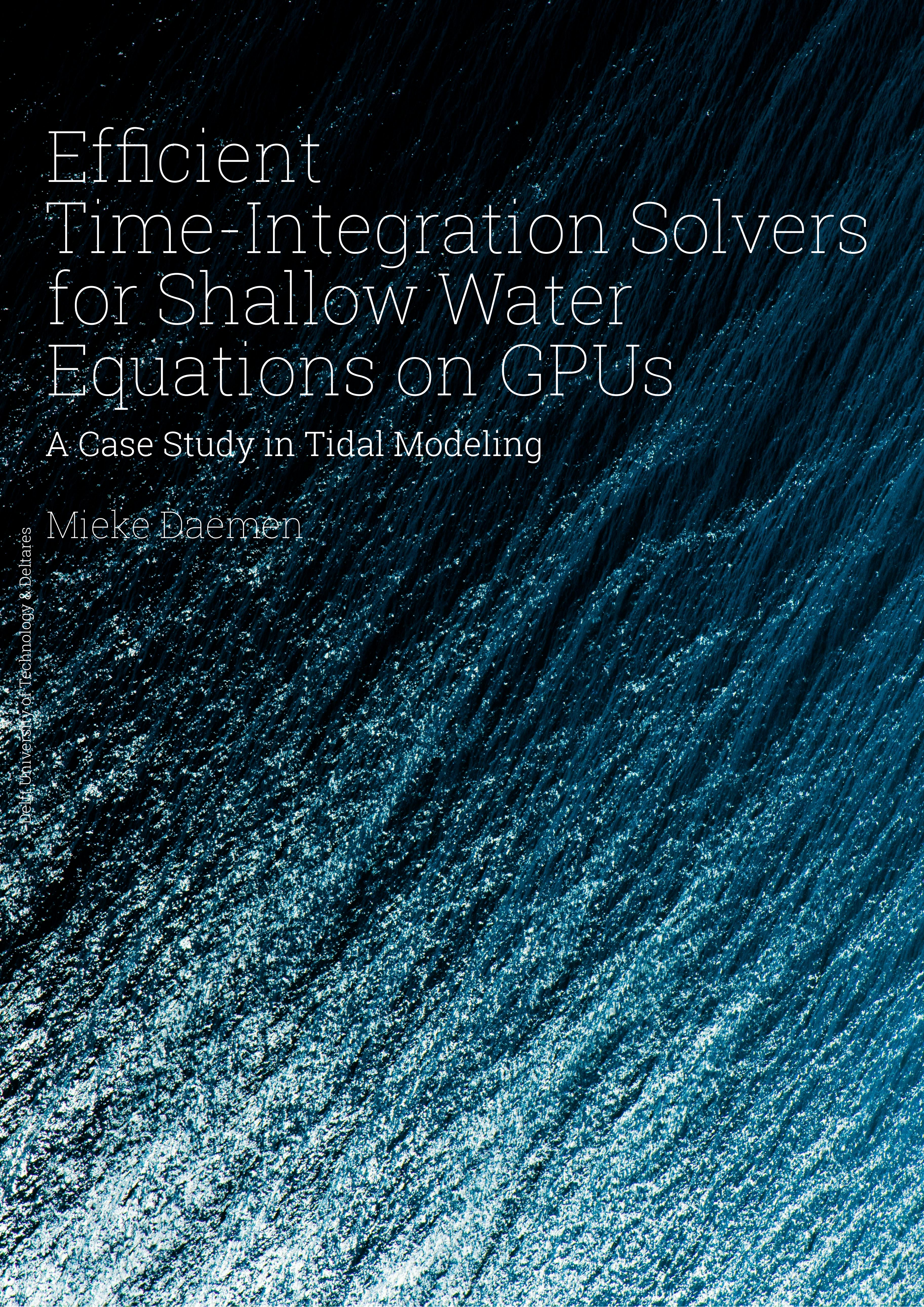


Efficient Time-Integration Solvers for Shallow Water Equations on GPUs

A Case Study in Tidal Modeling

Mieke Daemen

Delft University of Technology & Deltares

Efficient Time-Integration Solvers for Shallow Water Equations on GPUs

A Case Study in Tidal Modeling

by

Mieke Daemen

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on February 7, 2025 at 2:00 PM.

Student number:	4925106	
Thesis committee:	Prof. Dr. ir. M. Verlaan,	TU Delft & Deltares, supervisor
	Dr. J. Zhao,	TU Delft & Deltares, supervisor
	Prof. dr. ir. M. B. van Gijzen,	TU Delft
	ir. F. Zijl	Deltares

Preface

With this MSc thesis, I present my investigation into GPU-based time-integration solvers for modeling ocean tides. This work was conducted as part of my graduation internship, funded by Deltares, and contributes to the completion of my MSc degree in Applied Mathematics.

In writing this report, I have assumed that readers possess a basic understanding of numerical analysis and are familiar with concepts such as time-integration and finite-difference methods. However, familiarity with the specific time-integration schemes used in this research is not required. These methods and their properties are introduced and explained in Chapter 3.

I would like to thank Deltares for giving me the opportunity to do a graduation internship. I also wish to express my gratitude to Prof. Dr. Martin Verlaan and Dr. Jing Zhao for supervising my project. In particular, I am thankful to Dr. Jing Zhao for onboarding me at Deltares and assisting with practical matters, and to Prof. Dr. Martin Verlaan for introducing me to a very interesting and highly relevant research topic.

Within Deltares, I would like to thank Maarten Pronk (MSc) and Dr. Eric de Goede. I am grateful to Maarten Pronk for assisting me in creating my own functions within the OrdinaryDiff.jl framework and for teaching me valuable tricks in the Julia programming language. I also thank Dr. Eric de Goede for sharing scientific articles on solving Shallow Water Equations on GPUs and for introducing me to past research in GPU computing.

Furthermore, I would like to thank Prof. Dr. Martin van Gijzen and Dr. Firmijn Zijl for their participation in the graduation committee.

Finally, I would like to express my gratitude to my friends and family for their support throughout this journey. In particular, I am thankful to my aunt, Liesbeth Timmer, for her financial support of my espresso consumption. I also thank Roos van den Broek for encouraging me to take breaks and join her on refreshing walks.

Acknowledgments: In the process of writing this thesis, I used ChatGPT for redacting and refining certain sections of the text. However, all content was originally written by me, with ChatGPT serving solely as a tool for editing and polishing.

Mieke Daemen
Delft, January 2025

Summary

Graphics Processing Units (GPUs) represent a modern hardware innovation that can significantly accelerate shallow water equation (SWE) solvers. However, research on GPU-based solvers specifically designed for ocean applications remains limited.

The SWEs for ocean applications can be discretized using finite differences, resulting in a system of time-dependent ordinary differential equations (ODEs). These ODEs can then be solved on a GPU using various time-integration schemes, which can be categorized as explicit and implicit methods.

For explicit methods, the stability criterion imposes constraints on the allowable time step size, dictated by the Courant–Friedrichs–Lewy (CFL) condition. According to the CFL condition, the maximum allowable time step decreases with increasing water depth and finer grid resolutions. As the grid resolution increases, not only does the computational workload grow due to the larger number of grid points, but the reduction in time-step size necessitates a greater number of iterations to simulate the same physical time span. This combination significantly increases the overall computational cost.

In contrast, implicit methods are not constrained by the CFL condition, allowing for the use of larger time steps. However, each time step requires solving a nonlinear system of equations, which can be challenging to implement efficiently on a GPU.

The primary aim of this thesis is to evaluate the performance of various time-integration solvers implemented on a GPU for a simplified tidal model of the North Sea. The investigation includes a comparison of explicit second-, third-, and fourth-order Runge-Kutta (RK) and multistep methods as well as several second-order implicit schemes. All numerical schemes will be implemented on the GPU using the Julia programming language.

The DifferentialEquations.jl package was used to implement various explicit methods. On the GPU, the fourth-order RK4 scheme and the second-order, five-stage Optimized Runge-Kutta method (ORK256) showed the best performance. Both RK4 and ORK256 have stability conditions that allow larger time steps compared to other explicit methods. As a result, these two methods required fewer iterations to simulate the same time span, reducing computational times.

Moreover, a novel approach is proposed for solving the tidal model and addressing the nonlinear systems within the implicit time iterations. This approach is a combination of an implicit scheme with a pseudo-time-stepping approach and a multi-level technique. Various implicit schemes employing different pseudo-time-stepping solvers are analyzed and compared. The most efficient scheme in this study is the second-order Singly Diagonally Implicit Runge-Kutta (SDIRK2), combined with semi-explicit (IMEX) inner iterations within the pseudo-time solver. For problems on a high-resolution grid, a multi-level strategy was incorporated within the SDIRK2-solver. This solver will be referred to as SIM, short for SDIRK2-IMEX-Multilevel. For a high-resolution grid, the computational time was lower for the SIM method compared to both the RK4 and ORK256 methods. Unlike the explicit methods, the computational workload for the SIM method did not scale with the CFL condition. On a GPU, the SIM solver is the most time-efficient for high-resolution simulations of the simplified North Sea model.

Contents

1 Introduction

Forecasters depend on efficient numerical software to ensure accurate and timely sea-level predictions. The parallel implementation of numerical algorithms is essential for rapidly obtaining computational results. In recent decades, GPU-based parallel computing has emerged as an approach for increasing computational efficiency.

1.1. GPU Computing

Graphics Processing Units (GPUs) were initially developed for rendering graphics in video games and other visual applications. Over time, GPUs have evolved into powerful tools for parallelizing computational workloads. Unlike the CPU, which is optimized to execute a single set of operations very quickly within a thread and can handle a limited number of threads simultaneously (typically around 10), the GPU is designed to perform thousands of threads in parallel, making it efficient for tasks with a high degree of parallelism (Nvidia, 2024). In addition, GPU delivers significantly higher instruction throughput and memory bandwidth compared to a CPU within a comparable price and power range (Nvidia, 2024).

Despite these advantages, GPUs have limitations in terms of memory. The available memory on a GPU is typically much smaller than that on a CPU. If the GPU's memory becomes saturated, data must be transferred between the GPU and CPU, which slows down performance. Therefore, memory-intensive algorithms are likely to experience slower performance on the GPU.

1.2. Solving Shallow Water Equations on GPUs

Over the last 10 to 15 years, there has been a significant amount of research dedicated to solving shallow water equations (SWEs) on GPUs. The majority of studies in this field focus on explicit time-integration schemes due to their inherent parallelism. This parallelism makes explicit methods particularly attractive for GPU implementations. A large portion of the research on GPU-based SWE solvers focus on flood inundation modeling. For example, Rak et al. (2024) developed a GPU-based solver for flash-flood simulations, utilizing a second-order explicit Leapfrog time-integration scheme combined with finite volume discretization on a structured grid. Another study by Aureli et al. (2020) evaluated a numerical scheme for a real-life river flooding scenario, employing finite volume discretization combined with second-order Heun time integration. Moreover, Chapter 5 provides an overview of various studies on modeling flood inundations using GPUs. In these studies, the water depth is often shallow, and the stability conditions on time-steps are less restrictive compared to deep water scenarios. This characteristic allows for larger time-steps, improving computational efficiency while maintaining numerical stability. In contrast, in a model with a higher water depth the stability conditions are more restrictive. In Brodtkorb and Holm, 2021, the authors present a GPU-based method for SWEs with water depths of order $\mathcal{O}(10^2) - \mathcal{O}(10^3)$ meters.

