{\Delta x}$
0
$\frac{y_0}{\Delta x}$
$\frac{-y_0+y_1}{\Delta x}$
$\frac{-y_1+y_2}{\Delta x}$
$\frac{-y_2+y_3}{\Delta x}$
$\frac{-y_3+y_4}{\Delta x}$
$\frac{-y_4+y_5}{\Delta x}$

Figure 3.3: Convolution to obtain finite differences

3.3. Boundary conditions

Since tensors lack set-index operators, the command $y[i] = boundary\ value$ will raise an error in TensorFlow. To overcome this issue, one can use two methods to assign boundary values: Padding and Masks. First, Padding is considered.

3.3.1. Padding

One can use padding in various ways to implement boundary conditions. This work only describes an approach based upon the mode *"CONSTANT"*, since it has less restrictions for implementing boundary conditions compared to the mode *"SYMMETRIC"* or *"REFLECT"*. If one chooses to use padding to implement boundary conditions, the argument padding in the convolution function must be set to *"VALID"*. After performing a convolution operation with a convolution filter that represents the finite differences, a tensor is returned that is "stripped" of the boundary indices. Through the function *tf.pad()* with mode "CONSTANT" one can add the boundary values as defined by the initial value problem. Note, if the case of unequal boundary values it is necessary to pad each edge with the corresponding boundary value.

For instance, consider the same example as given in section 3.2. Suppose, at both boundaries Dirichlet boundary conditions are imposed, $y(0) = \alpha$ and $y(6) = \beta$. After performing the same convolution operation, with padding equal to "VALID", padding twice, one value to each corresponding boundary, will result in the correct tensor. This process is depicted in figure 3.4.

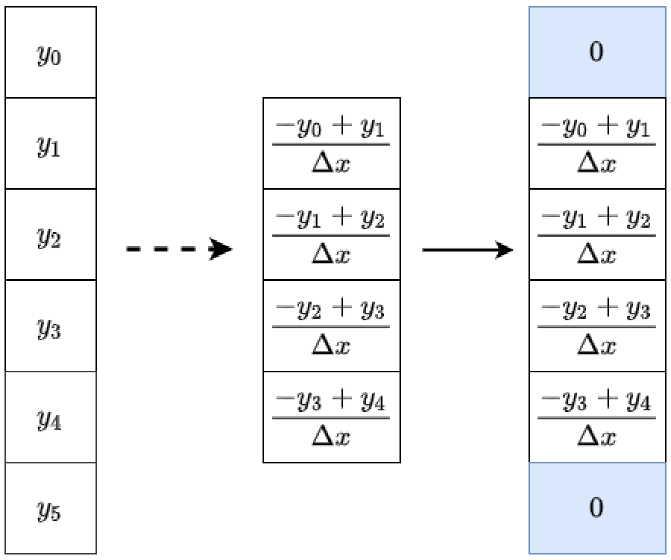
y_0
y_1
y_2
y_3
y_4
y_5
$\frac{-y_0+y_1}{\Delta x}$
$\frac{-y_1+y_2}{\Delta x}$
$\frac{-y_2+y_3}{\Delta x}$
$\frac{-y_3+y_4}{\Delta x}$
0
$\frac{-y_0+y_1}{\Delta x}$
$\frac{-y_1+y_2}{\Delta x}$
$\frac{-y_2+y_3}{\Delta x}$
$\frac{-y_3+y_4}{\Delta x}$
0

Figure 3.4: Padding, after performing convolution, with two constants to either end

3.3.2. Masks

The use of padding has a major disadvantage, padding only works on rectangular domains. So complex shapes, like rivers, are impossible to simulate with padding. If masks are used, two tensors need to be defined, $y_{interior}$ and $y_{boundary}$. $y_{interior}$ is a tensor with equal shape to the input tensor, where all indices equal 1, except for the boundary indices. $y_{boundary}$ is a tensor with equivalent shape to the input tensor, where all indices equal 0, only the boundary indices are nonzero as specified by the initial value problem. Note, if homogeneous Dirichlet boundary conditions are considered, then the boundary index of the mask is either equal to zero or does not need to be defined. When performing convolution, the argument *padding* in the convolution function should be set to "*SAME*", in order to maintain all grid points. After convolution has been performed and the next iteration $\boldsymbol{y}^{n+1}$ has been computed, one should multiply the tensor $\boldsymbol{y}^{n+1}$ with the mask $y_{interior}$ and consequently add the mask $y_{boundary}$.

For example, suppose the left Dirichlet boundary condition is imposed, $y(0) = \alpha$, where α is an arbitrary constant. Define two masks, $y_{interior}$ and y_{bound}, as tensors of length N_x, with the following values

$$y_{interior} = \begin{bmatrix} 0 & 1 & \cdots & 1 \end{bmatrix}, \qquad y_{bound} = \begin{bmatrix} \alpha & 0 & \cdots & 0 \end{bmatrix}.$$

Then, it's possible to assign the boundary value according to

$$\begin{aligned} \boldsymbol{y} &= \boldsymbol{y} * y_{interior} + y_{bound} \\ &= \boldsymbol{y} * \begin{bmatrix} 0 & 1 & \cdots & 1 \end{bmatrix} + \begin{bmatrix} \alpha & 0 & \cdots & 0 \end{bmatrix}. \end{aligned}$$

3.4. Shape

As mentioned before, the shape of a tensor is of great importance in TensorFlow, since certain functions in TensorFlow require input with specific shapes. For instance, using a tensor of shape (N_x, N_y) to store a two dimensional grid as input in the convolution function will raise an invalid argument error. The same error will occur if the convolution filter is defined with an unsupported shape. In order to prevent this error the tensors need to be reshaped.

For the implementation of a PDE solver, only the convolution function demands tensors to be of specific shapes. For the one dimensional case, the tensor containing the grid points ,$\boldsymbol{y}$, is of rank 1 and of shape (N_x) and the convolution filters are tensors of rank 1 and shape (3). The convolution function *tf.nn.conv1d()* requires an input tensor, as well as a convolution filter of at least rank 3. To meet this condition, reshape the tensor $\boldsymbol{y}$ to become a tensor of rank 3 with shape $(1, N_x, 1)$ and reshape the convolution filters to tensors of rank 3 and shape $(3, 1, 1)$. Note, this only has to be done once before the simulation is run.

When the two dimensional case is looked upon, both the input tensor $\boldsymbol{y}$ and the convolution filters must be tensors of rank 4 or higher in the convolution function *tf.nn.conv2d()*. To fulfil the requirement, the tensor $\boldsymbol{y}$ should be reshaped to a tensor of rank 4 and of shape $(1, N_x, N_y, 1)$. The convolution filter representing the spatial derivative with respect to x should be reshaped into a tensor of shape $(1, 3, 1, 1)$ and the convolution filter for representing the spatial derivative with respect to y should be reshaped into a tensor of shape $(3, 1, 1, 1)$.

Note, since the shape of the tensor $\boldsymbol{y}$ has been altered, an issue arises when padding or masks are used to implement the boundary conditions. Since TensorFlow isn't able to perform arithmetic operations between tensors with different shapes, the masks $y_{interior}$ and y_{bound} should be reshaped to be of similar shape as $\boldsymbol{y}$. Note, scalar multiplication will remain possible independent of the shape of the tensor. If the boundary conditions are implemented through padding, one should pay close attention to appending the boundary values to the correct axis in the tensor containing the spatial derivatives. To every other axis in the tensor no values should be appended, but this must be specified by the user. For example, after performing convolution the tensor containing the spatial derivatives $dydx$ is of shape $(1, N_x - 2, 1)$, then a zero should be appended to either end of the second axis and no values should be append to the first and third axis. As input for the function *tf.pad()*, the argument *paddings* should be set to $[[0, 0], [1, 1], [0, 0]]$.

3.5. Staggered Grids

Two problems arise when a PDE solver based upon a staggered grid is implemented. The discretized water level equations 2.11b and 2.14c and the discretized fluid velocity equations 2.11a, 2.14a and 2.14b have different scalars for the central differences. Since it is not possible to alternate between convolution filters, it is necessary to define two (three) separate tensors, $\boldsymbol{\zeta}$, $\boldsymbol{u}$, ($\boldsymbol{v}$,) with each a separate convolution filter.

This leads to the next issue for the two dimensional case. Since the shapes of the tensors $\boldsymbol{\zeta}, \boldsymbol{u}$ and $\boldsymbol{v}$ are different, namely (N_x, N_y), $(N_x, N_y + 1)$ and $(N_x + 1, N_y)$, adding them will raise an invalid argument error. Note there are other approaches to tackle this problem, however the proposed approach will use padding. After performing convolution with a filter containing two entries and padding equal *"VALID"* to obtain the spatial derivatives of $\boldsymbol{\zeta}$ with respect to x, a tensor of shape $(N_x, N_y - 1)$ is produced. After this, appending a single column of zeros to both the right and left of the convolution output if derivative to x is considered, a tensor of shape $(N_x, N_y + 1)$ is obtained, which is equivalent to the shape of tensor $\boldsymbol{u}$. Since the first and last column of the finite difference tensor are equal to zero, the boundary values of the tensor $\boldsymbol{u}$ remain constant. Analogous to the spatial derivative with respect to x, only for the spatial derivative with respect to y of $\boldsymbol{\zeta}$ the output of the convolution is a tensor of shape $(N_x - 1, N_y)$. So, it is necessary to append a row of zeros to the top and bottom of the output tensor.

3.6. Implementation one dimensional PDE solver

The scheme of the linearized SWE in equation 2.11 can be implemented in order to find a solution to the initial value problem with homogeneous boundary conditions. The simulation is run over an arbitrary time span T, which is split into N_{steps} time steps. Two separate tensors are defined, $\boldsymbol{\zeta}^0$ and $\boldsymbol{u}^0$, each of length N_x, in which the initial water- and fluid velocity grid points are stored, as defined in the initial conditions 2.5. Furthermore, the convolution filters to compute the spatial differences of $\boldsymbol{\zeta}$ and $\boldsymbol{u}$ are respectively the forward and backward difference filter, denoted as $diff_f$ and $diff_b$. To implement the boundary conditions masks are used. Since the IVP in section 2.3.1 is a homogeneous problem, only two masks concerning the interior points should be defined. These are $\zeta_{interior}$ and $u_{interior}$, both of length N_x. All these tensors should be reshaped to obtain the desired shape as mentioned in section 3.4. To sum up, the PDE solver is given in algorithm 1. For the implementation see Appendix A.

Algorithm 1: Pseudo Algorithm for a PDE solver with masks

```
Data: $\boldsymbol{\zeta}^0, \boldsymbol{u}^0, \zeta(0) = 0, u(L) = 0, \zeta_{interior}, u_{interior}, diff_f, diff_b$
Result: Numerical solution of $\boldsymbol{\zeta}$ and $\boldsymbol{u}$ at time $T$
Reshape $\boldsymbol{\zeta}^0, \boldsymbol{u}^0, \zeta_{interior}$ and $u_{interior}$ s.t. they are of shape $(1, N_x, 1)$;
Reshape $diff_f$ and $diff_b$ s.t. they are of shape $(3, 1, 1)$.
for n = 0,$\cdots$, $N_{steps} - 1$ do
    $\boldsymbol{\zeta}^n = \boldsymbol{\zeta}^n * \zeta_{interior}$;
    $dhdx$ = convolution of $\boldsymbol{\zeta}^n$ with $diff_f$;
    $\boldsymbol{u}^{n+1} = \boldsymbol{u}^{\boldsymbol{n}} + g\Delta t * d\zeta dx$;
    $\boldsymbol{u}^{n+1} = \boldsymbol{u}^{n+1} * u_{interior}$;
    $dudx$ = convolution of $\boldsymbol{u}^{n+1}$ with $diff_b$;
    $\boldsymbol{\zeta}^{n+1} = \boldsymbol{\zeta}^n + H\Delta t * dudx$.
Reshape $\boldsymbol{\zeta}^{N-1}$ and $\boldsymbol{u}^{N-1}$ s.t. they are of their original shape.
```

3.7. Implementation two dimensional PDE solver

Similar to the one dimensional case, three separate tensors are defined satisfying the initial conditions 2.7, these are: $\boldsymbol{\zeta}^0$, a tensor of shape (N_x, N_y), $\boldsymbol{u}^0$, a tensor of shape $(N_x, N_y + 1)$ and $\boldsymbol{v}^0$, a tensor of shape $(N_x, N_y + 1)$. The simulation is run over a time span T and split up into $N_{steps} = \frac{T}{\Delta t}$ time steps. Padding is used to implement the boundary conditions. Define two convolution filters. For the spatial derivatives of the water level with respect to x and y, as well as for the zonal- and meridional fluid velocity derivatives, the filters are

$$diff_x = \begin{bmatrix} -\frac{1}{\Delta x} & \frac{1}{\Delta x} \end{bmatrix}$$

and

$$diff_y = \begin{bmatrix} \frac{1}{\Delta y} & -\frac{1}{\Delta y} \end{bmatrix}^T .$$

After performing the convolution to $\boldsymbol{\zeta}^i$, appending zeros to the corresponding edges will ensure that the boundary values remain constant. The tensors $\boldsymbol{\zeta}^0, \boldsymbol{u}^0, \boldsymbol{v}^0, diff_x$ and $diff_y$ should be reshaped to be of the mandatory shapes as stated in section 3.4. Algorithm 2 encapsulates the previous. Turn to appendix B for the implementation.