In their 2021 study, Brotkorb and Holm present a GPU implementation of a finite-volume based solver for an ocean model, utilizing a second-order explicit Strong Stability Preserving Runge-Kutta scheme

(SSPRK22). A significant performance limitation when using such an explicit method for a deep ocean model is the Courant–Friedrichs–Lewy (CFL) condition. This condition requires that when the spatial discretization step size is reduced by a certain factor, the time step must also be reduced by a similar factor, thus limiting computational efficiency. For implicit methods, the CFL condition does not apply. However, the implementation of implicit methods involves solving nonlinear systems, which can be computationally expensive on a GPU. In contrast, explicit methods primarily rely directly on function evaluations.

The choice between explicit and implicit methods involves a series of trade-offs. Therefore, for researchers and forecasters seeking to utilize GPUs for faster sea-level predictions, it is valuable to compare the performance of various implicit and explicit time-integration schemes.

1.3. Master Thesis Project

The primary aim of this thesis is to analyze the performance of various time-integration solvers implemented on a GPU for a simplified North Sea tidal model. This model is based on the two-dimensional SWEs, which are discretized spatially using finite differences. This discretization transforms the equations into a system of time-dependent ordinary differential equations (ODEs), which are subsequently solved using various explicit and implicit methods.

Initially, explicit and implicit methods are examined independently. The computational efficiency of various second-, third-, and fourth-order explicit time-integration schemes is analyzed, and the fastest scheme is selected. Additionally, an efficient nonlinear solver is developed for implicit methods. Finally, the efficient implementation of the implicit scheme is compared against the fastest explicit method.

All experiments in this study were implemented in the Julia programming language. Explicit RK methods were implemented using the DifferentialEquations.jl package, while CUDA.jl was employed to effectively parallelize matrix operations.

This thesis is organized as follows: Chapters 2 through 5 provide a comprehensive literature review. Chapter 2 introduces the SWEs and their spatial discretization. Chapter 3 discusses various time-integration methods. Chapter 4 focuses on techniques for solving nonlinear systems encountered in implicit time-integration schemes. Finally, Chapter 5 presents the principles of GPU computing and provides a brief overview of various methods from the scientific literature for GPU-based SWE solvers. Chapters 6 through 8 present tests, results, and comparisons of various time-integration solvers. Chapter 6 focuses on the implementation and comparison of different explicit time-integration schemes. Chapter 7 addresses implicit schemes, integrating them with a pseudo-time-stepping method for solving nonlinear systems. Chapter 8 incorporates a multilevel strategy into the most efficient implicit solver found in Chapter 7. Finally, in Chapter 9 a summary of the study's key conclusions are presented.

2 Shallow Water Equations (SWEs)

The Shallow Water Equations (SWEs) are a system of hyperbolic partial differential equations used to describe fluid flow in contexts where the horizontal scale is much larger than the vertical scale. They are widely applied in modeling oceanic and coastal flows, capturing dynamics from deep waters to nearshore areas. Beyond their application in oceanography, the SWEs also serve as the foundation for many weather and climate models, making them essential for understanding large-scale geophysical phenomena.

The 2-dimensional SWEs can be derived by applying depth averaging to the Navier Stokes equations (Vreugdenhil, 1994), if we assume that horizontal length scale is larger than depth. The Navier Stokes Equations describe motion of viscous fluids in 3-dimensions and express the conservation of momentum.

In this study, time and space discretization are treated independently to facilitate the comparison of different time integration schemes. This chapter focuses on the spatial discretization of the SWEs. In subsequent chapters, it is assumed that a spatially discretized form of the SWEs already exists.

The first section provides a simplified description of the SWEs. Additionally, it presents the SWEs formulated in polar coordinates, which are used to describe water flows in oceans.

The second section of this paper focuses on the spatial discretization of the SWEs. It introduces the Arakawa C-grid as the preferred technique for structuring the numerical grid. The primary motivation for choosing the C-grid lies in its widespread use in ocean and river modeling, ensuring that the model and results of this study can be readily compared to other SWE-based models.

A wide range of approaches exists for spatially discretizing the SWEs on staggered grids, including finite element methods, finite volume methods, and finite difference methods. In this study, we present a finite difference discretization using a regular grid with staggering. Furthermore also an idea for finite volume methods is presented.

2.1. Shallow Water Equations

2.1.1. Deriving the SWEs from the Navier-Stokes Equations

The 2-dimensional SWEs are derived from the 3-dimensional Navier-Stokes Equations. The Navier Stokes Equations are based of conservation of momentum and mass, and describe fluid flows. It is assumed that the fluids are incompressible. The density of an incompressible fluid does not depend on pressure, but can still depend on other variables such as temperature (Vreugdenhil, 1994, p.14).

The following equations describe conservation of momentum (Vreugdenhil, 1994, p.15) :

$$\frac{\partial \rho u}{\partial t} + \frac{\partial \rho u^2}{\partial x} + \frac{\partial \rho uv}{\partial y} + \frac{\partial uw}{\partial x} - \rho f v + \frac{\partial p}{\partial x} - \frac{\partial \tau_{xx}}{\partial x} - \frac{\partial \tau_{xy}}{\partial y} - \frac{\partial \tau_{xz}}{\partial z} = 0, \tag{2.1}$$

$$\frac{\partial \rho v}{\partial t} + \frac{\partial \rho uv}{\partial x} + \frac{\partial \rho v^2}{\partial y} + \frac{\partial vw}{\partial x} - \rho f u + \frac{\partial p}{\partial y} - \frac{\partial \tau_{xy}}{\partial x} - \frac{\partial \tau_{yy}}{\partial y} - \frac{\partial \tau_{zy}}{\partial z} = 0, \tag{2.2}$$

$$\frac{\partial \rho w}{\partial t} + \frac{\partial \rho uw}{\partial x} + \frac{\partial \rho vw}{\partial y} + \frac{\partial w^2}{\partial x} + \frac{\partial p}{\partial z} + \rho g - \frac{\partial \tau_{xz}}{\partial x} - \frac{\partial \tau_{yz}}{\partial y} - \frac{\partial \tau_{zy}}{\partial z} = 0. \tag{2.3}$$

The equations (2.1) - (2.3) are based of a Cartesian coordinates expressed in (x, y, z)-variables. The velocities given by (u, v, w) represent the velocity components in the (x, y, z)-direction, respectively. The time is given by t, pressure is p, gravitational acceleration is g and ρ is the density. The value of $f = 2\Omega sin(\phi)$ is the Coriolis parameter, where Ω represents the angular speed of the earth and ϕ is the geographic latitude. The τ-parameter represents the viscous stresses of the system.

The incompressibility condition in our system of equations is given by (Vreugdenhil, 1994, p.17):

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0. \tag{2.4}$$

When modelling water flows in rivers and oceans a set of boundary-conditions are defined. The water depth is given by $a(x, y, t) = h(x, y, t) - z_b(x, y)$, where $h(x, y, t)$ is the water surface level and $z_b(x, y)$ is the bottom level. On the bottom of the ocean no-slip boundary conditions are defined and also no flux is going through the bottom:

$$u\frac{\partial z_b}{\partial x} + v\frac{\partial z_b}{\partial y} - w = 0, \quad z = z_b.$$

The flux through the surface is zero as well:

$$\frac{\partial h}{\partial t} = u\frac{\partial h}{\partial x} + v\frac{\partial h}{\partial y} - w = 0, \quad z = h.$$

Furthermore, at the free surface the pressure is equal to the atmospheric pressure $p = p_s$. On the surfaces the following shear stresses are defined:

$$\tau_{s,x} = -\tau_{xx}\frac{\partial h}{\partial x} - \tau_{xy}\frac{\partial h}{\partial y} + \tau_{xz}, \quad z = h.$$

Similar stresses are defined in the y-direction. Depending on the model also stress relations can be formulated on the bottom of the oceans.

Pressure Approximation

For 2-dimensional shallow water modelling we assume that the vertical length scale is smaller than the horizontal length scale. The vertical length scale is equal to the water depth a. The horizontal scales for x and y are given by L. The velocity components of u, v have a characteristic velocity U. The equations (2.1) and (2.2) representing momentum conservation in the x- and y-direction are of order $\mathcal{O}(UL^{-1})$. The z-component of Navier-Stokes is of order $\mathcal{O}(Wa^{-1})$, where W is the characteristic z-velocity component. For the third equation to of the same size as the first and second we need $W = \mathcal{O}(UaL^{-1})$ and we assume $\frac{a}{L} << 1$. If one tries to compare the sizes of all the components in the third equation, one finds that the acceleration terms and advective terms are of sizes $\mathcal{O}(\frac{Ua^{3/2}}{g^{1/2}L^2})$, $\mathcal{O}(\frac{U^2a}{gL^2})$, respectively. In contrast, the gravitational term is ρg and thus a lot larger compared to the other components. Hence only the pressure gradient and the gravitational terms remain in the equation and hydro-static pressure exist (Pugh and Woodworth, 2014, p.361):

$$\frac{\partial p}{\partial z} = -\rho g, \Rightarrow \tag{2.5}$$

$$p = \rho g(h - z) + p_s. \tag{2.6}$$

With this equality we find that for constant values of ρ, the following pressure gradients in the first and second equation of momentum preservation can be found:

$$\frac{\partial p}{\partial x} = -\rho g \frac{\partial h}{\partial x} + \frac{\partial p_s}{\partial x}, \quad \frac{\partial p}{\partial y} = -\rho g \frac{\partial h}{\partial y} + \frac{\partial p_s}{\partial y}.$$

Finally, the 2D- equations can be derived by depth averaging the Navier stokes equations with the compressiblity condition and substituting the boundary conditions.

$$\bar{u} := \frac{1}{a}\int_{z_b}^{h} u\,dz, \quad \bar{v} := \frac{1}{a}\int_{z_b}^{h} v\,dz.$$

For simplicity we from now on denote $\bar{u}$ and $\bar{v}$ as u and v.

This Chapter discusses the depth averaged, 2-dimensional SWEs. However, 3-dimensional SWEs also do exist. Modelling SWEs in 2-dimensions can lead to problems if for example the temperature in the ocean changes, in cases where cooling occurs on the surface. For those temperature changes, large vertical accelerations can occur. If we add those large vertical accelerations, then we no longer have the hydrostatic assumption and the model is non-hydrostatic. For example, in SWASH non-hydrostatic waves in coastal waters can be modelled.

2.1.2. Simplified forms of SWEs

By substituting boundary conditions, the hydrostatic pressure assumption and applying depth averaging to the Navier-Stokes equation, two-dimensional SWEs can be derived. The SWEs can be expressed in both conservative and non-conservative forms.

Conservative Form

The conservative form of the SWEs is derived from depth evergaging the SWEs. According to Vreugdenhil (1994, p.40), The two-dimensional SWEs in conservative form have the following structure:

$$\begin{aligned}
\frac{\partial h}{\partial t} + \frac{\partial (au)}{\partial x} + \frac{\partial (av)}{\partial y} &= 0,\\
\frac{\partial (au)}{\partial t} + \frac{\partial (au^2)}{\partial x} + \frac{(auv)}{\partial y} - fav + ga\frac{\partial h}{\partial x} + \text{stresses} + \text{body forces} &= 0,\\
\frac{\partial (av)}{\partial t} + \frac{\partial (auv)}{\partial x} + \frac{(av^2)}{\partial y} + fau + ga\frac{\partial h}{\partial y} + \text{stresses} + \text{body forces} &= 0.
\end{aligned}$$

Noticing that the system is shown as a balance of mass-flux and momentum-flux, the system as indeed a conservative form. Subsequently, an SWE in conservative form can be represented in the following way: $\mathbf{v}_t = \mathbf{f_1}(\mathbf{v})_x + \mathbf{f_2}(\mathbf{v})_y + \mathbf{s}$.

The coordinate system with (x, y) is given by Cartesian coordinates. The variables (u, v) are the depth-averaged velocity components in the x- and y- directions, respectively. The depth of the ocean is denoted by a and water elevation levels are given by $h(x, y)$. The gravitational acceleration is g.

Non-Conservative Form

Applying the chain rule to the SWEs in conservative form, yields the following system of equations in non-conservative form (Vreugdenhil, 1994, p.40):

$$\frac{\partial h}{\partial t} + \frac{\partial (au)}{\partial x} + \frac{\partial (av)}{\partial y} = 0, \tag{2.7}$$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} - fv + g\frac{\partial h}{\partial x} + \text{stresses} + \text{body forces} = 0, \tag{2.8}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + fu + g\frac{\partial h}{\partial y} + \text{stresses} + \text{body forces} = 0. \tag{2.9}$$

The SWEs in both conservative and non-conservative form are similar. However, for selecting a numerical method, the distinction between conservative and non-conservative form is relevant. For instance, for applying finite volume based spatial discretization with flux-limiters, having a system of equations in conservative form is necessary.

2.1.3. Model in Spherical Coordinates

For applications where the model covers a large portion of the ocean, the earths surface cannot be regarded as a flat plane. Hence, for large ocean models it is preferable to use spherical coordinates (Klingbeil et al., 2018). In (Westerink et al., 2008) a model for hurricane storm surges with polar coordinates is presented. This model has the following structure:

$$\frac{\partial h}{\partial t} + \frac{1}{R\cos(\phi)}\left(\frac{\partial(au)}{\partial\lambda} + \frac{\partial(av\cos(\phi))}{\partial\phi}\right) = 0, \tag{2.10}$$

$$\begin{aligned}\frac{\partial u}{\partial t} + \frac{1}{R\cos(\phi)}u\frac{\partial u}{\partial\lambda} + \frac{1}{R}v\frac{\partial u}{\partial\phi} - \left(\frac{u\tan(\phi)}{R} + f\right)v + \frac{1}{R\cos(\phi)}\frac{\partial}{\partial\lambda}[gh] + \\ + \text{stresses} + \text{sources} = 0,\end{aligned} \tag{2.11}$$

$$\begin{aligned}\frac{\partial v}{\partial t} + \frac{1}{R\cos(\phi)}u\frac{\partial v}{\partial\lambda} + \frac{1}{R}v\frac{\partial v}{\partial\phi} + \left(\frac{u\tan(\phi)}{R} + f\right)u + \frac{1}{R}\frac{\partial}{\partial\phi}[gh] + \\ + \text{stresses} + \text{sources} = 0,\end{aligned} \tag{2.12}$$

where λ denotes the longitude, ϕ is equal to the latitude. The radius of the earth is displayed by R and is equal to 6,370 km.

In the following sections we will explain what types of stresses and sources work on our Shallow Water Flows. Furthermore, we also explain how tides and storm-surges are modelled by adding certain sources/stresses.

Coriolis Force

The rotation of the earth changes the flows of the ocean. Since in absolute space flows of water are often moving in a uniform direction, water flows exists on a curved trajectory on the rotating sphere (Pugh and Woodworth, 2014,p. 98). Rather, on a rotating planet it is necessary to have certain types of forces working in specific angels for having the trajectory of motion along a straight line (Pugh and Woodworth, 2014,p. 98).

According to Newton's second law, in a system without acceleration the net forces on a system are equal to zero. When applying Newton's second law to movements in a coordinate system on a rotating planet, an additional form acceleration works on a system (Pugh and Woodworth, 2014, p.362). Hence an additional force works on a system, called the Coriolis force. In the equation (2.7-2.9) the Coriolis term is given by $-fv$ and fu in the momentum equations. The parameter f is the Coriolis parameter and is given by $f = 2\Omega\sin(\phi)$ (Vreugdenhil, 1994, p.15). The variables Ω and ϕ are the angular speed of the earth and the geographic latitude, respectively.

Bottom Stress

Bottom friction leads to a the damping of the flow and leads to a decrease in energy from the motion (Pugh and Woodworth, 2014,p.135). The bottom stresses are given by (Vreugdenhil, 1994):

$$\tau_{b,u} = \rho c_f u\sqrt{(u^2+v^2)}, \quad \tau_{b,v} = \rho c_f v\sqrt{(u^2+v^2)},$$

where c_f is the bottom stress coefficient, ρ is the ocean density and $\tau_{b,x}$ and $\tau_{b,y}$ represent the bottom stresses in the u and v component respectively. The stress-terms $\frac{1}{a\rho}\tau_{b,x}$ and $\frac{1}{a\rho}\tau_{b,y}$ can be added to the equations in non-conservative equations (2.11) or (2.12), respectively.

In Delft3D-FLOW the Chezy-coefficient is implemented for bottom friction (Gerritsen et al., 2008). The Chezy-coefficient is denoted by C_D. The bottom friction coefficient is given by:

$$c_f = \frac{g}{C_D^2}.$$

The bottom friction coefficient is not always linear or constant. One example of the many modification and additions is given by Westerink et. al.(2008). They employ a different bottom friction coefficient:

$$c_f = c_{f,\min}\left(1 + \frac{a_{\text{Depth}}}{a}^{\theta}\right)^{\frac{\gamma}{\theta}},$$

where a_{Depth} is the break depth. This coefficient is applied to a model with a coastline where breaking waves occur. If the dept of the ocean is larger than the breaking depth, the Chezy-formulation will still be applied.

Modelling Tides and Storm Surges

For the SWE-model in (Westerink et al., 2008), also a tide-potential term is added to both the momentum equations.

$$T_u = -\frac{1}{R\cos\phi}\frac{\partial}{\partial\lambda}(\eta\alpha), \quad T_v = -\frac{1}{R}\frac{\partial}{\partial\phi}(\eta\alpha)$$

The α and η represent the Newtonian equilibrium tide potential and the earth elasticity factor, respectively. The gradient of $\alpha\eta$ is equal to the effective tide producing force in the system (Reid, 1990p.559). Tides are defined by regular and periodic movements of sea levels and currents. Both the phases and amplitudes of periodic tidal movements are linked to periodic geophysical forces (Pugh and Woodworth, 2014, p. 9). The main geophysical forces are the changes of the gravitational field on the earth's surface caused by periodic movements of both the sun and the moon around the earth(Pugh and Woodworth, 2014, p. 9). For instance, if the moon is closer to the earth, the increased gravitational forces result in higher ocean-tides.

Understanding Newton's law of gravitation is crucial for modeling forces between the earth and the sun or moon. According Newton's gravitational law the net force of attraction between the earth and the moon can be computed as follows (Pugh and Woodworth, 2014, p.37):

$$\text{Force } = G\frac{m_e m_l}{R_l^2},$$

where m_e, m_l and R_l are the masses of the earth and moon and the distance between the two mass-centres, respectively. Furthermore, G represents the universal gravitational constant and its value is determined solely by the chosen units of mass, length, and force (Pugh and Woodworth, 2014, p.37). The universal gravitational constant, G, has dimensions of $M^{-1}L^3T^{-2}$ (Pugh and Woodworth, 2014, p.37). In SI units, its value is equal to $6.67 \times 10^{-11}\,\text{N}\,\text{m}^{-2}\,\text{kg}^{-2}$. The net force of attraction between the earth and the sun can be computed in a similar fashion. For the moon-earth system, we find that they move around a common centre of mass. The acceleration of both the earth and moon towards the mass centre is generated by the attraction force. In this 2-body system, every particle of the earth and the moon has a circular movements with the same radius corresponding to its centre of mass. The force needed for every particle of the earth to accelerate such that it performs the circular movements with the same radius, is identical for a particle in the centre of the earth. However, for particles on the earth closer to the moon then the centre particle the gravitational forces are greater than needed to maintain the circular movements. The force difference between the forces necessary for maintaining the periodic orbit and the actual forces lead to periodic tides.

For modeling periodic tides the concept of gravitational potential is used in the shallow water models. Gravitational potential is defined as the work necessary to remove a particle of unit mass with an infinite distance from a celestial body while taking the gravitational attraction force into account. The gravitational potential caused by the moon, at location P on the Earth's surface is:

$$-\frac{gm_l}{||M-P||_2},$$

where M is the moon's centre of mass. With the gravitational potential, one can find the equilibrium tide. The equilibrium tide is the water level elevation in equilibrium with the tides forces.

Finally, the equilibrium tide potential η caused by both the moon and the sun can be represented by (Reid, 1990, p.559):

$$\begin{aligned}\eta(t,\phi,\eta) = (3\sin(\phi)^2 - 1)(F_{m0}(t) + F_{s0}(t)) + \sin(\phi)(F_{m1}(\lambda,t) + F_{s1}(\lambda,t)) +& \\ \cos(\phi)^2(F_{m2}(\lambda,t) + F_{s2}(\lambda,t)),&\end{aligned} \tag{2.13}$$

where F_{ij}, for $i = m, s$, moon and sun is :

$$F_{i0} = G_i(\sin\delta_i - \frac{1}{3}),$$
$$F_{i1} = G_i \sin 2\delta_i \cos(\lambda - \lambda_i),$$
$$F_{i2} = G_i \cos{\delta_i}^2 \cos(2\lambda - 2\lambda_i),$$

and also:

$$G_i = \frac{3m_i R^4}{4m_e d_i(t)^4}, \tag{2.14}$$

where m_i is the mass of the moon or sun and $d_i(t)$ is the distance between the centre of the earth and the sun/moon. The mass of the earth is given by m_e.

The spatial parameters λ_i and δ_i, represent the longitude and declination depending on time of the moon or sun relative to the Earth reference frame, respectively.