Algorithm 2: Pseudo Algorithm for a PDE solver with padding

```
Data: $\boldsymbol{\zeta}^0, \boldsymbol{u}^0, \boldsymbol{v}^0, u(0,y) = \alpha, u(L,y) = \beta, v(x,0) = \gamma, v(x,H) = \delta, diff_x, diff_y, c_f$
Result: Numerical solution of $\boldsymbol{\zeta}, \boldsymbol{u}$ and $\boldsymbol{v}$ at time $T$
Reshape $\boldsymbol{\zeta}^0, \boldsymbol{u}^0$ and $\boldsymbol{v}^0$ s.t. they are of shape $(1, N_x, N_y, 1)$,$(1, N_x, N_y + 1, 1)$,$(1, N_x + 1, N_y, 1)$
  respectively;
Reshape $diff_x$ and $diff_y$ s.t. they are of shape $(1, 2, 1, 1)$ and $(2, 1, 1, 1)$.
for $n$ = $0, \cdots, N_{steps} - 1$ do
    $d\zeta dx$ = convolution of $\boldsymbol{\zeta}^n$ with $diff_x$;
    Append a single column of zeros to both the right and left of $d\zeta dx$;
    $\boldsymbol{u}^{n+1} = (1 - c_f \Delta t) * \boldsymbol{u}^n - g\Delta t * d\zeta dx$;
    $dhdy$ = convolution of $\boldsymbol{\zeta}^n$ with $diff_y$;
    Append a single column of zeros to both the top and bottom of $d\zeta dy$;
    $\boldsymbol{v}^{n+1} = (1 - c_f \Delta t) * \boldsymbol{v}^n - g\Delta t * d\zeta dy$;
    $dudx$ = convolution of $\boldsymbol{u}^{n+1}$ with $diff_x$;
    $dvdy$ = convolution of $\boldsymbol{v}^{n+1}$ with $diff_y$;
    $\boldsymbol{\zeta}^{n+1} = \boldsymbol{\zeta}^i - H\Delta t * (dudx + dvdy)$;
Reshape $\boldsymbol{\zeta}^{N-1}, \boldsymbol{u}^{N-1}$ and $\boldsymbol{v}^{N-1}$ s.t. they are of their original shape.
```

4 Automatic Differentiation

An outline to the mathematics behind adjoint sensitivity is given in this chapter. Moreover, it will be explained how this adjoint is calculated using Automatic Differentiation (AD). Since calculating the adjoint requires GPU memory, 'checkpointing' will be introduced to make it feasible to run larger adjoint simulations when working with fine meshes. Finally, two algorithms will be proposed to run adjoint equations, of which one implements the checkpointing method.

4.1. Adjoint Sensitivity

In the previous chapter a model was created in order to solve the forward problem. This PDE solver can be denoted by the equation $L(x,p) = 0$, where the algorithms 1 and 2 given input variables p computes the output variables x, which are $u(,v)$ and ζ. For AD it is necessary that the partial derivative $\frac{\partial L}{\partial x}$ is non-singular for all x in the domain.

Then one defines some cost function $J(x,p)$, which produces a scalar output. In sensitivity analysis one wants to compute the differential $\frac{dJ}{dp}$. $\frac{dJ}{dp}$ can be used in many ways, for instance: The sensitivity of the cost function with respect to the input parameters or to find a solution to an gradient based optimization problem $min_p J$ [4]. In this paper the focus lies on computing the sensitivities with respect to the input parameters.

The computation of $\frac{dJ}{dp}$ is analogous to the derivation by Bradley in *"PDE-constrained optimization and the adjoint method"* (2010) [4]. It holds that

$$\frac{dJ}{dp} = \frac{\partial J}{\partial x}\frac{\partial x}{\partial p} + \frac{\partial J}{\partial p}. \tag{4.1}$$

In addition, since $L(x,p) = 0$,

$$\frac{dL}{dp} = \frac{\partial L}{\partial x}\frac{\partial x}{\partial p} + \frac{\partial L}{\partial p} = 0$$

This can be rewritten into,

$$\frac{\partial x}{\partial p} = -\frac{\partial L}{\partial x}^{-1}\frac{\partial L}{\partial p}.$$

Substituting into equation 4.1, gives

$$\frac{dJ}{dp} = -\frac{\partial J}{\partial x}\frac{\partial L}{\partial x}^{-1}\frac{\partial L}{\partial p} + \frac{\partial J}{\partial p}$$

The term $\frac{\partial L}{\partial x}^{-1}\frac{\partial L}{\partial p}$ can be seen as,

$$\frac{\partial L}{\partial x}^{T}\lambda = -\frac{\partial J}{\partial x}^{T}, \tag{4.2}$$

where T is the transpose, λ is the so-called vector of *adjoint variables* and equation 4.3 is called the *adjoint equation*. Then finally in terms of λ one finds the equation in order to determine $\frac{dJ}{dp}$,

$$\frac{dJ}{dp} = \lambda^{T}\frac{\partial L}{\partial p} + \frac{\partial J}{\partial p}. \tag{4.3}$$

Note, if the cost function is independent of the input parameters, the last term in equation 4.3 reduces to zero.

4.2. Automatic Differentiation

Evaluating the term $\frac{\partial L}{\partial p}$ in equation 4.3 is complex. To determine the value of this term AD can be applied. AD is a technique which allows a user to calculate partial derivatives of a numerical function implemented in a computer program. All numerical operations defined in a numerical function can be viewed as a sequence of elementary assignments, such as binary arithmetic and transcendental functions, of which the derivatives are known [3]. Through storage of the operations in this sequence, it is possible to differentiate this sequence by repeated application of the chain rule to obtain the derivative of the output with respect to the input. Since the partial derivatives are found "analytically", AD leads to exact numerical derivatives, with the exception of rounding errors [3]. There are two main modes of AD: Forward and Reverse accumulation mode, also referred to as tangent linear and adjoint mode. Since the aim is to develop adjoint CFD code, the reader is referred to the article *"Automatic Differentiation in Machine Learning: a Survey"*(2017) by Baydin et al. for the tangent linear mode [3].

The evaluation of a numerical function can be stored in a Wenger List, also mentioned as evaluation trace [26]. The notation follows the notation used by Griewank and Walter in their book *"Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation"* (2008) in order to evaluate a numerical function[14]:

- Input variables $v_{i-n} = x_i$ for $i = 0, ..., n$.
- Intermediate/working variables v_i for $i = 1, ..., l$
- Output variables $v_{m-i} = y_{m-i}$ for $i = m-1, ..., 0$

A Wengert list represents the data structure of the intermediate variables v_i as well as the elementary assignments that were performed to obtain these intermediate variables. An example of a Wengert list can be seen in table 4.2. To visualize a Wengert list, a computational graph can also be produced, see figure 4.1.

Adjoint mode AD is a two stage method. In the first stage, the simulation is run forward and the Wengert list is stored in the GPU/CPU memory. In the second stage, a partial derivative of an output variable y_i with respect to the input variable x_i is calculated through repetitive application of the chain rule

$$\frac{\partial y_m}{\partial x_i} = \frac{\partial y_m}{\partial v_i}\frac{\partial v_i}{\partial x} = \frac{\partial y_m}{\partial v_{i+1}}\frac{\partial v_{i+1}}{\partial v_i}\frac{\partial v_i}{\partial x_i} = \dots$$

The derivative of an output variable with respect to a certain intermediate/input variable is also referred to as the adjoint and is commonly denoted as

$$\bar{v}_i = \frac{\partial y_i}{\partial v_i}.$$

In order to get an idea of AD, lets apply it to equation 2.11a. A wengert list can be visualized in a computational graph, see figure 4.1. In order to get the value of $\boldsymbol{\zeta}^{n+1}$, the input variables that are needed are: $\boldsymbol{u}^{n+1}, H, \Delta t, \Delta x, \boldsymbol{\zeta}^{n}$. These input variables are respectively denoted as: $v_{-4}, v_{-3}, ..., v_0$. Since the masks are not used as input variables, they aren't watched and thus the computational graph uses less memory. Then at each working variable an elementary assignment is performed and stored. There are a total of 6 working variables. Finally one output variable is produced, $v_7 = y = \boldsymbol{\zeta}^{n+1}$. The evaluation trace, which contains the corresponding working variables, is shown in table 4.2.

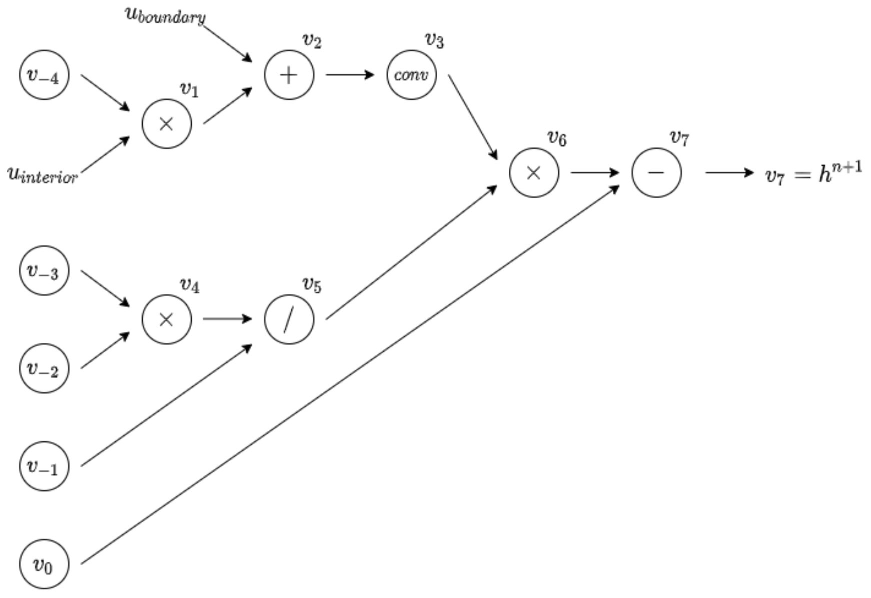

Figure 4.1: Computational graph of numerical equation 2.11a

In the second stage, the reverse adjoint trace is computed, as shown on the right side of table 4.2. For the adjoint of the output it holds that $\overline{y} = 1$. Then by propagating backward though the derivatives, one can obtain the sensitivities with respect to the initial variables.