For modeling deep ocean tides on wide, the tides are often a result of the gravitational forces (Pugh and Woodworth, 2014, p.106). In shelf seas, tides are caused by tides from the deep oceans. Researchers often do not include the tide-potential for small scale local models. The tidal oscillations included in local models via the boundary conditions.

Irregular patterns in surface level elevation in oceans are a result of non-tidal components in the SWE-model. Non-tidal ocean movements are often caused by changes in wind and air pressures. For adjusting surface water elevation levels directly to air pressure changes, the local inverse barometer (LIB) relation exist (Pugh and Woodworth, 2014,p.155):

$$\Delta p = -\rho g \Delta h,$$

where p_s is the atmospheric air pressure. However this relation only holds if the atmospheric air pressure is considered to be the only meteorological forcing an only forcing in an ocean model.

The atmospheric air-pressure term can be included in the model by adding the following source-terms given in polar coordinates:

$$\frac{1}{R\cos\phi}\frac{\partial}{\partial\lambda}(\frac{p_s}{\rho}), \quad \frac{1}{R}\frac{\partial}{\partial\phi}(\frac{p_s}{\rho}),$$

to the equations (2.11) and (2.12), respectively.

Another meteorological forcing in the SWEs is given by the wind stress. Since storm surges are often induced by large wind-speeds, modelling wind-stresses is crucial for modelling non-tidal ocean waves.

Wind Stress

Let Variables $\tau_{s\lambda}$ and $\tau_{s\phi}$ in equations represent the λ and ϕ components of stresses induced by wind, respectively. They can be formulated as (Reid, 1990, p. 558):

$$\tau_{s\lambda} = \rho_a C_d W^2 \cos(\theta), \quad \tau_{s\theta} = \rho_a C_d W^2 \sin(\theta),$$

where W denotes the wind speed at the height of 10 meters above the sea surface, θ depicts the azimuth measured counterclockwise from east to the direction of the wind flow. Furthermore, ρ_a represents the density of the air at the oceans surface and C_d is the drag coefficient.

Subsequently, the total magnitude of the wind stresses is computed as follows in Delft3D-FLOW (Gerritsen et al., 2008, p.32):

$$|\tau_s| = \rho_a C_d W^2.$$

The drag coefficient C_d depends on W and its values range from 10^{-3} for low wind speeds to 3×10^{-3} for very violent storms (Reid, 1990, p. 558). For modelling storm surges, choosing the values of the wind coefficient is essential.

The drag coefficient is given by Garratt's formula (Garratt, 1977):

$$C_d = (0.75 + 0.067W) \times 10^{-3}.$$

This formula is valid for values of W between 4 and 45 m/s (Reid, 1990, p.558). Many storm surge models utilize a constant value of 3×10^{-3}, if wind-speeds are assumed to be high (Reid, 1990, p.558).

Thus, the wind-stresses can be included in the SWE-model in polar coordinates by adding the following additional stress-terms:

$$-\frac{\tau_{s\lambda}}{a\rho_0}, \quad -\frac{\tau_{s\lambda}}{a\rho_0},$$

in the equations (2.11) and (2.12), respectively.

In (Pugh and Woodworth, 2014,p.156) an extremely simplified approximation for a one-dimensional storm surge is given:

$$\frac{\partial h}{\partial x} \approx \frac{\rho_a C_d W^2}{\rho_0 g z_b},$$

which corresponds a steady state momentum equation in (2.9) without advection, Coriolis and other stresses. This simple equation can be used for estimating a storm surge. For a storm with a wind-speed of 30 m/s, blowing over 200 km of water and and a 30 m deep ocean, we can expect the water levels to rise with approximately 1.6 m (Pugh and Woodworth, 2014,p. 157).

Horizontal Eddy Viscosity

A model might incorporate an additional parameter for horizontal eddy viscosity, which serves to spatially smooth the current fields within the model (Pugh and Woodworth, 2014, p.156). This smoothing is frequently a rough attempt to represent the missing physics of energy dissipation in a simplified set of 2D SWEs, arising from internal waves and tides (Pugh and Woodworth, 2014, p.156). Since, in our study we include a tidal component and bottom friction, we choose not to include this horizontal eddy viscosity.

To the momentum equations of the non conservative SWEs in (**eq:SWE2**) and (**eq:SWE2**), terms of the following type can be added

$$\nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \quad \text{and} \quad \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right),$$

respectively, where ν is the horizontal viscosity coefficient. As a result, if we add these components to the SWEs the system of equations becomes parabolic.

2.1.4. Initial and Boundary Conditions

Closure of the system of SWEs requires both initial conditions and boundary conditions. For the predictions of storm surges the initial conditions are $u, v, h = 0$ for $t = t_0$.

For modelling flows both closed and open boundary conditions are defined within Delft3D-FLOW (Gerritsen et al., 2008). Closed boundary conditions are defined along coastal lines and river banks. Closed boundaries are considered natural. On a closed boundary the velocities normal to the boundary-contour are chosen to be equal to zero. Another boundary condition type is an open boundary condition. Open boundary conditions are considered artificial and should be defined such that water flows freely through them. For example when modelling an ocean flows, the open boundaries are defined on the ocean itself.

2.2. Spatial Discretization

For solving the SWEs in the previous section numerically the first step is discretizing the equations in both space and time. In this subsection spatial discretization is discussed. In the next chapter discretization in the temporal direction is adressed.

A numerical mesh covering the computational domain is needed for the spatial discretization. In our study a structured grid consisting of rectangular cells will be applied.

2.2.1. Staggered Spatial Grids

One can position all the variable at the same grid-points on a collocated grid. A potential problem of a collocated grid is partial decoupling of even and odd nodes, for cases where one applies central discretization. The partial decoupling can be circumvented by setting up a staggered grid.

Whereas on a collocated grid all the variables are assigned to the same grid-points, on a on a staggered grid different variables are defined on different grid-points. Working with staggered grids has multiple benefits. A significant advantage would be that fewer discrete state variables are needed compared to spatial-discretizations on collocated grids, while the numerical errors do not change (De Goede, 2020). Another reason for selecting staggered grids, according to Stelling (1983), would be the fact that spurious oscillations are more effectively reduced.

For discretizing 2-dimensional SWEs on a staggered grid different types of grids can be selected. Tree choices of staggered grids are the Arakawa B,D and D grid (Miller, 2007,p.42). In figure one can observe different orderings of variable positions. On a Arakawa A-grid, the pressure and the velocities in x- and y- directions are all defined on the same points. On a B-grid the values h are estimated at the corners of the cell, while the velocities are assigned to the centre of the grid-cell. On a C-grid the values of h are approximated at the centre of every grid-cell. The velocity u is approximated in the middle of the upper and lower boundary of cell, while v is approximated in the centre of the left and right boundary of the cell. Furthermore, on a D-grid a very similar ordering is applied. However, the values of u are positioned at the left and right boundary, while the positions of v are placed on the upper and lower boundary.

According to (Klingbeil et al., 2018), the B- and C- Arakawa grids are used for ocean modelling on structured grids. For ocean modelling a common habit is selecting a B-grid for models on a coarse grid and C-grid for models on fine meshes (Miller, 2007,p.46). According to De Goede, 2020, the most commonly used grid is C-Arakawa. He states that in about 85% of cases C-grids are applied to shallow water model on structured grids. In our study a C-grid is chosen, since the prevalence of C-grids in coastal ocean models will make it less difficult for us to compare our results with different existing shallow water models.

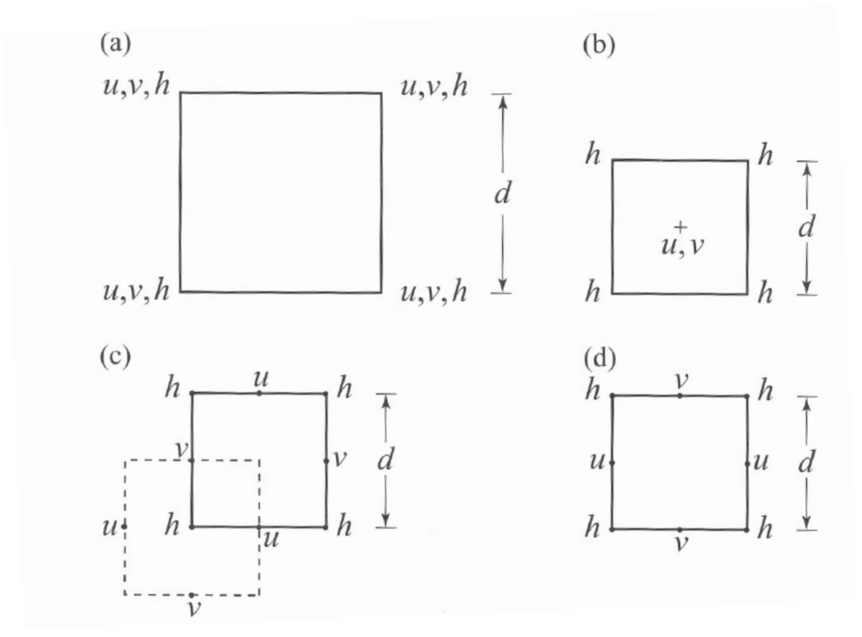


Figure 2.1: Arakawa Grid-types (Miller, 2007)

2.2.2. Finite Difference and Finite Volume Methods for Spatial Discretization

Both finite volume methods and finite difference methods are commonly used to spatially discretize the SWEs on structured grids.

For finite differences methods, derivatives are approximated with finite differences. With Taylor approximations, a finite difference approximation can be found:

$$f(x+h) = f(x) + h\frac{df}{dx}(x) + \frac{h^2}{2}\frac{d^2f}{dx} + \mathcal{O}(h^3). \tag{2.15}$$

where h is defined as a numerical grid distance. Furthermore (2.15 can be rewritten as follows:

$$\frac{df}{dx}(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h) \tag{2.16}$$

The estimation in (2.16) is referred to as forward difference approximation, since the values of the derivative in x are estimated with both function evaluations in x and $x+h$. The forward difference method is called a first order method, since the error between the derivative in x and its approximation is of order $\mathcal{O}(h)$. Other finite difference approximation schemes for first order derivatives are backward differences and central differences. For backward differences the values of the derivative in x are estimated with function evaluations in x and $x-h$. With the function evaluations in $x+h$ and $x-h$, central difference approximations are computed.

In contrast, finite volume methods approximations are not based on grid-points but on grid-cells of a domain (LeVeque, 2002). For applying a finite volume method one needs to partition the domain in to subdomains Ω_i. These subdomains are referred to as control volumes. Integrating $\nabla \cdot \mathbf{f}$ on a control volume by applying Gauss theorem yields:

$$\iint_{\Omega_i} \nabla \cdot \mathbf{f}\, dS = \int_{\partial\Omega_i} \mathbf{n} \cdot \mathbf{f}\, dx, \tag{2.17}$$

where $\mathbf{n}$ is the outward normal vector to $\partial\Omega_i$ (Knabner and Angermann, 2021). Different numerical integration schemes exists to approximate the right hand side of (2.17). To summarize, the finite volume method is a flux based numerical scheme. At boundaries of a control volumes the flow is approximated and used to compute a derivative. Hence finite volume methods are conservative. For applications where conservation laws are defined, the conservative property is a major benefit.