Forward evaluation trace			Reverse adjoint trace						
v_{-4}	$=$	u^{n+1}	$\overline{v}_{-4}$	$=$	$\overline{v}_1 \frac{\partial v_1}{\partial v_{-4}}$	$=$	$-\frac{H\Delta t}{\Delta x} \times u_{interior}$		
v_{-3}	$=$	H	$\overline{v}_{-3}$	$=$	$\overline{v}_4 \frac{\partial v_4}{\partial v_{-3}}$	$=$	$\frac{1}{\Delta x -2}$	$=$	$\frac{\Delta t}{\Delta x}$
v_{-2}	$=$	Δt	$\overline{v}_{-2}$	$=$	$\overline{v}_4 \frac{\partial v_4}{\partial v_{-2}}$	$=$	$\frac{1}{\Delta x} \times v_{-3}$	$=$	$\frac{H}{\Delta x}$
v_{-1}	$=$	Δx	$\overline{v}_{-1}$	$=$	$\overline{v}_5 \frac{\partial v_6}{\partial v_{-1}}$	$=$	$= -v_3 \times \frac{-1_4}{v_1^2}$	$=$	$-v_3 \frac{H\Delta t}{\Delta x^2}$
v_0	$=$	ζ^n	$\overline{v}_0$	$=$	$\overline{v}_7 \frac{\partial v_7}{v_0}$	$=$	1×1	$=$	1
v_1	$=$	$v_{-4} \times u_{interior}$	$\overline{v}_1$	$=$	$\overline{v}_2 \frac{\partial v_2}{\partial v_1}$	$=$	$\overline{v_2} \times 1$	$=$	$-\frac{H\Delta t}{\Delta x}$
v_2	$=$	$v_{-4} \times u_{boundary}$	$\overline{v}_2$	$=$	$\overline{v}_3 \frac{\partial v_3}{\partial v_2}$	$=$	$-\frac{dtH}{dx} \times 1$	$=$	$-\frac{H\Delta t}{\Delta x}$
v_3	$=$	$conv(v_3)$	$\overline{v}_3$	$=$	$\overline{v}_6 \frac{\partial v_6}{\partial v_3}$	$=$	$-1 \times v_5$	$=$	$-\frac{H\Delta t}{\Delta x}$
v_4	$=$	$v_{-3} \times v_{-2}$	$\overline{v}_4$	$=$	$\overline{v_5} \frac{\partial v_5}{v_4}$	$=$	$1 \times \frac{1}{v_{-1}}$	$=$	$= \frac{1}{\Delta x}$
v_5	$=$	$\frac{v_4}{v_{-1}}$	$\overline{v}_5$	$=$	$\overline{v}_6 \times \frac{\partial v_6}{v_5}$	$=$	$-1 \times v_3$	$=$	$-v_3$
v_6	$=$	$v_3 \times v_5$	$\overline{v}_6$	$=$	$\overline{v}_7 \frac{\partial v_7}{\partial v_6}$	$=$	1×-1	$=$	-1
v_7	$=$	$v_0 - v_6$	$\overline{v}_7$	$=$	1				

Table 4.1: Adjoint AD, where $\boldsymbol{\zeta}^{n+1}$ is evaluated. After running the forward sequence, which is presented on the left, the adjoint operations are run in order to determine the sensitivity with respect to input variables in a reverse pass, starting at $\overline{v}_7$.

Note, it is sometimes unnecessary and/or undesirable to derive the sensitivity of the output variables with respect to some initial variables. For instance, it is of no interest to compute the sensitivity of the cost function with respect to the gravitational acceleration, since it is relatively constant around the earth. Because watching all variables requires lots of memory from the GPU/CPU, which unfortunately is scarce, TensorFlow can be given the assignment of which variables a computational graph should be build. This reduces the memory usage to store the dependencies between the variables, resulting in the possibility to run a longer simulation or/with a finer mesh.

4.2.1. Memory allocation

As mentioned, the memory of a GPU/CPU is finite. It is possible to estimate the amount of memory required to store the Wengert list. We denote the memory used by the variable M in bytes. At each working variable in a Wengert list, a tensor is stored. The memory that is allocated to store this tensor depends on two factors, the shape and the data type of the tensor. A definition of the shape of a tensor has already been given in section 3.1.2. The data type explains how the bytes in memory should be interpreted that are allocated to a tensor item. It describes the *Type* of the data, e.g. integer, float, complex. In addition, the *Size* of the data explains how much bytes are allocated to store the object. In TensorFlow either 16, 32 or 64 bits are allocated to store an integer or floating point. In this research a size of 32 bits was used. 32 bits is equivalent to $4B$. Note, it is possible to use 64 bits to store an integer or floating point, this will come at the cost of higher memory use. On the contrary, it is not desirable to use 16 bits since this will significantly reduce the precision of the model. So, denote the memory allocated to a tensor at an input variable x_i or working variable v_i by M_tensor_i in B. For M_tensor_i it holds that the memory allocated is equal to the product of the length of the axis times the *Size* of the data type as stated in equation 4.4

$$Mtensor_i = 4 * shape. \tag{4.4}$$

The amount of tensors that need to be stored depend on three things. The amount of input variables, denotes this number by n, the amount of working variables created at a single time step denoted this number by l and the amount of time steps N_steps the computation is split. As such, an estimation for the amount of memory used to store the tape is given by

$$M = \sum_{i=1}^{n} Mtensor_i + N_{steps} \sum_{i=1}^{l} Mtensor_i. \tag{4.5}$$

4.3. Checkpointing

Tracking fewer parameters will still be insufficient when running computations with a large number of time steps or using a high number of grid points or cells. To combat the issue of having an inadequate amount of CPU/GPU memory, it is possible to break up the evaluation trace through "*Checkpointing*", as proposed by Griewank and Walther[13]. Note, Griewank and Walther proposed a sophisticated algorithm "*Revolve*", where an optimal trade-off between memory allocation and computation time is achieved. However, in this work only a basic strategy will be implemented to make it feasible to run larger adjoint numerical simulations.

Instead of running the forward model while storing the Wengert list and computing the adjoint instantly with respect to the initial variables, the computation is split in several parts. First, the forward model is run without storing a Wengert list. Along the way N_c checkpoints are set, where the state of the variables $\boldsymbol{u}^k$ (,$\boldsymbol{v}^k$ included if the 2D case is considered) and $\boldsymbol{\zeta}^k$ are stored on a stack at time t^k, where $t^k = k\Delta t\frac{N_{steps}}{N_c}$ and $k \in (0, 1, ..., N_c - 1)$. After the simulation has completed, the last state of variables taken at time t^{N_c-1} are released from the stack. Then the numerical simulation is run from t^{N_c-1} to T, while recording the elementary assignments of the PDE solver. Hereafter, the adjoint is computed from T to t^{N_c-1}. This adjoint is then used, as initial condition for the adjoint solver at t^{N_c-1}, when the procedure is repeated in the interval (t^{N_c-2}, t^{N_c-1}). This procedure is repeated for all intervals (t^k, t^{k+1}), where $k \in (N_c - 2, N_c - 3, ..., 0)$. A sketch of this process when the number of checkpoints $N_c = 4$ is given in figure 4.2.

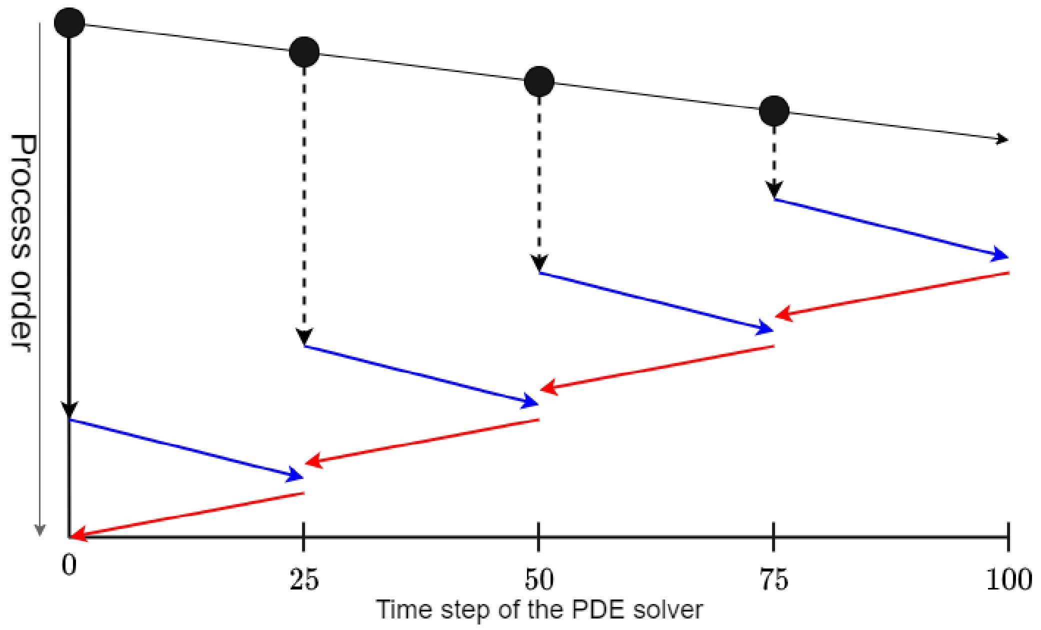

Figure 4.2: Sketch of checkpointing algorithm for 100 time steps, and 4 checkpoints. The black line represent the numerical simulation, on which the dots are the state of the variables stored on a stack. The blue lines, are the forward simulations where the numerical dependencies are stored. On the red lines the adjoint mode is run.

4.3.1. Memory allocation

Since the aim of checkpointing is to reduce memory usage, we can estimate how much is required. The amount of memory allocated to a tensor at each working variable is equivalent as stated in equation 4.4. The memory usage varies on only two points. First, the amount of iterations where the assignments are being recorded is reduced to $\frac{N_{steps}}{N_c}$, which is rounded upwards. Second, storing the state of the variables $\boldsymbol{u}^k$, ($\boldsymbol{v}^k$ included if the 2D case is considered) and $\boldsymbol{\zeta}^k$ at N_c checkpoints on a stack requires additional memory. Let c denote the number of state variables, that is $c = 2$ in the one dimensional case and $c = 3$ in the three dimensional case. To summarize, an estimation for the memory use is given by equation 4.6.

$$M = \sum_{i=1}^{n} Mtensor_i + \frac{N_{steps}}{N_c} \sum_{i=1}^{l} Mtensor_i + N_c \sum_{i=1}^{c} Mtensor_i \tag{4.6}$$

4.4. API Automatic Differentiation TensorFlow

In TensorFlow, the API for AD can be accessed through *tf.GradientTape()* [11]. Within the context of *tf.GradientTape()* the evaluation trace(s) of the initial parameter(s) p in function $L(x,p) = 0$ is (are) stored in the GPU/CPU memory. TensorFlow must be given the command *GradientTape.watch()* in order to store the Wengert list of each specified parameter, which is given as argument. Then with the function *GradientTape.gradient()*, one can compute the differential of the target function, the cost function $J(x,p)$, with respect to a source that is being ”watched”, an initial parameter p. Note, if the user wants to compute multiple differentials in a single evaluation, one must create a *persistent* tape through the argument *persistent=True*. Otherwise, after a single call on the tape the Wengert list will be discarded and an additional call on the tape will raise an error.

4.5. Implementation single backward pass

In sections 3.6 and 3.7 algorithms 1 and 2 were provided to evaluate u (, v) and ζ. In order to perform sensitivity analysis, perform the forward simulation within the context of the GradientTape. If it is desired to asses the sensitivity with respect to multiple inputs it's necessary to create a persistent tape. Before performing the forward simulation, specify which initial parameters p need to be watched. Hereafter, run the forward simulation as stated in algorithms 1 and 2. After the forward simulation is completed, define the cost function $J(x,p)$. Note, this still has to be done inside the context of the GradientTape. Subsequently, leave the context of GradientTape and compute the differential(s) $\frac{dJ}{dp}$ in order to evaluate the sensitivity. Algorithm 3 summarises all this for the one dimensional case. In appendix A the implementation is given. An algorithm for a two dimensional sensitivity analysis is identical, only the forward simulation as stated in algorithm 2 is used.

Algorithm 3: Pseudo Algorithm for a evaluation of sensitivity of cost function

```
Data: $\boldsymbol{\zeta}^0, \boldsymbol{u}^0, \zeta(0) = \alpha, u(L) = \beta, h_{interior}, u_{interior}, diff_f, diff_b$
Result: Evaluation of cost function $J(x,p)$, with
Reshape $\boldsymbol{\zeta}^0, \boldsymbol{u}^0, h_{interior}$ and $u_{interior}$ s.t. they are of shape $(1, N_x, 1)$;
Reshape $diff_f$ and $diff_b$ s.t. they are of shape $(3,1,1)$;
Initialize a tape, persistent tape if necessary;
Specify which parameters $p$ that need to be watched.
for i = 0,$\cdots$, $N_{steps} - 1$ do
    Compute $\boldsymbol{u}^{i+1}$ and $\boldsymbol{\zeta}^{i+1}$ as stated in algorithm 1
Define the cost function $J(x,p)$;
Compute the adjoints(s) $\frac{dJ(x,p)}{dp}$;
```

4.6. Implementation checkpointing

As stated in section 4.3 for fine meshes CPU/GPU memory might be insufficient, when performing sensitivity analysis. The implementation of the checkpointing method is similar to that of a single backward pass, but the difference lies in the definition of the cost function $J(x,p)$ in the backwards pass. Where in the single backward pass it was possible to differentiate the cost function $J(x,p)$ with respect to the initial parameter(s) directly, in the case of checkpointing the user has to redefine the cost function at the end of the simulation. First, evaluate u (,v) and ζ as proposed in algorithm 1 (2), while setting N_c checkpoints at timestep $t^k = k\Delta t\frac{N_{steps}}{N_c}$, where $k \in \{0,1,\dots,N_c-1\}$. After the forward simulation has run, release the last state of the variables taken at time t^{N_c-1} from the stack and define the cost function $J^{N_{steps}-1}(\boldsymbol{x}^{N_{steps}-1})$ at the end of the simulation, where $\boldsymbol{x}^{N_{steps}-1}$ is the tensor that contains the state of the variables $\boldsymbol{u}^{N_{steps}-1}$ (,$\boldsymbol{v}^{N_{steps}-1}$) and $\boldsymbol{\zeta}^{N_{steps}-1}$. For simplicity the cost function is defined independent of the input parameters $\boldsymbol{p}^{N_c-1}$, which is the tensor containing the state of the variables $\boldsymbol{u}^{N_c-1}$ (,$\boldsymbol{v}^{N_c-1}$) and $\boldsymbol{\zeta}^{N_c-1}$. Then compute the adjoint

$$\overline{\boldsymbol{p}^{N_c-1}} = \frac{dJ^{N_{steps}-1}}{d\boldsymbol{p}^{N_c-1}}. \tag{4.7}$$

Note, it is important to delete the created Wengert list from the memory, otherwise the out of memory will still occur.

Hereafter, for $k \in \{N_c-2,\dots,0\}$ again release $\boldsymbol{u}^k$ (,$\boldsymbol{v}^k$) and $\boldsymbol{\zeta}^k$ from the stack and initiate the GradientTape. Note, a new cost function $J^{k+1}(\boldsymbol{x}^{k+1})$ has to be defined at the end of the intermediate simulation t^{k+1}. Define the cost function as stated in equation 4.8.

$$J^{k+1}(\boldsymbol{x}^{k+1}) = \boldsymbol{x}^{k+1} \cdot \overline{\boldsymbol{p}^{k+1}}. \tag{4.8}$$

Next, outside of the context of the GradientTape compute the adjoint

$$\overline{\boldsymbol{p}^k} = \frac{dJ^{k+1}}{d\boldsymbol{p}^k}. \tag{4.9}$$

.
Again, delete the Wengert list from the GPU/CPU memory. Algorithm 4 encapsulates all the previous for the one dimensional case. For the one dimensional and two dimensional implementation look at Appendix A and B respectively.

Algorithm 4: Pseudo Algorithm for a evaluation of sensitivity of cost function through checkpointing

```
Data: $\boldsymbol{\zeta}^0, \boldsymbol{u^0}, \zeta(0) = \alpha, u(L) = \beta, \zeta_{interior}, u_{interior}, diff_f, diff_b$
Result: Evaluation of cost function $J(x,p)$
Reshape $\boldsymbol{\zeta}^0, \boldsymbol{u^0}, \zeta_{interior}$ and $u_{interior}$ s.t. they are of shape $(1, N_x, 1)$;
Reshape $diff_f$ and $diff_b$ s.t. they are of shape $(3,1,1)$;
Initialize a persistent tape;
Specify which parameters $\boldsymbol{p}$ that need to be watched;
for $n = 0, \ldots, N_{steps} - 1$ do
    Compute $\boldsymbol{u}^{n+1}$ and $\boldsymbol{\zeta}^{n+1}$ as stated in algorithm 1;
    if $n \ mod(N_c) = 0$ then
        save $\boldsymbol{u}^n$ and $\boldsymbol{\zeta}^n$ on a stack;
for $k \in (N_c - 1, \ldots, 0)$ do
    Restore $\boldsymbol{u}^k$ and $\boldsymbol{\zeta}^k$ from the stack;
    Initiate persistent tape and specify initial parameters $\boldsymbol{p}$ that need to be stored;
    for $n = k\frac{N_{steps}}{N_c}, \ldots, (k+1)\frac{N_{steps}}{N_c}$ do
        Compute $\boldsymbol{u}^{n+1}$ and $\boldsymbol{\zeta}^{n+1}$ as stated in algorithm 1;
    if $k \in (N_c - 2, \ldots, 0)$ then
        Define the cost function $J^{k+1}(\boldsymbol{x}^{k+1})$ as $\boldsymbol{x}^{k+1} \cdot \overline{\boldsymbol{p}^{k+1}}$;
        Compute the adjoints $\overline{\boldsymbol{p}^k}$;
        Discard the Wengert list;
    else
        Define the cost function $J^{N_{steps}-1}(\boldsymbol{x}^{N_{steps}-1})$;
        Compute the adjoint $\overline{\boldsymbol{p}^{N_c-1}}$;
        Discard the Wengert list;
```

Forward evaluation trace			Reverse adjoint trace		
v_{-2}	$=$	Z	$\overline{v}_{-2}$	$=$	$\overline{v}_1 \frac{\partial v_1}{\partial v_{-2}}$
v_{-1}	$=$	Y	$\overline{v}_{-1}$	$=$	$\overline{v}_2 \frac{\partial v_2}{\partial v_{-1}}$
v_0	$=$	X	$\overline{v}_0$	$=$	$\overline{v}_2 \frac{\partial v_2}{v_0}$
v_1	$=$	$exp(v_{-2})$	$\overline{v}_1$	$=$	$\overline{v}_3 \frac{\partial v_3}{\partial v_1}$
v_2	$=$	$v_{-1} + v_0$	$\overline{v}_2$	$=$	$\overline{v}_3 \frac{\partial v_3}{\partial v_2}$
v_3	$=$	v_1 / v_2	$\overline{v}_3$	$=$	1

Table 4.2: Adjoint AD, where $\boldsymbol{\zeta}^{n+1}$ is evaluated. After running the forward sequence, which is presented on the left, the adjoint operations are run in order to determine the sensitivity with respect to input variables in a reverse pass, starting at $\overline{v}_7$.

5 Numerical Results

The numerical results are produced in this chapter. The results of the forward simulation through the use of algorithms 1 and 2 of a gaussian disturbance in the one- and two-dimensional basins as staded in section 2.3. Furthermore, an example will be given on sensitivity analysis and a small remark on checkpointing is made. Ultimately, a benchmark is produced in order to check the efficiency of the forward solver.

5.1. Numerical solutions

The solution to the IVPs stated in section 2.3 will be presented with the use of the algorithms 1 and 2.

5.1.1. One dimensional case

With the help of the PDE solver, see appendix A for the implementation, a numerical solution is produced to the initial value problem described in section 2.3.1 over a time span of $T = 3000s$ and a basin of length $L = 100km$. Take as time step $\Delta t = 6.00s$ and the number of grid points $N_x = 1000$, resulting the spatial step $\Delta x = 99.95m$ and as a consequence the CFL-criterion 2.15 is satisfied. In order to check the accuracy of the numerical results, a comparision will be drawn to the analytical solutions for the water level in equation 2.20 and fluid velocity in equation 2.21. In figures 5.1 and 5.2 the initial conditions are displayed, where figure 5.1 shows the initial gaussian disturbance in the water level and it is visible in figure 5.2 that the initial fluid velocity equals zero in the entire domain.

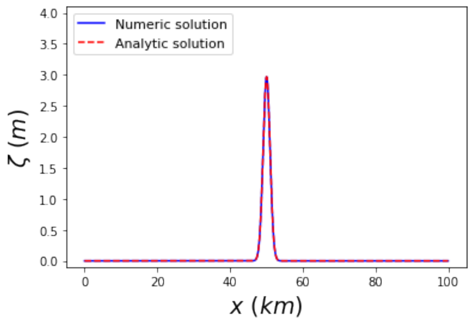

Figure 5.1: Initial Water level

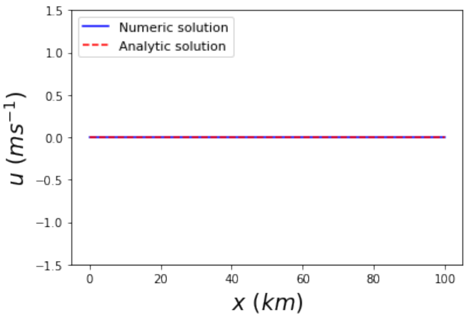

Figure 5.2: Initial fluid velocity

The numerical solutions, as well as their analytical counterpart are shown both halfway and at the end of the simulation. Figures 5.3 and 5.4 give the water level and fluid velocity after 25 minutes and the water level and fluid velocity after 50 minutes are shown in figures 5.5 and 5.6.

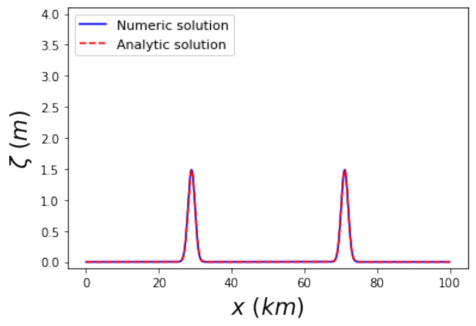

Figure 5.3: Water level after 25 minutes

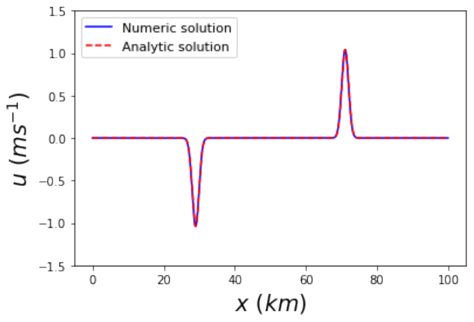

Figure 5.4: Fluid velocity after 25 minutes

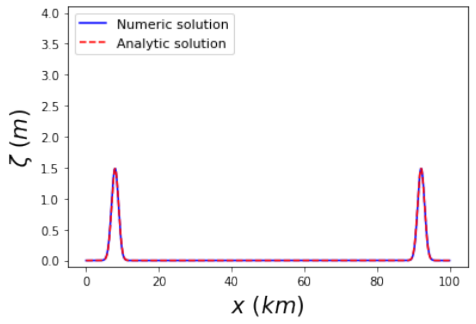

Figure 5.5: Water level after 50 minutes

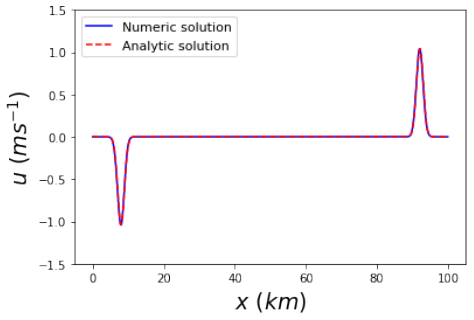

Figure 5.6: Fluid velocity after 50 minutes

In figures 5.3, 5.4, 5.5, 5.6 it can been seen that from the initial gaussian disturbance portrayed in figure 5.1 two waves propagate away from the initial position in the basin. At first glance, the numerical solution seems to propagate evenly with the analytical solution, but if a closer look is taken in figures 5.7 and 5.8, it appears that the numerical solution seems to propagates slower than the analytical solution. This can be attributed to phenomenon of numerical dissipation [16].

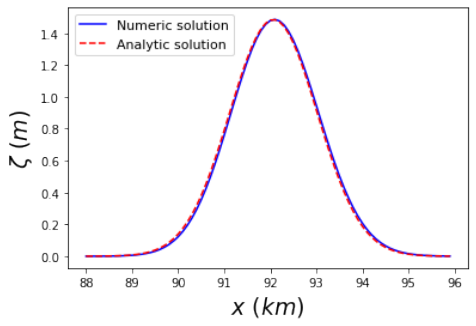

Figure 5.7: Right travelling wave from figure 5.5 enlarged

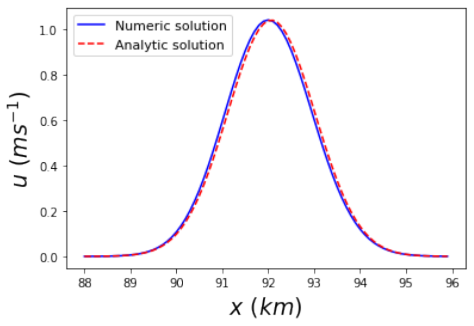

Figure 5.8: Fluid velocity right travelling wave from figure 5.6 enlarged

5.1.2. Two dimensional case

The numerical solution to the IVP in section 2.3.2 is calculated by the PDE solver for the two dimensional case, see Appendix B for the implementation, over a time span of $T = 3000s$ and a basin of length $L = 10.00km$ and width $W = 10.00km$. The domain is divided into $1.e4$ grid cells, so $N_x = N_y = 100$, resulting in a spatial step of $\Delta x = \Delta y = 100.00m$ and the time step is set to $\Delta t = 0.60s$. As a consequence, the CFL-criterion as stated in equation 2.16 is met. The friction coefficient due to bottom roughness c_f equals $1.93e.{-4}$. First both the initial condition and end of simulation of the water level are represented. After that, the numerical solutions to the zonal- and meridional fluid velocity are produced.

The initial water level and the water level after 5 and 50 minutes are displayed in figures 5.9, 5.10 and 5.11.