2.2.3. Discretization of Shallow Water Equations Using Finite Differences

With finite differences the SWEs can be discretized. To illustrate this point, a linearized SWE-model is spatially discretized. First the equations from (2.7)-(2.9), without bottom friction, advection and Coriolis are simplified:

$$\frac{\partial h}{\partial t} = -D\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right), \tag{2.18}$$

$$\frac{\partial u}{\partial t} = -g\frac{\partial h}{\partial x}, \tag{2.19}$$

$$\frac{\partial u}{\partial t} = -g\frac{\partial h}{\partial y}, \tag{2.20}$$

, where D denotes the depth below the reference. The equations (2.18)-(2.20) can be discretized in the spatial direction with both forward and backward finite difference schemes:

$$\begin{aligned} \left.\frac{\partial h}{\partial x}\right|_{x=x_i, y=y_i} &\approx \frac{h_{i+1,j} - h_{i,j}}{\Delta x}, \\ \left.\frac{\partial u}{\partial x}\right|_{x=x_{i-1/2}, y=y_i} &\approx \frac{u_{i,j} - u_{i-1,j}}{\Delta x}. \end{aligned}$$

The SWEs in (2.7)-(2.9) contain advective-terms. In Delft3D-FLOW the advective terms are also part of the model. The advection-components are spatially discretized with finite differences in Delft3D-FLOW.

Spatial Discretization in Delft3D-FLOW

One of four discretization techniques for the advective terms in the SWEs used in Delft3D-FLOW is the cyclic method (Gerritsen et al., 2008). For river and flooding simulations other discretization schemes are used in Delft3D-FLOW. The cyclic method in Delft3D-Flow is a combination of second order central differences with a second order upwind scheme.

The second order differences are:

$$u\frac{\partial u}{\partial x}\Big|_{x=x_i,y=y_j} \approx u_{i,j}\frac{u_{i+1,j}-u_{i-1,j}}{2\Delta x}.$$

Since only applying second order differences to a problems leads to problems with diagonal dominance of a system, a second order upwind scheme is added as well (De Goede, 2020). The following second order upwind scheme is used (Gerritsen et al., 2008):

$$u\frac{\partial u}{\partial x}\Big|_{x=x_i,y=y_j} \approx \begin{cases} u_{i,j}\frac{u_{i-2,j}-4u_{i-1,j}+3u_{i,j}}{2\Delta x}, & u_{i,j}\geq 0 \\ u_{i,j}\frac{-u_{i+2,j}+4u_{i-1,j}-3u_{i,j}}{2\Delta x}, & u_{i,j}<0. \end{cases}$$

These two time integration-schemes are applied to different stages of an alternating-direction implicit (ADI) scheme, resulting in a third order phase-error reduction for every time-step (De Goede, 2020).

A potential disadvantage of using a second order upwind method on a GPU, is a larger stencil. An alternative to the second order upwind scheme is a finite volume method (FVM) with a flux limiter.

2.2.4. Flux-Based Spatial Discretization Schemes

For applying a finite volume method it is necessary to have a system of equations in conservative form. The advection terms in the 2-dimensional SWEs can be presented in conservative form and thereafter be spatially discretized with an FVM scheme. An example of an equation in conservative form, with derivatives with respect to x and y is:

$$u_t + f(u)_x + g(u)_y = 0.$$

The variable $u(t)$ is discretized as $u_{ij}(t) = u(t, x_i, y_i)$ and used for defining an ordinary differential equation. We try to estimate the fluxes (LeVeque, 2002):

$$F_{i-1/2,j}(u(t)) \approx \frac{1}{\Delta y}\int_{y_{j-1/2}}^{y_{j+1/2}} f(u(t, x_{i-1/2}, y))\,dx, \tag{2.21}$$

$$G_{i,j-1/2}(u(t)) \approx \frac{1}{\Delta x}\int_{x_{j-1/2}}^{x_{j+1/2}} g(u(t, x, y_{i-1/2}))\,dx. \tag{2.22}$$

These fluxes correspond to the fluxes along local boundaries at $x = x_{i-1/2}$ and $y = y_{j-1/2}$.

With the notation of these fluxes the following time depended ODE can be derived.

$$\frac{du_{ij}(t)}{dt} = -\frac{1}{\Delta x}(F_{i+1/2,j}(t) - F_{i-1/2,j}(t)) - \frac{1}{\Delta y}(G_{i,j+1/2}(t) - G_{i,j-1/2}(t)). \tag{2.23}$$

The problem (2.23) can be solved with number of different time integration schemes. An overview of different time-integration solves can be found in chapter 3.

Flux-Limiters

The fluxes given by (2.22) can be discretized with high and low resolution schemes (Griffiths and Schiesser, 2012). Depending on the gradients a flux-limiter can be used to select different schemes.

$$F_{i-1/2,j}(u(t)) = F^L_{i-1/2,j}(u(t)) - \theta(rx_{ij})(F^H_{i-1/2,j}(u(t)) - F^L_{i-1/2,j}(u(t))), \tag{2.24}$$

$$G_{i,j-1/2}(u(t)) = G^L_{i,j-1/2}(u(t)) - \theta(ry_{ij})(G^H_{i,j-1/2}(u(t)) - G^L_{i,j-1/2}(u(t))). \tag{2.25}$$

The variables G^L and H^L correspond to the low resolutions schemes. The variables G^H and H^H represent the the higher resolution schemes. The function θ is a flux-limiter. The variables rx and ry represent approximated ratios of gradients(Griffiths and Schiesser, 2012):

$$rx_{ij} = \frac{u_{ij} - u_{i-1,j}}{u_{i+1,j} - u_{ij}}, \quad ry_{ij} = \frac{u_{ij} - u_{i,j-1}}{u_{i,j+1} - u_{ij}}.$$

If the value of θ is equal to 0, the low resolution scheme is chosen. If the value of θ is equal to 1, the high resolution scheme is chosen.

Total Variation Diminishing (TVD)

How does one choose a flux limiter? An important property for analyzing flux limiters is the concept of total variation. This concept was introduced in 1983 by Ami Harten (Harten, 1983) for developing accurate and robust numerical schemes while also reducing spurious oscillations.

The total variation for a discrete grid function is (LeVeque, 2002):

$$TV(u(x,t^n)) = \sum_i |u(x_{j+1},t^n) - u_(x_j,t^n)|,$$

where t^n is the n-th time-step and x_j is the j-th grid-point. A numerical method is total variation diminishing (TVD) if :

$$TV(u(x,t^{n+1}) \leq TV(u(x,t^n)).$$

If we want a flux-limiter method to be TVD, the flux limiter needs to satisfy the following constraints (Leveque, 2002):

$$0 \leq \frac{\theta(r_1)}{r_1} \leq 2, \quad \text{and,} \quad 0 \leq \theta(r_2) \leq 2.$$

With these constraints a Shweby region can be defined. If The flux-limiter has only values in the grey area of figure (2.2) then the flux limiter is TVD (Font, 2015). Also different limiter-functions are shown in the figure and in table 2.1. Often a flux-limits are selected based on a trial-and-error procedure.

Flux- limiter	$\theta(r)$	$\lim_{R\to\infty}\theta(r)$
Minmod	$\max(0,\min(1,r))$	1
Monotonized Central (MC)	$\max(0,\min(2r,(0.5)(1+r),2)$	2
Superbee	$\max(0,\min(2r,1),\min(r,2))$	2
Van Leer	$(r+\|r\|)/(1+r)$	2
Van Albada	$(r^2+r)/(r^2+1)$	1

Table 2.1: Most Common limiters (Griffiths and Schiesser, 2012)

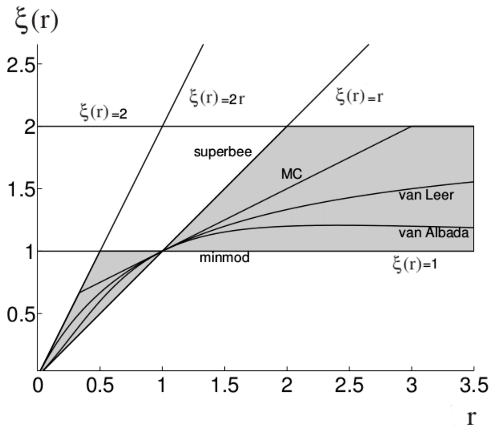


Figure 2.2: Sweby Region (Font, 2015)

High-Low Order Flux- schemes

In equation (2.25) high and low resolution schemes are presented. For a low resolution scheme a first order upwind method could be chosen such that:

$$F^L_{i+1/2,j}(u(t)) = f(u_{ij}(t)), \quad F^L_{i-1/2,j}(u(t)) = f(u_{i-1,j}(t)),$$
$$G^L_{i,j+1/2}(u(t)) = g(u_{ij}(t)), \quad G^L_{i,j-1/2}(u(t)) = g(u_{i,j-1}(t)).$$

Letting the function $\theta = 0$, yields the following ODE:

$$\frac{du_{ij}(t)}{dt} = -\frac{1}{\Delta x}(f(u_{ij}(t)) - f(u_{i-1,j}(t))) - \frac{1}{\Delta y}(g(u_{ij}(t)) - g(u_{i,j-1}(t)).$$

If the value of the flux limiter function is equal to 0, the flux-limiter scheme is equal to an upwind method. The downside of the upwind method is erroneous diffusion in numerical solutions. With a flux-limiter high resolution schemes can be chosen for $\theta \approx 1$ to avoid errors.

For a high resolution scheme a second order scheme could be chosen such that:

$$\begin{aligned} F^H_{i+1/2,j}(u(t)) &= \frac{f(u_{ij}(t)) + f(u_{i+1,j}(t))}{2}, \quad & F^H_{i-1/2,j}(u(t)) &= \frac{f(u_{i-1,j}(t)) + f(u_{ij}(t))}{2}, \\ G^H_{i,j+1/2}(u(t)) &= \frac{g(u_{ij}(t)) + g(u_{i,j+1}(t))}{2}, \quad & G^H_{i,j-1/2}(u(t)) &= \frac{g(u_{i,j-1}(t)) + g(u_{i,j}(t))}{2}. \end{aligned}$$

Letting the function $\theta = 1$, yields the following ODE:

$$\frac{du_{ij}(t)}{dt} = -\frac{1}{2\Delta x}(f(u_{i+1,j}(t)) - f(u_{i-1,j}(t))) - \frac{1}{2\Delta y}(g(u_{i,j+1}(t)) - g(u_{i,j-1}(t)).$$

In this case the flux-scheme is identical to the central difference scheme. A potential problem with using only central differences are divergent numerical solutions. To avoid having divergent numerical solution, we choose a flux-limiter that is not equal to 1 on the entire domain.

3 Time-Integration Methods

The Shallow Water Equations (SWE) contain derivatives with respect to both time and space. Discretizing the SWEs in the spatial direction yields the following initial value problem:

$$\frac{d\mathbf{V}(t)}{dt} = \mathbf{f}(t, \mathbf{V}(t)), \quad \mathbf{V}(t_0) = \mathbf{V}_0. \tag{3.1}$$

In this chapter, the objective is to find different numerical solutions of Equation (3.1). An analytic expression of a solution to this problem is (Vuik et al., 2018, p.67):

$$\mathbf{V}(t) = \mathbf{V}_0 + \int_{t_0}^{t} \mathbf{f}(s, \mathbf{V}(s))\, ds. \tag{3.2}$$

Since the right-hand side of (3.2) contains an integrand involving $\mathbf{V}(t)$, the problem in (3.2) is difficult to solve analytically. Therefore, the goal is to find numerical approximations for both (3.1) and (3.2). This is achieved by first discretizing the time domain. Let Δt denote the time step size, and let $t_n = t_0 + n\Delta t$. The value of $\mathbf{V}^n$ corresponds to the function $\mathbf{V}(t)$ evaluated at $t = t_n$. By considering (3.2), the values of $\mathbf{V}^{n+1}$ can be computed as follows (Vuik et al., 2018, p. 67):

$$\mathbf{V}^{n+1} = \mathbf{V}^n + \int_{t_n}^{t_{n+1}} \mathbf{f}(s, \mathbf{V}(s))\, ds. \tag{3.3}$$

Furthermore, Equation (3.1) can be numerically solved using a multi-step method (Stoer and Bulirsch, 1993, p. 455). In such a method, the values of $\mathbf{V}(t)$ at time step $t = t_{n+p} = t_0 + (n+p)\Delta t$ are computed based on previously computed values $\mathbf{V}(t_i)$, where $t = t_0 + i\Delta t$ and $n - j \leq i \leq n + p$. By considering Equation (3.2), a numerical solution can be obtained by approximating the integral in the following iteration (Stoer and Bulirsch, 1993, p. 455):

$$\mathbf{V}^{n+k} = \mathbf{V}^{n-j} + \int_{x_{n-j}}^{x_{n+k}} \mathbf{f}(s, \mathbf{V}(s))\, ds \tag{3.4}$$

Various methods can be used to approximate the integrals on the right-hand sides of (3.3) and (3.4). For explicit methods, the integral is approximated using $\mathbf{V}^n$. In multi-step methods, the integral is approximated using values from previous iterations of $\mathbf{V}$. However, for implicit methods, the integral in (3.2) depends on $\mathbf{V}^{n+1}$, requiring the solution of a system to compute $\mathbf{V}^{n+1}$ (Süli and Mayers, 2003, p. 324).

Various numerical methods can be used to solve multi-dimensional, time-dependent problems. Different time-integration methods have distinct properties in terms of stability, computational complexity, and error size. Therefore, comparing various time-integration schemes is necessary for identifying the most effective methods for SWEs.

The main objective of this chapter is to compare the performance of different time-integration techniques in solving a multi-dimensional, time-dependent ODE. The first section provides an overview of both explicit and implicit methods. The second and third sections briefly discuss key concepts related to time-integration schemes, such as stability and local truncation error, and provide an overview of Runge-Kutta schemes. Various methods are then compared. Finally, the last section explains the concept of semi-implicit methods and outlines their advantages.

3.1. Explicit and Implicit Methods

In this section, we show how different explicit and implicit time-integration schemes work. These schemes are found in the Julia Package DifferentialEquations.jl (Rackauckas and Nie, 2017).

The Forward Euler-method is a first order explicit method. This is the simplest method (Süli and Mayers, 2003) and is obtained by applying the left rectangle rule to the integral in (3.3) (Vuik et al., 2018, p.67):

$$\int_{t_n}^{t_{n+1}} \mathbf{f}(s, \mathbf{V}(s))\, ds \approx (t_{n+1} - t_n)\mathbf{f}(t_n, \mathbf{V}(t_n)). \tag{3.5}$$

Substituting (3.5) in equation (3.3) yields:

$$\mathbf{V}^{n+1} = \mathbf{V}^n + \Delta t \mathbf{f}(t_n, \mathbf{V}^n).$$

Likewise with many other integration rules, many different time-integration schemes can be derived for both explicit and implicit methods. Furthermore applying the right rectangle rule to equation (3.3) yields the Backward-Euler integration scheme:

$$\mathbf{V}^{n+1} = \mathbf{V}^n + \Delta t \mathbf{f}(t_{n+1}, \mathbf{V}^{n+1}).$$

This method is implicit since $\mathbf{V}^{n+1}$, appears on the right hand side.

Furthermore, by applying the the trapezoidal rule to the right hand side of (3.3), the second order trapezoidal method can be found:

$$\mathbf{V}^{n+1} = \mathbf{V}^n + \frac{\Delta t}{2}\left((\mathbf{f}(t_n, \mathbf{V}^n) + \mathbf{f}(t_{n+1}, \mathbf{V}^{n+1})\right).$$

This is a second order implicit method, and is also referred to as Crank-Nickelson (CN). An explicit version can be found by estimating $\mathbf{V}^{n+1}$ with $\mathbf{V}^n + \Delta t \mathbf{f}(t_n, \mathbf{V}^n)$, by substituting forward Euler. This method is called the Heun-scheme.

The second order Heun-scheme is:

$$\begin{aligned} k_1 = \mathbf{f}(t_n, \mathbf{V^n}), \quad k_2 &= \mathbf{f}(t_{n+1}, \mathbf{V}^n + \Delta t k_1) \\ \mathbf{V}^{n+1} &= \mathbf{V}^n + \frac{\Delta t}{2}(k_1 + k_2) \end{aligned}$$

Moreover another explicit scheme in the DifferentialEquations.jl, package is the second order Ralston scheme:

$$\begin{aligned} k_1 = \mathbf{f}(t_n, \mathbf{V^n}), \quad k_2 &= \mathbf{f}(t_n + \frac{3}{4}\Delta t, \mathbf{V}^n + \frac{3}{4}\Delta t k_1) \\ \mathbf{V}^{n+1} &= \mathbf{V}^n + \frac{\Delta t}{3}k_1 + \frac{2\Delta t}{3}k_2. \end{aligned}$$

3.1.1. Multistep Methods

In contrast to one-step methods, for multi-step methods a multi-step method approximation at time t_n depends on both previous time steps earlier than only the previous time step. By approximating the integrals at (3.4), various Adam-Bashforth methods can be derived.

The two step-second order Adam-Bashforth scheme:

$$\mathbf{V}^{n+2} = \mathbf{V}^{n+1} + \frac{3\Delta t}{2}\mathbf{F}(\mathbf{V}^{n+1}) - \frac{\Delta t}{2}\mathbf{F}(\mathbf{V}^{n}).$$

Note: there is no AB2 in differentialEquations.jl. However using a trick with the SplitODESolvers could be considered. By splitting an equation such that the stiff part is equal to 0, and choosing the "CNAB2"-option we can apply a second order Adams-Bashforth. Nevertheless the this option will also be seen as an implicit method. As a consequence using the CNAB2-method by setting the implicit part equal to 0, is very slow.

The third step-third order Adam-Bashforth scheme:

$$\mathbf{V}_{n+3} = \mathbf{V}_{n+2} + \frac{\Delta t}{12}(23\mathbf{F}(\mathbf{V}^{n+2}) - 16\mathbf{F}(\mathbf{V}^{n+1}) + 5\mathbf{F}(\mathbf{V}^{n}))$$

Note: This scheme is included in the DifferentialEquations.jl package.

Another very common time-integration scheme is the Leapfrog method. The leapfrog method is a multi-step method, that uses central differences to approximate the a derivative:

$$\mathbf{V}^{n+1} = \mathbf{V}^{n-1} + 2\Delta t\mathbf{F}(\mathbf{V}^{n}).$$

BDF-Methods The backward difference formula (BDF) methods form a family of implicit time-integration schemes that are also classified as multi-step methods.

In (L. Wang and Yu, 2020) a second-order BDF method is used for modeling fluid flows. At time t^{n+1} the numerical solution of Equation (3.1) is (L. Wang and Yu, 2020):

$$\mathbf{V}^{n+1} = \frac{4}{3}\mathbf{V}^{n} - \frac{1}{2}\mathbf{V}^{n-1} + \frac{2\Delta t}{3}\mathbf{f}(\mathbf{V}^{n+1}, t^{n+1}). \tag{3.6}$$

3.2. Local Truncation Error (LTE)

For comparing the accuracy of a numerical method, we need measures to determine the size of numerical errors. One way of measuring numerical errors we consider is the (local truncation error) LTE. The LTE is the difference between the numerical solution and exact solution at a specified step (Burden and Faires, 2011,p.276). The LTE at time step t_{n+1} is defined as:

$$\tau^{n+1} = \frac{\mathbf{x}^{n+1} - \mathbf{z}^{n+1}}{\Delta t}, \tag{3.7}$$

where x^{n+1} is the exact solution at time step $n+1$ and $\mathbf{z}^{n+1}$ is the numerical solution that is computed by applying one step of a numerical scheme to the exact solution $\mathbf{x}^{n}$.

For example we want to compute the LTE of forward Euler method that is being applied to solve a ODE of the form: $\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, t)$ with the exact solution: $\mathbf{x}(t)$. The numerical solution at t_{n+1} is equal to $\mathbf{w}^{n+1} = \mathbf{w}^{n} + \Delta t\mathbf{f}$. The LTE is:

$$\tau^{n+1} = \frac{\mathbf{x}(t_{n+1}) - (\mathbf{x}(t_n) + \Delta t\mathbf{f}(\mathbf{x}(t_n), t))}{\Delta t}.$$

By applying a Taylor expansion around $\mathbf{x}(t_n)$, we find:

$$\mathbf{x}(t_{n+1}) = \mathbf{x}(t_n) + \Delta t\frac{d\mathbf{x}}{dt}\bigg|_{t_n} + \mathcal{O}(\Delta t^2).$$

Substituting both the Taylor expansion and the expression of the ODE-system in the LTE yields:

$$\tau^{n+1} = \frac{\mathbf{x}(t_n) + \Delta t\mathbf{f}(\mathbf{x}(t_n), t_n) + \mathcal{O}(\Delta t^2) - (\mathbf{x}(t_n) + \Delta t\mathbf{f}(\mathbf{x}(t_n), t_n))}{\Delta t} = \mathcal{O}(\Delta t).$$

Hence the LTE of the forward Euler method is: $\mathcal{O}(\Delta t)$. The forward Euler method is a first order method. In general, a time-integration scheme is of order p, if the LTE is of order $\mathcal{O}(\Delta t^p)$.

LTE in Multi-step Methods

One can also compute the LTE for multi-step methods. For computing the values of $\mathbf{z}^{n+1}$ in equation (3.7), it is necessary have exact solutions at more than one previous time step.