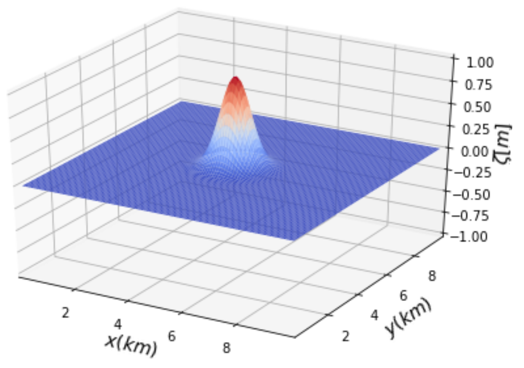

Figure 5.9: Initial Water level

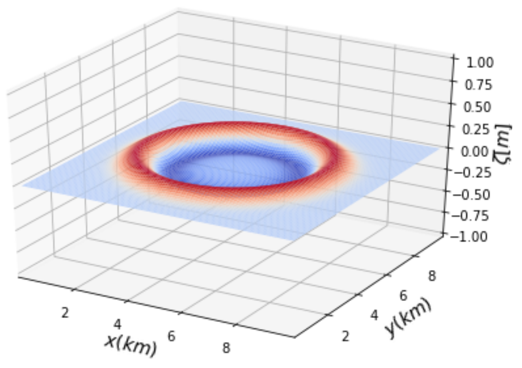

Figure 5.10: Water level after 5 minutes

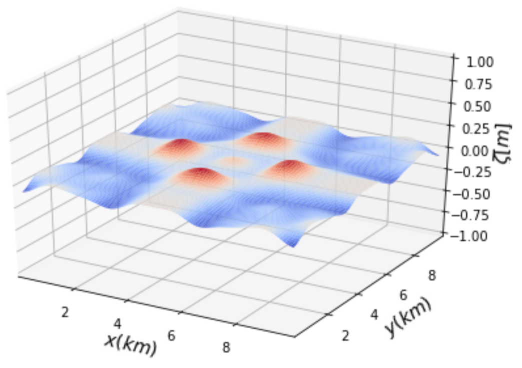

Figure 5.11: Water level after 50 minutes

Looking at figures 5.9, 5.10 and 5.11, two things stand out. First, the droplet propagates evenly across the surface. Second, due to the friction term present in the zonal- and meridional momentum equations 2.1b and 2.1c, the droplet loses height over time.

Advancing to the fluid velocity, a quiver plot is produced in order to asses the propagation speed of the droplet in the basin. Two quiver plots are produced, figure 5.12 gives the quiver plot after five minutes and figure 5.13 after fifty minutes.

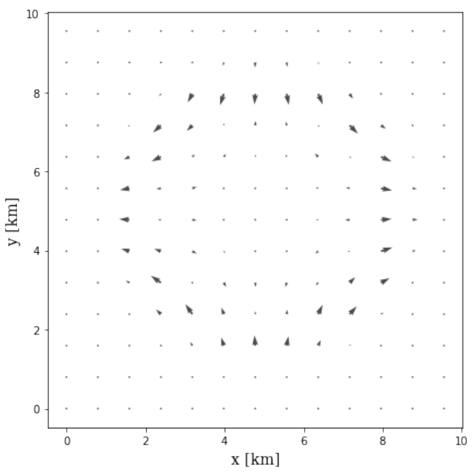

Figure 5.12: Quiver plot after 5 minutes

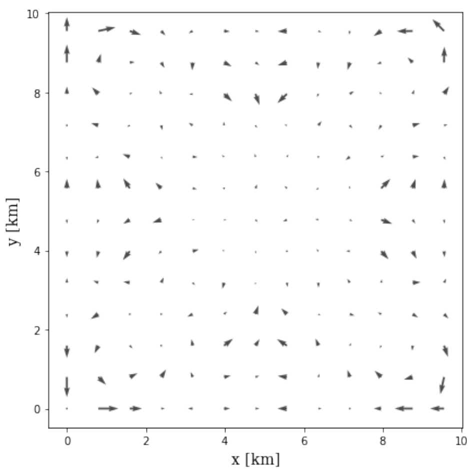

Figure 5.13: Quiver plot after 50 minutes

Figure 5.12 shows that the droplet is propagating in two ways. Towards the inside and towards the outside. Where the speed towards the outside is greater than towards the outside.

5.2. Sensitivity Analysis

As stated in section 2.5, the general solution to a wave-like equation is the sum of a right- and left travelling wave. As such, a perfect way to give an example of sensitivity analysis is a reverse experiment of the initial value problem in section 2.3.1. Instead of looking how a gaussian disturbance propagates in a basin over time, it is possible to look how a gaussian disturbance at the end of a simulation is the sum of two waves at an earlier stage. If the same gaussian disturbance is used, as in equation 2.5a, the sensitivity should be a reflection of the forward simulation, as portrayed in figures 5.5 and 5.6.

The same Boundary conditions as in equation 2.6 are used, only the initial conditions are set equal to zero. The time of the simulation is 50 minutes, with $\Delta t = 0.06s$ and $\Delta x = 99.95m$. Define the cost function, J, as the dot product of the final water level h^{N+1} and the droplet as stated in the initial conditions in equation 2.5a, so

$$J = \boldsymbol{\zeta}^{N-1} \cdot f(x).$$

Then we want to compute the adjoint with respect to both the initial water level and the initial fluid velocity, so

$$\frac{\partial J}{\partial \boldsymbol{\zeta}^0}, \frac{\partial J}{\partial \boldsymbol{u}^0}.$$

The sensitivity with respect to the initial water level and the initial are presented in respectively in figure 5.14 and 5.15. For the implementation, see appendix A.

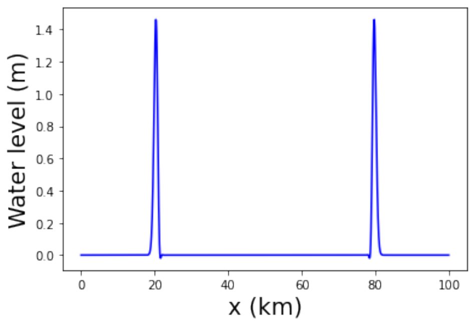

Figure 5.14: Sensitivity with respect to initial water level

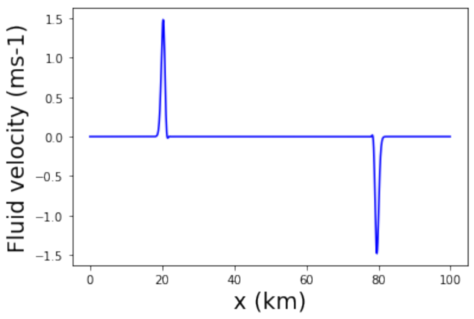

Figure 5.15: Sensitivity with respect to initial fluid velocity

In figure 5.15 it can be observed, that the left swell in figure 5.14 has a positive velocity, which means it's travelling towards to center. The right swell in figure 5.15 on the other hand has a negative velocity, which means it moves in opposite direction and thus towards to center. It was stated in section 2.5 that the solution to wave like equation exists of a left and right travelling wave. Since the swells are moving towards each other, this is in agreement with d'Alembert's solution.

Note, in figures 5.14 and 5.15, it can be noticed that there is a small spike at the bottom of each tail at the center and that the adjoint has some noise between the spikes. As stated in section 4.2, AD is able to produce exact numerical derivatives, which results in this noise.

5.2.1. Checkpointing

Equation 4.5 gave an estimate of how much memory was required to store an evaluation trace. The simulation is run on the NVIDIA Quadro M1200 GPU, of which 3030 MB of memory is allocated within the context *tf.GradientTape*. The amount of input variables is two, these are both of shape $(1, N_x, 1)$. Hence the amount of memory allocated two these two tensors is $Mtensor_i = 1 * N_x * 1 * 4 = 4N_x$.e-6 MB. The amount of working variables produced for a part of a timestep can be observed from figure 4.1. Note, in the computational graph dt, dx and H are also input variables, this is no longer case in our implementation. As such, the working variable v_4 and v_5 are no longer part of the computational graph for this example. Looking at figure 4.1 it can be seen that 5 working variables are created in order to determine $\boldsymbol{\zeta}^1$. These working variables all store a tensor of shape $(1, N_x, 1)$. So, the amount of memory allocated at each working variable is again $Mtensor_i = 1 * N_x * 1 * 4 = 4N_x$.e-6 MB. Note, in order to determine $\boldsymbol{u}^1$ similar reasoning applies to see that 5 working variables are created, where at each working variable a tensor of memory size $Mtensor_i = 4N_x$.e-6 MB is stored. If one wants to perform sensitivity in the same basin and over an equal time span, only with $N_x = 1.e5$ grid points, the spatial step will satisfy $\Delta x = 1.00m$ and the time step must be set to $\Delta t = 0.06s$ in order to meet the CFL-criterion 2.15. Combing the previous, according to the equation 4.5 the total amount of memory needed is $1.e6$ MB.

Since the memory required exceeds the amount of memory available on the GPU, an out of memory error will be produced. With the help of equation 4.6, we can determine a minimum number of checkpoints in order to run for a grid with $N_x = 1.e5$ grid points. Solving equation 4.6 for N_c, it is found that a minimum of $N_c = 68$ checkpoints must be set.

Having said that, TensorFlow will still produce an out of memory error. After performing the simulation for a variety of integers, it was found that a minimum of $N_c = 89$ checkpoints must be set to prevent the out of memory error. When $N_c \in \{68, \ldots, 88\}$ the out of memory error no longer occurs in the forward pass where the elementary operations are stored in memory, but in the calculation of the adjoint.

5.3. Computational time

Since our PDE solver is able to use GPU in parallel in order to accelerate computations, there is possibility that it is faster than implementations of PDE solvers in common programms that run sequentially on the CPU. A dominant programming language in the field of large scale numerical simulation, is a "modern version" of Fortran (90/95/03/08). Fortran is well suited to run numerical simulations fast on a CPU, and is able to run processes in parallel. However, as mentioned before parallelizing requires more manual labour.

A benchmark is produced, where the PDE solver for the two dimensional IVP implemented in TensorFlow is compared for both the CPU and GPU in parallel, to a Fortran CPU scalar implementation written by Martin Verlaan in Fortran90, see appendix D, which runs on a single CPU core. In addition, the benchmark also includes an non-vectorized loop over grid implementation of a PDE solver in NumPy, see appendix C. The goal of this benchmark is to give a reference on how fast or slow TensorFlow is compared to other programming languages. As compiler, Intel(R) Fortran Compiler is used, which is part of the software development product Intel(R) Parallel Studio XE. The benchmark is run on the 4 core CPU Intel(R) Core(TM) i7-7700HQ CPU clocked at 2.80GHz and on the NVIDIA Quadro M1200 GPU. A basin of Length $L = 10km$ and Width $W = 10km$ is considered for the simulation, which will be run over time span of 50 minutes, with $\Delta t = 0.06s$ to ensure that the CFL-criterion, as stated in equation 2.16, is matched at all times. The number of grid cells are powers of 10, and the largest number of grid cells equals 1.e7. Note, for the TensorFlow-CPU and NumPy, the largest number amount of grid cells are 1.e7 and 5.e3 respectively. The Benchmark is depicted in figure 5.16.

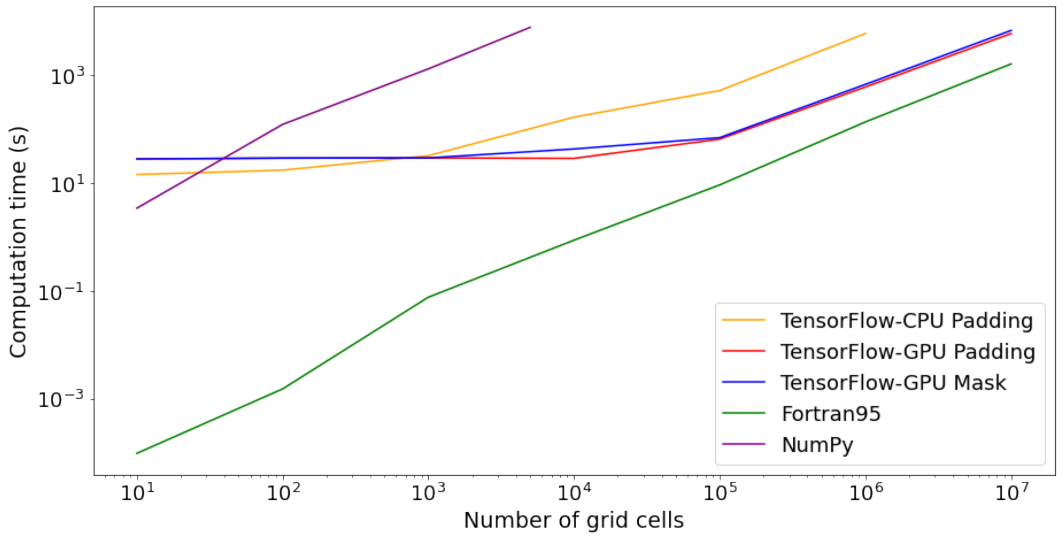

Figure 5.16: Benchmark for 50000 timesteps

Looking at the results of the benchmark in figure 5.16 a few things stand out. First, where the Fortran implementation has a very small startup time, the CPU implementation takes about 15 seconds and both GPU implementations take roughly 30 seconds to perform a computation, but then remains constant for the CPU when the number of grid cells is smaller than 100, and the number of grid cells is smaller than 1.e4 for the GPU. This can be attributed to the overhead of TensorFlow. When the mesh is fine the implementation makes full use of the processing power of both the CPU and GPU.