For multi-step methods we have a numerical scheme of the form (Burden and Faires, 2011, p. 306):

$$\mathbf{w}^{n+1} = \sum_{i=0}^{k-1} \mathbf{w}^{n-i} a_{k-1-i} + \Delta t \sum_{i=-1}^{k-1} \mathbf{f}(t_{n-i}, \mathbf{w}^{n-i}) b_{k-1-i}.$$

for solving $\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, t)$. Therefore we can derive the LTE for multi-step methods by substituting:

$$\mathbf{z}^{n+1} = \sum_{i=0}^{k-1} \mathbf{x}^{n-i} a_{k-1-i} + \Delta t \sum_{i=-1}^{k-1} \mathbf{f}(t_{n-i}, \mathbf{x}^{n-i}) b_{k-1-i},$$

in equation (3.7).

For having numerical methods that converge to the exact solution if the time step goes to zero, we need both the consistency and the stability constrains, according to Lax's theorem (Vuik et al., 2018,p.75). We have consistency if the LTE converges to zero, if the time step goes to zero. Stability is necessary for global differences between the numerical solution and exact solution to not drastically increase.

3.3. Stability Analysis

It is important to know whether a numerical methods is numerically stable, since, knowing whether a method will converge or diverge is necessary. A method is numerically stable if the numerical errors do not increase. Often systems are described as stable if a small perturbation of a parameter or initial condition lead to big deviations in the final solutions.

3.3.1. Error Function and Stiffness

Examining the stiffness of the system that we are trying to solve, is necessary for examining stability. If we have a linear system, the equation (2.1) could be rewritten as:

$$\frac{d\mathbf{V}(t)}{dt} = A\mathbf{V}(t),$$

such that A is a matrix. This system has the following error equation (van Kan et al., 2019, p.107) :

$$\frac{d\varepsilon}{dt} = A\varepsilon(t). \tag{3.8}$$

For the nonlinear case with the following ODE:

$$\frac{d\mathbf{V}(t)}{dt} = f(\mathbf{V}(t)), \tag{3.9}$$

The Jacobian of $J = \nabla(\mathbf{f}(\mathbf{V}))_{\mathbf{V}}$ can be computed and the error function is:

$$\frac{d\varepsilon}{dt} = J\varepsilon$$

The analytic solution is of the form: $\varepsilon(t) \propto \exp{(At)}$. for the linear problem. For the nonlinear problem the error function is $\varepsilon(t) \propto \exp{(Jt)}$. Hence for stability of the analytical solution the real parts of the eigenvalues of the Jacobean need to be non positive.

Stiffness For stiffness for a linear system a large ratio between the largest and smallest eigenvalues of A exists (Süli and Mayers, 2003, p.345). Since the function $\mathbf{f}$ in equation (3.9) is nonlinear, the error function depends on the Jacobian. (Süli and Mayers, 2003, p.345). If the eigenvalues of the Jacobian have negative real parts and the ratios between the smallest and largest eigenvalues are large, the system is often regarded as stiff (Süli and Mayers, 2003, p.345).

3.3.2. Example: Stability for Linearized 2-Dimensional SWEs

For a 2-dimensional, simplified SWE, we have the following spatially discretized model from the previous chapter in subsection 2:

$$\frac{dh_{i,j}}{dt} = -D\frac{u_{i+1/2,j} - u_{i-1/2,j}}{\Delta x} - D\frac{v_{i,j+1/2} - v_{i,j-1/2}}{\Delta y}, \tag{3.10}$$

$$\frac{du_{i+1/2,j}}{dt} = -g\frac{h_{i+1,j} - h_{i,j}}{\Delta x}, \tag{3.11}$$

$$\frac{dv_{i,j+1/2}}{dt} = -g\frac{h_{i,j+1} - h_{i,j}}{\Delta y}. \tag{3.12}$$

Differentiating the equation in (3.10) with to time once and substituting the equations (3.11) and (3.12) yields the following:

$$\frac{dh_{i,j}}{dt} = gD\frac{h_{i-1,j} - 2h_{i,j} + h_{i+1,j}}{\Delta x^2} + gD\frac{h_{i,j+1} - 2h_{i,j} + h_{i,j-1}}{\Delta y^2}. \tag{3.13}$$

The right had side of equation 3.13 corresponds to a spatially discretized 2-dimensional Laplace operator. For $\Delta x = \Delta y = h$, the following eigenvalues can be found such that $\frac{d^2h_{i,j}}{dt^2} = \lambda^2 h_{i,j}$ (Vuik and Lahaye, 2019,p. 31):

$$\lambda_{k,l}^2 = \frac{-4Dg}{h^2}(1 - \frac{1}{2}\cos(\pi hk) - \frac{1}{2}\cos(\pi hl)), \Rightarrow$$

$$\lambda_{k,l} = \pm i\frac{2\sqrt{Dg}}{h}\sqrt{(1 - \frac{1}{2}\cos(\pi hk) - \frac{1}{2}\cos(\pi hl))}.$$

Furthermore the spectral radius of the problem is:

$$|\lambda_{k,l}| \leq 2\frac{\sqrt{2Dg}}{h}.$$

Since the eigenvalues of our problem are purely imaginary, the exact solutions are periodic. With no significant difference between the real components of the eigenvalues, this system does not exhibit stiffness in the traditional sense. Therefore, the equations in (3.10)-(3.12) are not stiff. However, the system may still present numerical challenges, particularly for explicit time-stepping methods, as small time steps may be required for having a stable numerical method.

Remark In case the SWEs are one-dimensional then $\frac{\partial}{\partial y} = 0$ and $v = 0$. The spectral radius of the one-dimensional problem is $\rho \leq \frac{\sqrt{2Dg}}{h}$.

3.3.3. CFL Condition

Another necessary, but not sufficient condition for numerical stability is the Courant-Friedrichs-Lewy (CFL) condition (Miller, 2007, p.10). This condition is satisfied if the numerical domain of dependence contains the analytical domain of dependence. To be able to compute a wave's amplitude accurately on a discretized spatial grid, the time step must be smaller than the time it takes for the wave to travel between adjacent grid points (CFD-Wiki, 2012). Hence, if one decreases the spatial grid size by a factor p, the upper limit for time step must also be decreased by a factor p.

The CFL-condition is thus a distinctive case of the eigenvalues of the Jacobian being in the stability region of the time-integration method.

For very simple one-dimensional SWEs the CFL condition is similar to: $\frac{\Delta t}{\Delta x}|c_{\max}| < K$, where $c_{\max}$ is a maximum speed and K depends on the numerical scheme. The characteristic wave-speeds of a one dimensional SWE-model is $c = U \pm \sqrt{gH}$,where H is the water depth and g gravitational acceleration.

A more general CFL-condition for the 2D-case is (CFD-Wiki, 2012):

$$\Delta t\left(\frac{u_x}{\Delta x} + \frac{v_y}{\Delta y}\right) < K,$$

where u_x and v_y are magnitudes of velocity in the x and y direction, respectively.

3.3.4. Stability Conditions

Usually other stability conditions that are part of a specific numerical scheme need to be satisfied. With for example a 1-dimensional linear test problem we can derive such conditions. A test problem is for example:

$$\frac{de(t)}{dt} = \lambda e(t), \lambda < 0.$$

Applying Euler Forward yields:

$$e^{n+1} = (1 + \Delta t\lambda)e^n,$$

To avoid having solutions that become very large over time we need to have the numerical condition: $|1 + \Delta t\lambda| < 1$. Hence restricting the size of the time step is necessary for the Euler forward method. In the next section different stability conditions corresponding to distinct numerical methods, are shown. In figure (2.1) and (2.2) different stability regions are shown.

For nonlinear systems the procedure for determining numerical stability changes slightly. If we have a nonlinear one dimensional ODE:

$$\frac{dx(t)}{dt} = f(t, x). \tag{3.14}$$

We apply linear approximation around a point $(t, x) = (t_1, x_1)$ and substitute in equation (3.14):

$$\frac{dx(t)}{dt} \approx f(t_1, x_1) + (x - x_1)f_x(t_1, x_1) + (t - t_1)f_t(t_1, x_1),$$

where we now an find the error function with $\lambda = f_x(t_1, x_1)$. For the Euler forward technique the numerical stability condition is $|1 + \Delta t f_x(t_1, x_1)| < 1$. Thus, for nonlinear problems approximations of the ODE can be used to check whether the numerical stability condition holds.

3.4. Runge-Kutta Methods

For time integration, Runge-Kutta (RK) methods are a category of methods that include implicit, explicit and methods with variable time steps. The Euler forward and backward methods are also Runge-Kutta methods.

3.4.1. Explicit Runge-Kutta methods

It is possible to approximate the integral in equation (3.3), by applying a quadrature scheme of the following form:

$$\mathbf{V}^{n+1} = \mathbf{V}^n + \Delta t \sum_{i=1}^{s} b_i \mathbf{f}(t^n + c_i\Delta t, \mathbf{V}(t_n + c_i\Delta t)),$$

where c_i and b_i are coefficients. Since we do not know the values of $\mathbf{V}(t^n + c_i\Delta t)$, approximations of $\mathbf{V}$ at a variety of times in the time-interval have to be made. Let k_i denote an approximation of $\mathbf{V}(t^n + c_i\Delta t)$. Also for every RK-method, $k_1 = \mathbf{f}(t^n, \mathbf{V}^n)$ and $c_1 = 0$ are defined. Henceforth every value of k_i is computed with the values of previous approximations. Hence, every explicit RK-scheme has the following structure (Rackauchas, 2020):

$$\begin{aligned} k_1 &= \mathbf{f}(t^n, \mathbf{V}^n), \\ k_2 &= \mathbf{f}(t^n + c_2\Delta t, \mathbf{V}^n + \Delta t a_{21}k_1), \\ k_3 &= \mathbf{f}(t^n + c_3\Delta t, \mathbf{V}^n + \Delta t(a_{31}k_1 + a_{32}k_2)), \\ &\vdots \\ \mathbf{V}^{n+1} &= \mathbf{V}^n + \Delta t(b_1k_1 + \ldots + b_sk_s). \end{aligned}$$

where s represents the number of stages. A stage is an approximation k_i. The a_{ij} are called coefficients. For RK-methods a tableau-notation can be used:

$$\begin{array}{c|cccc} 0 & & & & \\ c_2 & a_{21} & & & \\ c_3 & a_{31} & a_{32} & & \\ \dots & \dots & \dots & \dots & \\ \hline & b_1 & b_2 & \dots & b_s \end{array}$$