Second, it's clearly visible that the TensorFlow implementation that runs on the CPU is not an efficient process, on one hand because of the significantly longer duration relative to Fortran, independent of the number of grid cells. On the other hand, because of the significantly longer duration for a large number of grid cells, $(N_x \times N_y) \geq 1$.e4, in comparision with both TensorFlow-GPU implementations. As mentioned in section 3.2, the convolution operation is a small operation, which can be performed in parallel, as such for large tensors performing convolution on the GPU is more efficient as compared to the CPU.

Third, figure 5.16 shows that implementing boundary conditions with padding requires less computation time, as compared to masks. This comes to no surprise, since four additional computations need to be performed each iteration, of which two are large scale matrix multiplication.

Finally, the numerical simulation in Fortran is constantly quicker, independent of the amount of grid cells. In addition, the NumPy implementation is dreadfully slow. Note, the NumPy implementation loops through a grid, which generally requires an significant amount of computation time. For a fine mesh, $(N_x \times N_y) \geq 1$.e5, the Fortran implementation requires roughly five times less time to finish a simulation relative to both TensorFlow-GPU implementations. It is important to keep in mind that all these implementations can be optimized further. Careful optimization in Fortran with MPI and/or GPU is likely to give the fastest code, but this will come at the larger of effort to code these optimizations and implement the adjoint computations. The same can be said for the Numpy implementation, where the loop over grid implementation can be replaced by a function from for example the SciPy library.

6 Conclusion

A main goal of this work was to answer the question whether it was possible to obtain an efficient PDE solver implemented in TensorFlow. Efficient can be interpreted in two ways. On one hand, can one a implement an PDE solver with a low level of complexity, but on the other hand is the PDE solver computationally efficient. As shown in chapter 3 with relative ease one can implement a PDE solver in TensorFlow. There were only two key problems that needed to be resolved. The first issue was how the spatial derivatives could be computed with the finite difference method. This could easily be done through convolution. The second problem concerned how the boundary conditions could be implemented. Two methods were presented, masks and padding. Regarding the accuracy of the PDE solver, in figures 5.5 and 5.6 it was shown that the PDE solver was able to produce numerical results that closely approximated their analytical counterpart.

However, regarding the second perspective it can be concluded that although TensorFlow is able to perform computations in parallel through to use of the CUDA of a NVIDIA GPU, other software programs, like the Intel Fortran Compiler, are able to produce results faster while performing computational tasks sequentially on a single CPU core. The Fortran code can be adapted manually with e.g. OpenMP or MPI to run in parallel and reduce the computation time. This step will require manual effort. Nevertheless, the implementation in TensorFlow still requires less running time compared to a straight forward implementation in NumPy.

Another main goal of this paper was the use of the Automatic Differentiation API in TensorFlow in order to perform sensitivity analysis. Again in chapter 4 it was shown that with one can compute differentials of an cost function with respect to certain input parameters with only a few extension to the code in TensorFlow. Not only is it possible to perform sensitivity computations with very little manual labour, TensorFlow also automatically harnesses the power of the CUDA of a NVIDIA GPU. This can be seen as a great advantage over software programs like Fortran or NumPy, where the creation of adjoint code that is able to perform parallel computations on the GPU requires manual effort. There do exist compilers that produce adjoint code such as TAPENADE.

All in all, TensorFlow can be considered as an useful tool in the field of CFD, because of the relative ease of programming and the availability of an API for Automatic Differentiation. However, it is a bit cumbersome to rewrite everything in terms of TensorFlow. Since only a relatively simple test-case was considered, in further research it must be tested whether TensorFlow can be used in complex studies concerning sensitivity analysis or data assimilation.

7 Discussion

Similar to the example given by Daoust et al., in this work only a PDE solver for linear partial differential equations was derived, whereas the shallow water equations 2.1 and 2.3 are mostly made up of nonlinear terms. It might be interesting to implement a solver for non-linear partial differential equations. Contrary to Daoust et al. this PDE solver was based upon a staggered grid. Where Daoust et al. didn't gave a clear explanation on how to implement boundary conditions, this paper has clarified this issue to a greater extent. Nonetheless, only Dirichlet boundary conditions were considered. In the future, a method to implement Neumann and Robin boundary conditions can be derived. Moreover, Daoust et al. had defined functions to perform convolution that reshaped the tensor in every iteration. In order to improve the computational efficiency, the tensors were only reshaped at the start and at the end of the simulation.

In this work only rectangular domains were treated. For further research it is advised to explore the possibility to implement complex domains, such as rivers, estuaries or coastal domains. It is expected that this can be performed with the help of a mask. One can define an interior mask of where the indices are zeros on the indices where the domain is not defined. Furthermore, the PDE solver was based upon a semi-implicit scheme, further research is needed to see whether implicit schemes are also possible.

Another main part of this research was dedicated to the application of Automatic Differentiation. Only a theoretical simulation was performed to show to ability to perform sensitivity analysis. Further research can be done for more physically relevant sensitivity analysis, such as friction. In addition it could be useful to use the API GrandientTape for a real data-assimilation case study. No serious issues are foreseen, it is advised to define the cost function as mentioned in the 4D-Var data assimilation system by EMWCF. In addition, AD also finds a wide range of application across different field of studies as mentioned in the introduction. Again, it might be useful to explore the use of TensorFlow in these fields.

Also, the checkpointing strategy outlined in this paper is a pretty basic one. In further research, it is recommended to implement a strategy, where an optimal trade off is made between memory allocation and computation time, such as the *Revolve* algorithm by Griewank and Walther [13]. In addition, it might be interesting to use for instance Hybrid Checkpointing or writing all state variables to Disk in order to run adjoint simulations for grids that contain more than $1.e5$ grid points or grid cells.

Finally, next to TensorFlow, other differentiable programming frameworks, like Theano or Pytorch, could be used in theory for the same purpose. In further research a comparison can be made between these differentiable prorgramming frameworks in the ease of use and computational efficiency of an implementation of a PDE solver and/or adjoint code.

A

Implementation 1D

```python
def Gaus_Dist(N_x):
    gaus_x = np.linspace(-1, 1,int(N_x/10)+1, endpoint=True, dtype=np.float32)
    a = 0.2
    gaus_smooth = 1/(a*np.sqrt(np.pi)) * np.exp(-(1/2)*(gaus_x*gaus_x)/(a**2)) # Density function of Normal distribution with mean zero and variance a
    gaus_dist = np.zeros(N_x,dtype=np.float32)
    for i in range(int((N_x/10)+1)):
        gaus_dist[int((N_x/2) - (N_x/20))+i]= gaus_smooth[i]
    t_gaus_dist = tf.reshape(tf.Variable(gaus_dist),(1,N_x,1))
    return t_gaus_dist

def Initial_Tensors(N_x):
    h_0 = tf.Variable(tf.zeros([1,N_x,1],dtype=tf.float32))
    u_0 = tf.Variable(tf.zeros([1,N_x,1],dtype=tf.float32))
    return h_0,u_0

def Boundary_Masks(N_x,r_val,l_val):
    int_r = np.ones(N_x,dtype=np.float32); int_r[-1]=0;int_r = tf.reshape(tf.constant(int_r),(1,N_x,1))        #Interior Mask left
    bnd_r = np.zeros(N_x,dtype=np.float32); bnd_r[-1]= r_val;bnd_r = tf.reshape(tf.constant(bnd_r),(1,N_x,1)) #Boundary Mask left
    int_l = np.ones(N_x,dtype=np.float32); int_l[0]=0; int_l = tf.reshape(tf.constant(int_l),(1,N_x,1))        #Interior Mask right
    bnd_l = np.zeros(N_x,dtype=np.float32); bnd_l[0]= l_val; bnd_l = tf.reshape(tf.constant(bnd_l),(1,N_x,1)) #Boundary Mask right
    return int_r,bnd_r,int_l,bnd_l

def Convolution_Filters():
    diff_b = tf.reshape(tf.constant([-1.,1.,0.]),[3,1,1])              #Backward Difference Filter
    diff_f = tf.reshape(tf.constant([0.,-1.,1.]),[3,1,1])              #Forward Difference Filter
    return diff_b,diff_f

def PDE_Solver(h_0,u_0,T,dt,N_x,L,H,l_val,r_val):
    dx = tf.constant(L/(N_x+0.5))
    T = 60.*T
    timesteps = tf.math.round((T/dt))
    g = 9.81
    h = h_0; u = u_0
    int_r,bnd_r,int_l,bnd_l = Boundary_Masks(N_x,r_val,l_val)
    diff_b,diff_f = Convolution_Filters()
    for i in range(int(timesteps)):
        h = h*int_l + bnd_l                                                        # Setting Boundary value
        dhdx = tf.nn.conv1d(h, diff_f, stride=1,padding =”SAME”)                   # Spatial derivative of h
        u = (1.-dt*f)*u - (g*dt)/(dx) * dhdx                                       # Next iteration of u
        u = u*int_r + bnd_r                                              # Setting Boundary value
        dudx = tf.nn.conv1d(u, diff_b, stride=1,padding =”SAME”)         # Spatial derivative u
        h = h - (H*dt/dx) * dudx                                         # Next iteration of h
    return h,u

def Checkpoints(timesteps,N_c):
    check_h = {}
    check_u = {}
    check_list = []
    interval_r = int(timesteps/N_c)
    if timesteps % N_c == 0:
        interval_ir = interval_r
    else:
        interval_ir = timesteps - N_c*interval_r
    for i in range(0,timesteps,interval_r):
        check_list.append(i)
        check_u[’u_’+str(i)]= 0
        check_h[’h_’+str(i)]= 0
    return interval_r, interval_ir,check_h,check_u,check_list

def Checkpoint_Simulation_Forward(h_0,u_0,T,dt,N_x,L,H,l_val,r_val,N_c):
    dx = tf.constant(L/(N_x+0.5))
    T = 60.*T
    timesteps = int(tf.math.round((T/dt)))
    g = 9.81
    h = h_0; u = u_0
    int_r,bnd_r,int_l,bnd_l = Boundary_Masks(N_x,r_val,l_val)
    diff_b,diff_f = Convolution_Filters()
    interval_r, interval_ir,check_h,check_u,check_list = Checkpoints(timesteps,N_c)
    for i in range(timesteps):
        if i in check_list:
            check_h[’h_’+str(i)] = h
            check_u[’u_’+str(i)] = u
        h = h*int_l + bnd_l                                              # Setting Boundary value
        u = (1.-dt*0)*u - (g*dt)/(dx) * tf.nn.conv1d(h, diff_f, stride=1,padding =”SAME”) # Next iteration of u
        u = u*int_r + bnd_r                                              # Setting Boundary value
        h = h - (H*dt/dx) * tf.nn.conv1d(u, diff_b, stride=1,padding =”SAME”)          # Next iteration of h
    return h, u,check_u, check_h,interval_r,interval_ir,check_list,int_r,bnd_r,int_l,bnd_l,
    diff_b,diff_f

def Checkpoint_Simulation_init(h_k,u_k,dt,dx,g,H,f,interval_ir,int_r,bnd_r,int_l,bnd_l,diff_b
    ,diff_f):
    t_gaus_dist = Gaus_Dist(N_x)
    with tf.GradientTape(persistent=True) as tape:
        h=h_k
        u=u_k
        tape.watch(h_k)
        tape.watch(u_k)
        for i in range(interval_ir):
            h = h*int_l + bnd_l #Setting Boundary value
            dh_dx = tf.nn.conv1d(h, diff_f, stride=1,padding =”SAME”)
            u = (1.-dt*f)*u - (g*dt)/(dx) * dh_dx#tf.nn.conv1d(h, diff_f, stride=1,padding =”SAME”) #Next iteration of u
            u = u*int_r + bnd_r #Setting Boundary value
            du_dx = tf.nn.conv1d(u, diff_b, stride=1,padding =”SAME”)
            h = h - (H*dt/dx) * du_dx #tf.nn.conv1d(u, diff_b, stride=1,padding =”SAME”) #Next iteration of h
        cost = tf.tensordot(t_gaus_dist,h,axes=3)
    dJ_dh = tape.gradient(cost,h_k)
    dJ_du = tape.gradient(cost,u_k)
    del tape
    return dJ_dh,dJ_du
```

```

def Checkpoint_Simulation(h_k,u_k,dt,dx,g,H,f,interval_r,dJ_dh,dJ_du,int_r,bnd_r,int_l,bnd_l,
    diff_b,diff_f):
    with tf.GradientTape(persistent=True) as tape:
        h=h_k                                                                                  u=u_k
        tape.watch(h_k)
        tape.watch(u_k)
        for i in range(interval_r):
            h = h*int_l + bnd_l
     #Setting Boundary value
            u = (1.-dt*f)*u - (g*dt)/(dx) * tf.nn.conv1d(h, diff_f, stride=1,padding ="SAME")
     #Next iteration of u
            u = u*int_r + bnd_r
     #Setting Boundary value
            h = h - (H*dt/dx) * tf.nn.conv1d(u, diff_b, stride=1,padding ="SAME")
     #Next iteration of h
        cost = tf.tensordot(dJ_dh,h,axes = 3) + tf.tensordot(dJ_du,u,axes=3)
    dJ_dh = tape.gradient(cost,h_k)
    dJ_du = tape.gradient(cost,u_k)
    del tape
    return dJ_dh,dJ_du

def Sensitivity_Analysis_Checkpoint(h_0,u_0,dt,T,L,H,N_x,l_val,r_val,N_c):
    h_k,u_k,check_u,check_h,interval_r,interval_ir,check_list,int_r,bnd_r,int_l,bnd_l,diff_b,
    diff_f = Checkpoint_Simulation_Forward(h_0,u_0,T,dt,N_x,L,H,l_val,r_val,N_c)
    k = check_list[-1]
    h_k = check_h['h_'+str(k)]; u_k = check_u['u_'+str(k)]
    dJ_dh,dJ_du = Checkpoint_Simulation_init(h_k,u_k,dt,dx,g,H,f,interval_ir,int_r,bnd_r,
    int_l,bnd_l,diff_b,diff_f)
    for k in check_list[-2::-1]:
        dJ_dh,dJ_du = Checkpoint_Simulation(check_h['h_'+str(k)],check_u['u_'+str(k)],dt,dx,g
    ,H,f,interval_r,dJ_dh,dJ_du,int_r,bnd_r,int_l,bnd_l,diff_b,diff_f)
    return dJ_dh,dJ_du
```

B
Implementation 2D

```python
#Solver Functions
def Initial_Tensors(N_x,N_y):
    h_0 = tf.Variable(tf.zeros([1,N_x,N_y,1], dtype= tf.float32))
    u_0 = tf.Variable(tf.zeros([1,N_x,N_y+1,1], dtype= tf.float32))
    v_0 = tf.Variable(tf.zeros([1,N_x+1,N_y,1], dtype= tf.float32))
    return h_0, u_0, v_0

def Convolution_Filters():
    diff_x = tf.reshape(tf.constant(np.asarray([[-1.,1.]]),dtype=1),[1,2]+[1,1])
    diff_y = tf.reshape(tf.constant(np.asarray([[1.],[-1.]]),dtype=1),[2,1]+[1,1])
    return diff_x, diff_y

def Pads():
    h_x_pad = tf.constant([[0,0],[0,0],[1,1],[0,0]])
    h_y_pad = tf.constant([[0,0],[1,1],[0,0],[0,0]])
    return h_x_pad, h_y_pad

def Masks(N_x,N_y,alpha,beta,eta,zeta):
    int_h =  np.zeros((N_x,N_y),dtype=np.float32);int_h[1:-1,] = 1;int_h = tf.reshape(tf.
    constant(int_h),(1,N_x,N_y,1))
    int_u =  np.zeros((N_x,N_y+1),dtype=np.float32);int_u[:,1:-1] = 1;int_u = tf.reshape(tf.
    constant(int_u),(1,N_x,N_y+1,1))
    int_v =  np.zeros((N_x+1,N_y),dtype=np.float32);int_v[1:-1,] = 1;int_v = tf.reshape(tf.
    constant(int_v),(1,N_x+1,N_y,1))
    bnd_h =  np.zeros((N_x,N_y),dtype=np.float32);bnd_h[0,] = 1;bnd_h[-1,] = 1;bnd_h = tf.
    reshape(tf.constant(bnd_h),(1,N_x,N_y,1))
    bnd_u =  np.zeros((N_x,N_y+1),dtype=np.float32);bnd_u[:,0] = alpha;bnd_u[:,-1] = beta;
    bnd_u = tf.reshape(tf.constant(bnd_u),(1,N_x,N_y+1,1))
    bnd_v =  np.zeros((N_x+1,N_y),dtype=np.float32);bnd_v[0,] = eta;bnd_v[-1,] = zeta;bnd_v =
     tf.reshape(tf.constant(bnd_v),(1,N_y+1,N_y,1))
    return int_h, int_u, int_v, bnd_h, bnd_u, bnd_v

def PDE_Solver_Padding(h_0,u_0,v_0,N_x,N_y,T,dt,H,W):
    h = h_0; u = u_0; v = v_0
    dx = tf.constant(L/(N_x), dtype = tf.float32) ; dy = tf.constant(W/(N_y), dtype = tf.
    float32)
    diff_x, diff_y = Convolution_Filters()
    h_x_pad, h_y_pad = Pads()
    T = 60. * T
    timesteps = int(tf.math.round(T/dt))
    g = tf.constant(9.81,dtype=tf.float32)
    f = tf.constant(1/(0.06*24*3600),dtype=tf.float32)
    for i in range(timesteps):
        # Calculating u at next timestep
        dh_dx = tf.nn.conv2d(h, diff_x, strides = 1, padding = "VALID")   #computes spatial
    derivative h w.r.t. x
        dh_dx = tf.pad(dh_dx, h_x_pad, mode = "CONSTANT",constant_values=0) #pad spatial
    derivative h
        u = u - (g*dt/dx) * dh_dx
        # Calculating v at next time step
        dh_dy = tf.nn.conv2d(h, diff_y, strides = 1, padding = "VALID")   #computes spatial derivative h w.r.t. y
        dh_dy = tf.pad(dh_dy, h_y_pad, mode = "CONSTANT",constant_values=0) #pad spatial derivative h
        v = v - (g*dt/dy) * dh_dy
        # Calculating h at next time step
        du_dx = tf.nn.conv2d(u, diff_x, strides = 1, padding = "VALID")     #computes spatial derivative u w.r.t. x
        dv_dy = tf.nn.conv2d(v, diff_y, strides = 1, padding = "VALID")     #computes spatial derivative v w.r.t. y
        h = h - H*dt*(du_dx/dx + dv_dy/dy)
    return h,u,v

def PDE_Solver_Masks(h_0,u_0,v_0,N_x,N_y,T,dt,H,W, alpha, beta, eta, zeta):
    h = h_0; u = u_0; v = v_0
    dx = tf.constant(L/(N_x), dtype = tf.float32) ; dy = tf.constant(W/(N_y), dtype = tf.float32)
    diff_x, diff_y = Convolution_Filters()
    h_x_pad, h_y_pad = Pads()
    int_h, int_u, int_v, bnd_h, bnd_u, bnd_v = Masks(N_x,N_y, alpha, beta, eta, zeta)
    T = 60. * T
    timesteps = int(tf.math.round(T/dt))
    g = tf.constant(9.81,dtype=tf.float32)
    f = tf.constant(1/(0.06*24*3600),dtype=tf.float32)
    for i in range(timesteps):
        # Calculating u at next timestep
        dh_dx = tf.nn.conv2d(h, diff_x, strides = 1, padding = "VALID")   #computes spatial derivative h w.r.t. x
        dh_dx = tf.pad(dh_dx, h_x_pad, mode = "CONSTANT",constant_values=0) #pad spatial derivative h
        u = int_u * (u - (g*dt/dx) * dh_dx)
        # Calculating v at next time step
        dh_dy = tf.nn.conv2d(h, diff_y, strides = 1, padding = "VALID")   #computes spatial derivative h w.r.t. y
        dh_dy = tf.pad(dh_dy, h_y_pad, mode = "CONSTANT",constant_values=0) #pad spatial derivative h
        v = v - (g*dt/dy) * dh_dy
        # Calculating h at next time step
        du_dx = tf.nn.conv2d(u, diff_x, strides = 1, padding = "VALID")     #computes spatial derivative u w.r.t. x
        dv_dy = tf.nn.conv2d(v, diff_y, strides = 1, padding = "VALID")     #computes spatial derivative v w.r.t. y
        h = h - H*dt*(du_dx/dx + dv_dy/dy)
    return h,u,v

def Checkpoints(timesteps,N_c):
    check_h = {}
    check_u = {}
    check_v = {}
    check_list = []
    interval_r = int(timesteps/N_c)
    if timesteps % N_c == 0:
        interval_ir = interval_r
    else:
        interval_ir = timesteps - N_c*interval_r
    for i in range(0,timesteps,interval_r):
        check_list.append(i)
        check_v['v_'+str(i)]= 0
        check_u['u_'+str(i)]= 0
        check_h['h_'+str(i)]= 0
    return interval_r, interval_ir,check_h,check_u,check_v,check_list

def Checkpoint_Simulation_Forward(h_0,u_0,v_0,N_x,N_y,T,dt,H,W,N_c):
    h = h_0; u = u_0; v= v_0
    dx = tf.constant(L/(N_x-1), dtype = tf.float32) ; dy = tf.constant(W/(N_y-1), dtype = tf.float32)
    diff_x, diff_y = Convolution_Filters()
    h_x_pad, h_y_pad = Pads()
    T = 60. * T
    timesteps = int(tf.math.round(T/dt))
    g = tf.constant(9.81,dtype=tf.float32)
    interval_r, interval_ir,check_h,check_u,check_v,check_list = Checkpoints(timesteps,N_c)
    for i in range(timesteps):
        if i in check_list:
            check_h['h_'+str(i)] = h
            check_u['u_'+str(i)] = u
            check_v['v_'+str(i)] = v
        # Calculating u at next timestep
        dh_dx = tf.nn.conv2d(h, diff_x, strides = 1, padding = "VALID")   #computes spatial derivative h w.r.t. x
        dh_dx = tf.pad(dh_dx, h_x_pad, mode = "CONSTANT",constant_values=0) #pad spatial derivative h
        u = u - (g*dt/dx) * dh_dx
        # Calculating v at next time step
        dh_dy = tf.nn.conv2d(h, diff_y, strides = 1, padding = "VALID")   #computes spatial derivative h w.r.t. y
        dh_dy = tf.pad(dh_dy, h_y_pad, mode = "CONSTANT",constant_values=0) #pad spatial derivative h
        v = v - (g*dt/dy) * dh_dy
        # Calculating h at next time step
        du_dx = tf.nn.conv2d(u, diff_x, strides = 1, padding = "VALID")     #computes spatial derivative u w.r.t. x
        dv_dy = tf.nn.conv2d(v, diff_y, strides = 1, padding = "VALID")     #computes spatial derivative v w.r.t. y
        h = h - H*dt*(du_dx/dx + dv_dy/dy)
    return h,u,v,check_u, check_v,check_h,interval_r,interval_ir,check_list,diff_x, diff_y, h_x_pad, h_y_pad

def Checkpoint_Simulation_init(h_k,u_k,v_k,dt,dx,g,H,interval_ir,diff_x,diff_y,h_x_pad, h_y_pad):
    t_gaus_dist = tf.reshape(np.exp(-((x_grid_h-L/2)**2/(2*(0.05E+4)**2) + (y_grid_h-W/2)**2/(2*(0.05E+4)**2))),[1,N_x,N_y,1])
    with tf.GradientTape(persistent=True) as tape:
        h=h_k
        u=u_k
        v=v_k
        tape.watch(h_k)
        tape.watch(u_k)
        tape.watch(v_k)
        for i in range(interval_ir):
            dh_dx = tf.nn.conv2d(h, diff_x, strides = 1, padding = "VALID")   #computes spatial derivative h w.r.t. x
            dh_dx = tf.pad(dh_dx, h_x_pad, mode = "CONSTANT",constant_values=0) #pad spatial derivative h
            u = u - (g*dt/dx) * dh_dx
            # Calculating v at next time step
            dh_dy = tf.nn.conv2d(h, diff_y, strides = 1, padding = "VALID")   #computes spatial derivative h w.r.t. y
            dh_dy = tf.pad(dh_dy, h_y_pad, mode = "CONSTANT",constant_values=0) #pad spatial derivative h
            v = v - (g*dt/dy) * dh_dy
            # Calculating h at next time step
            du_dx = tf.nn.conv2d(u, diff_x, strides = 1, padding = "VALID")     #computes spatial derivative u w.r.t. x
            dv_dy = tf.nn.conv2d(v, diff_y, strides = 1, padding = "VALID")     #computes spatial derivative v w.r.t. y
            h = h - H*dt*(du_dx/dx + dv_dy/dy)
        cost = tf.tensordot(t_gaus_dist,h,axes=3)
    dJ_dh = tape.gradient(cost,h_k)
    dJ_du = tape.gradient(cost,u_k)
    dJ_dv = tape.gradient(cost,v_k)
    del tape
    return dJ_dh,dJ_du,dJ_dv

def Checkpoint_Simulation (h_k,u_k,v_k,dt,dx,g,H,interval_r,dJ_dh,dJ_du,dJ_dv,diff_b,diff_f, h_x_pad, h_y_pad):
    with tf.GradientTape(persistent=True) as tape:
        h=h_k
        u=u_k
        v=v_k
        tape.watch(h_k)
        tape.watch(u_k)
        tape.watch(v_k)
        for i in range(interval_ir):
            dh_dx = tf.nn.conv2d(h, diff_x, strides = 1, padding = "VALID")   #computes spatial derivative h w.r.t. x
            dh_dx = tf.pad(dh_dx, h_x_pad, mode = "CONSTANT",constant_values=0) #pad spatial derivative h
            u = u - (g*dt/dx) * dh_dx
            # Calculating v at next time step
            dh_dy = tf.nn.conv2d(h, diff_y, strides = 1, padding = "VALID")   #computes spatial derivative h w.r.t. y
            dh_dy = tf.pad(dh_dy, h_y_pad, mode = "CONSTANT",constant_values=0) #pad spatial derivative h
            v = v - (g*dt/dy) * dh_dy
            # Calculating h at next time step
            du_dx = tf.nn.conv2d(u, diff_x, strides = 1, padding = "VALID")     #computes spatial derivative u w.r.t. x
            dv_dy = tf.nn.conv2d(v, diff_y, strides = 1, padding = "VALID")     #computes spatial derivative v w.r.t. y
            h = h - H*dt*(du_dx/dx + dv_dy/dy)
        cost = tf.tensordot(dJ_dh,h,axes = 3) + tf.tensordot(dJ_du,u,axes=3) + tf.tensordot(dJ_dv,v,axes = 3)
    dJ_dh = tape.gradient(cost,h_k)
    dJ_du = tape.gradient(cost,u_k)
    dJ_dv = tape.gradient(cost,v_k)
    del tape
    return dJ_dh,dJ_du,dJ_dv

def Sensitivity_Analysis(h_0,u_0,dt,T,L,H,N_x,N_y,N_c):
    h,u,v,check_u, check_v,check_h,interval_r,interval_ir,check_list,diff_x, diff_y,h_x_pad, h_y_pad = Checkpoint_Simulation_Forward(h_0,u_0,v_0,N_x,N_y,T,dt,H,W,N_c)
    k = check_list[-1]
    h_k = check_h['h_'+str(k)]; u_k = check_u['u_'+str(k)]; v_k = check_v['v_'+str(k)]
    dJ_dh,dJ_du,dJ_dv = Checkpoint_Simulation_init(h_k,u_k,v_k,dt,dx,g,H,interval_ir,diff_x, diff_y,h_x_pad, h_y_pad)
    for k in check_list[-2::-1]:
        dJ_dh,dJ_du,dJ_dv = Checkpoint_Simulation (h_k,u_k,v_k,dt,dx,g,H,interval_r,dJ_dh, dJ_du,dJ_dv,diff_b,diff_f,h_x_pad, h_y_pad)
        print(k)
    return dJ_dh,dJ_du,dJ_dv
```

C Implementation NumPy

```python
import numpy as np
from timeit import default_timer as timer

minutes_to_seconds=60.0
hours_to_seconds=60.0*minutes_to_seconds
days_to_seconds=24.0*hours_to_seconds

#constants
g = 9.81                                          #gravity [m s-2]
f=1.0/(0.06*days_to_seconds) #damping time-scale [s-1]
d=10.0                                            #depth below z=0 reference [m]

#geometry
n_max=101;m_max= 101 #number of nodes i.e. one more than the number of cells
x_length=10000.0; y_length=10000.0
dx=x_length/(m_max-1.0)
dy=y_length/(n_max-1.0)

#Time
t_stop=50.0*minutes_to_seconds
dt=0.001*minutes_to_seconds

#Grid
u = np.zeros((n_max,m_max),dtype=np.float32)
v = np.zeros((n_max,m_max),dtype=np.float32)
h = np.zeros((n_max,m_max),dtype=np.float32)

#Initial Conditions
h[int(n_max/2),int(m_max/2)] = 1.0

print("CFL =", np.sqrt(g*d)*dt/dx + np.sqrt(g*d)*dt/dy)

#Forward Simulation
t= 0
istep = 0

while t < t_stop:
    for i in range(0,n_max-1):
        for j in range(1,m_max-1):
            u[i,j] = u[i,j] + dt * (-g *(h[i,j]-h[i-1,j])/dx - f*u[i,j])
    for i in range(1,n_max-1):
        for j in range(0,m_max-1):
            v[i,j] = v[i,j] + dt * (-g *(h[i,j]-h[i,j-1])/dy - f*v[i,j])
    for i in range(1,n_max-1):
        for j in range(1,m_max-1):
            h[i,j] = h[i,j]  + dt * (0.5 *- 0.5*(u[i+1,j]*d/dx) + 0.5*(u[i,j]*d/dx) - 0.5*(v[
    i,j+1]*d/dy)  + 0.5*(v[i,j]*d/dy))
    t = t +dt
```

D
Implementation Fortran90

```fortran
program wave_xy
      ! Simulate free surface wave in x-y-domain
      implicit none

      !conversion
      real, parameter         :: minutes_to_seconds=60.0
      real, parameter         :: hours_to_seconds=60.0*minutes_to_seconds
      real, parameter         :: days_to_seconds=24.0*hours_to_seconds
      !constants
      real, parameter         :: g=9.81       !gravity [m s-2]
      real, parameter         :: f=1.0/(0.06*days_to_seconds) !damping time-scale [s-1]
      real, parameter         :: d=10.0       ! depth below z=0 reference [m]
      !geometry
      integer,parameter       :: n_max=11, m_max= 101 !number of nodes +, i.e. one more than the number of cells o
      real, parameter         :: x_length=10000.0, y_length=10000.0
      real                    :: dx=x_length/(m_max-1.0)
      real                    :: dy=y_length/(n_max-1.0)
      !time
      real                    :: t_start=0.0
      real                    :: t_stop=50.0*minutes_to_seconds
      real                    :: dt=0.1*minutes_to_seconds !dt=0.1*minutes_to_seconds
      !initial state
      real,dimension(m_max,n_max) :: u=0.0,v=0.0,h=0.0

      !temporary variables
      integer                 :: istep
      real                    :: t

      !temporary
      integer                 :: m,n

      !initialization
      do m=1,m_max
        do n=1,n_max
          h(m,n) = exp(-(m*dx-x_length/2.)**2./(1000.**2.)-(n*dy-y_length/2.)**2./(1000.**2.))
        end do
      end do

      !time loop
      t=t_start
      istep=0
      do while (t<t_stop)
         !if (mod(istep,10)==0) then
         !    print *, "t=",t
         !    print *, "h=", h
         !endif
         ! u
         do n=1,(n_max-1)
         do m=2,(m_max-1)
            !if ((n==5) .and. (m==5)) then
            !     print *, "start debugging here"
            !endif
            u(m,n) = u(m,n)                                          &
                    +dt*(                                            &
                        -g*(h(m,n)-h(m-1,n))/dx                      &
                        -f*u(m,n)                                    &
                        )
         enddo
         enddo
         ! v
         do n=2,(n_max-1)
         do m=1,(m_max-1)
            v(m,n) = v(m,n)                                          &
                    +dt*(                                            &
                        -g*(h(m,n)-h(m,n-1))/dy                      &
                        -f*v(m,n)                                    &
                        )
         enddo
         enddo
         ! h
         do n=1,(n_max-1)
         do m=1,(m_max-1)
            h(m,n) = h(m,n)                                          &
                    +dt*(                                            &
                        - 0.5*(u(m+1,n)*d/dx)                        &
                        + 0.5*(u(m   ,n)*d/dx)                       &
                        - 0.5*(v(m,n+1)*d/dy)                        &
                        + 0.5*(v(m,n   )*d/dy)                       &
                        )
         enddo
         enddo
         ! update vars
         t=t+dt
      enddo
end program
```

Bibliography

[1] Cuda explained - why deep learning uses gpus. https://deeplizard.com/learn/video/6stDhEA0wFQ, 2020.

[2] A.Arakawa and V.R. Lamb. Computational design of the basic dynamical processes of the ucla general circulation model. *Methods in Computational Physics: Advances in Research and Applications*, 17:173–265, 1997. doi: https://doi.org/10.1016/B978-0-12-460817-7.50009-4.

[3] Atılım Günes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, 18(1):5595–5637, 2017.

[4] Andrew M. Bradley. *PDE-constrained optimization and the adjoint method.* 2010.

[5] M. B. Gijzen C. Vuik, F. J. Vermolen and M. J. Vuik. *Numerical Methods for Ordinary Differential Equations*. VSSD, 2007.

[6] Lamberta Billy Daoust, Mark and Abhinav Prakash. Partial differential equations. https://github.com/tensorflow/examples/blob/master/community/en/pdes.ipynb, 2019.

[7] AJC Barré de Saint-Venant. Théorie et équations générales du mouvement non permanent des eaux courantes. *Comptes Rendus des séances de l'Académie des Sciences, Paris, France, Séance*, 17(73):147–154, 1871.

[8] C. Lubivh E. Haier and G. Wanner. Geometric numerical integration illustrated by the st¨ormer–verlet method. *Acta Numerica*, 12:399–450, 2003. doi: https://doi.org/10.1017/S0962492902000144.

[9] Hesham El-Rewini and Mostafa Abd-El-Barr. *Advanced computer architecture and parallel processing*, volume 42. John Wiley & Sons, 2005.

[10] Abadi M. et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

[11] Abadi M. et al. tf.gradienttape. https://www.tensorflow.org/api_docs/python/tf/GradientTape, 2020.

[12] Abadi M. et al. Introduction to tensors. https://www.tensorflow.org/guide/tensor, 2020.

[13] Andreas Griewank and Andrea Walther. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45, 2000.

[14] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2008. ISBN 978–0–898716–59–7. URL http://bookstore.siam.org/ot105/.

[15] Richard Haberman. *Applied Partial Differential Equations with Fourier Series and Boundary Value Problems*, volume 5. Pearson Education Limited, 2014.

[16] F.Vermolen H. Kraaijenvanger J. van Kan, A.Segal. *Numerical Methods for Partial Differential Equations*. Delft Academic Press / VSSD, 2019.

[17] Wesley M Johnston, JR Paul Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM computing surveys (CSUR)*, 36(1):1–34, 2004.

[18] H Lewy, K Friedrichs, and R Courant. Über die partiellen differenzengleichungen der mathematischen physik. *Mathematische annalen*, 100:32–74, 1928.

[19] Zulfiqar A Memon, Fahad Samad, Zafar Rehman Awan, Abdul Aziz, and Shafaq Siraj Siddiqi. Cpu-gpu processing. *International Journal of Computer Science and Network Security*, 17(9): 188–193, 2017.

[20] Fedor Mesinger. A method for construction of second-order accuracy difference schemes permitting no false two-grid-interval wave in the height field. *Tellus*, 25(5):444–458, 1973.

[21] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020. URL `https://developer.nvidia.com/cuda-toolkit`.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[23] Otmane Souhar and Jean-Baptiste Faure. Approach for uncertainty propagation and design in saint venant equations via automatic sensitive derivatives applied to saar river. *Canadian journal of civil engineering*, 36(7):1144–1154, 2009.

[24] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL `http://arxiv.org/abs/1605.02688`.

[25] Cornelis Boudewijn Vreugdenhil. *Numerical methods for shallow-water flow*, volume 13. Springer Science & Business Media, 2013.

[26] Robert E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.