For deriving a RK-scheme with order-sizes of at most order four one find the right coefficients by applying Taylor-expansions around $(t^n, \mathbf{V}^n)$ for the different values of k_i. For a approximation with four stages: $k_i,\ i = 1, 2, 3, 4$, the following conditions have to be satisfied (Rackauchas, 2020) in order to have a fourth order method:

$$\sum_{i=1}^{4} b_i = 1, \quad \sum_{i=2}^{4} b_i c_i = \frac{1}{2}, \quad \sum_{i=2}^{4} b_i c_i^2 = \frac{1}{3}, \quad \sum_{i=2}^{4} b_i c_i^3 = \frac{1}{4},$$
$$b_3 a_{32} c_2 + b_4 a_{42} c_2 + b_4 a_{43} c_3 = \frac{1}{6}, \quad b_3 c_3 a_{32} c_2 + b_4 c_4 a_{42} c_2 + b_4 c_4 a_{43} c_3 = \frac{1}{8},$$
$$b_3 a_{32} c_2^2 + b_4 a_{42} c_2^2 + b_4 a_{43} c_3^2 = \frac{1}{12}, \quad b_4 a_{43} a_{32} c_2 = \frac{1}{24}.$$

One of the most well known RK-methods is the RK4-method. This method is given by the following scheme.

$$k_1 = \mathbf{f}(t_n, \mathbf{V^n}), \quad k_2 = \mathbf{f}(t_n + \frac{\Delta t}{2}, \mathbf{V}^n + \frac{\Delta t}{2} k_1)$$
$$k_3 = \mathbf{f}(t_n + \frac{\Delta t}{2} \mathbf{V}^n + \frac{\Delta t}{2} k_2), \quad k_4 = \mathbf{f}(t_{n+1}, \mathbf{V}^n + \Delta t k_3),$$
$$\mathbf{V}^{n+1} = V^n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4).$$

The tableau corresponding to RK4 is:

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

3.4.2. Implicit Runge-Kutta methods

Implicit RK-methods do exists as well. For implicit RK-methods the values of $k_1, ..., k_s$ depend on each other. An implicit RK-method has the following structure (Iserles, 2008):

$$k_j = \mathbf{V}^n + \Delta t \sum_{i=1}^{s} a_{ji} \mathbf{f}(t^n + c_i \Delta t, k_i), \quad j = 1, 2, \dots s,$$
$$\mathbf{V}^{n+1} = \mathbf{V}^n + \Delta t \sum_{j=1}^{s} b_j \mathbf{f}(t^n + c_j \Delta t, k_j).$$

Hence the tableau for an implicit RK method is of the form:

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ c_3 & a_{31} & a_{32} & \dots & a_{3s} \\ \dots & \dots & \dots & \dots & \\ \hline & b_1 & b_2 & \dots & b_s \end{array}$$

Since the Backward Euler from section 3.1 is also an implicit RK-method, the corresponding tableau is:

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$$

3.4.3. Adaptive time stepping

RK-methods with adaptive time stepping, change the time step per iteration based on an estimation of the LTE. For the estimation of the LTE, a higher and lower order method are used to compute approximate the errors.

With a lower order scheme an approximation is computed:

$$\mathbf{W}^{n+1} = \mathbf{V}^n + \Delta t \sum_{i=1}^{s} d_i k_i,$$

where $\mathbf{W}^{n+1}$ denotes the lower order approximtation. For the higher order approximation the same values of k_i are used, and hence the same values of a_{ij} and c_i.

$$\mathbf{V}^{n+1} = \mathbf{V}^n + \Delta t \sum_{i=1}^{s} b_i k_i.$$

Hence we can approximate the local error as follows (Iserles, 2008, p.108):

$$\frac{1}{\Delta t} E = ||\tau^{n+1}|| \approx \frac{1}{\Delta t} ||\mathbf{W}^{n+1} - \mathbf{V}^{n+1}|| = ||\sum_{i=1}^{s} (d_i - b_i) k_i||.$$

For every iteration of the time-integration methods the local error approximation is computed. If the error is above a certain threshold, a smaller time step is chosen. For every time step the following criterion could for example be (Iserles, 2008, p.108) :

$$E \leq \delta \Delta t,$$

where δ is an error threshold. If the criterion is not satisfied, one can choose a smaller step-size. However if the criterion is satisfied and is way smaller than the error-threshold, one could increase the step-size.

There exist many methods for changing the time step sizes per iteration. A very common method is the P-control or proportional controller (Rackauckas, 2020b). The time step size is changed in proportion to a error ratio. The error ratio is given by q (Rackauckas, 2020b) :

$$E_s = \left|\left| \frac{E}{max(V^n, V^{n+1})\tau_r + \tau_a} \right|\right|, \tag{3.15}$$

where E is the local error, τ_r is the relative tolerance and τ_a is the absolute tolerance. If $q < 1$, the error is less than the tolerance. The new value of Δt_{new} is computed by multiplying q with the time step size (Rackauckas, 2020b). The downside of the P-control is that the step-sizes can vary a lot. To smooth the variation in the step-sizes per for different iterations, a PI-control is often used (Rackauckas, 2020b). With a PI-control, the new step size is modified with the error from the previous step size (Rackauckas, 2020b).

An explicit RK-method with adaptive step-sizes has the following scheme of coefficients:

$$\begin{array}{c|cccc} 0 & & & & \\ c_2 & a_{21} & & & \\ c_3 & a_{31} & a_{32} & & \\ \dots & \dots & \dots & \dots & \\ \hline & b_1 & b_2 & \dots & b_s \\ & d_1 & d_2 & \dots & d_3 \end{array}$$

The most commonly used Runge-Kutta method is the Dormand-Prince (DP5) method Rackauchas, 2020. This method is particularly effective for large values of Δt, as it can maintain accuracy under a specified error threshold. The tableau for this method is as follows:

$$
\begin{array}{c|ccccccc}
0 & & & & & & & \\
\frac{1}{5} & \frac{1}{5} & & & & & & \\
\frac{3}{10} & \frac{3}{40} & \frac{9}{40} & & & & & \\
\frac{4}{5} & \frac{44}{45} & \frac{-56}{15} & \frac{32}{19} & & & & \\
\frac{8}{9} & \frac{19372}{6561} & \frac{-25360}{2187} & \frac{64448}{6561} & \frac{-212}{729} & & & \\
1 & \frac{9017}{3168} & \frac{-355}{33} & \frac{46732}{5247} & \frac{49}{176} & \frac{-5103}{18656} & & \\
1 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & \frac{-2187}{6784} & \frac{11}{84} & \\
\hline
 & \frac{35}{384} & 0 & \frac{500}{1113} & \frac{125}{192} & \frac{-2187}{6784} & \frac{11}{84} & 0 \\
 & \frac{5179}{57600} & 0 & \frac{7571}{16695} & \frac{393}{640} & \frac{-92097}{339200} & \frac{187}{2100} & \frac{1}{40}
\end{array}
$$

A big advantage of this method is that the coefficients a_{si} are equal to the constants b_i. The last derivative evaluation can thus be reused in the control step.

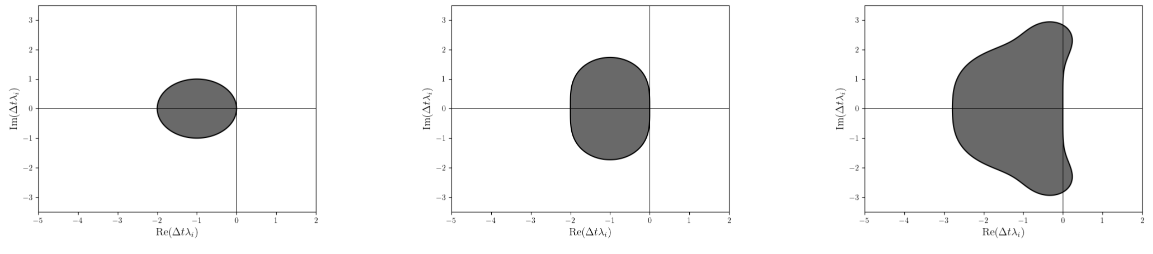
3.5. Comparing Different Methods

In this subsection the objective is to compare different explicit methods in terms of stability, error and computational complexity.

If our PDE is linear and $\mathbf{f} = A$, such that λ_i is an eigenvalue of A. We have the following stability conditions and errors for explicit methods (Vuik et al., 2018, p.79) :

Table 3.1: Stability conditions (Vuik et al., 2018, p.79).

method	Stability-condition	LTE
Forward Euler	$\lvert 1+\lambda_i\Delta t\rvert \leq 1$	$\mathcal{O}(\Delta t)$
Heun/midpoint	$\lvert 1+\lambda_i\Delta t+\frac{\Delta t^2}{2}\lambda_i^2\rvert \leq 1$	$\mathcal{O}(\Delta t^2)$
RK4-method	$\lvert 1+\lambda\Delta t+\frac{1}{2}(\lambda_i\Delta t)^2+\frac{1}{6}(\lambda_i\Delta t)^3+\frac{1}{24}(\lambda_i\Delta t)^4\rvert \leq 1$	$\mathcal{O}(\Delta t^4)$
Adams-Bashforth 2	$\lvert\frac{1}{2}+\frac{3}{4}\lambda_i\Delta t \pm \frac{1}{2}\sqrt{1+\lambda\Delta t+\frac{9}{4}(\lambda\Delta t)}\rvert \leq 1$	$\mathcal{O}(\Delta t^2)$
Leap-frog	$\lvert\lambda_i\Delta t \pm \sqrt{1+\lambda_i^2\Delta t}^2\rvert \leq 1$	$\mathcal{O}(\Delta t^2)$
Implicit Euler	$\frac{1}{1-\lambda_i\Delta t}$ < 1	$\mathcal{O}(\Delta t)$
CN	$\frac{1+\lambda_i\frac{\Delta t}{2}}{1-\lambda_i\frac{\Delta t}{2}}$ < 1	$\mathcal{O}(\Delta t^2)$

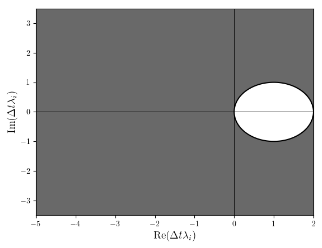


(a) Euler Forward

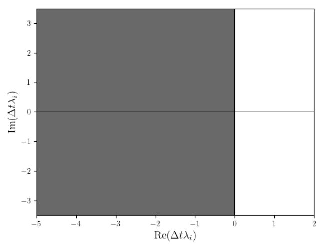
(b) Heun/Midpoint/Ralton

(c) RK4

Figure 3.1: Stability Regions



(a) Euler Backwards



(b) Trapezoidal method

Figure 3.2: Stability Regions

Comparing Various Explicit Methods

If all the eigenvalues are real and negative, the Adam-Bashforth has a smaller stability region compared to Heun scheme. However the Heun's scheme costs twice the amount of computations Compared to Adams-Bashford per time step. Hence, it might be useful to compare those two schemes. Forward Euler has a higher error, but is less computationally expensive. According to Safarzadeh Maleki and Khan, 2015, applying to second order Adams-Bashford for solving a one dimensional SWEs, with finite volumes, is the most efficient and accurate option compared to Heun and forward Euler. Hence comparing the second order Adams-Bashford with other explicit schemes could be a good idea.

Different explicit Adams Bashfort methods have different stability regions. In figure (3.3) those different stability regions can be seen (Grandclément and Novak, 2009). As the order of the error increases, the stability region decreases for different orders of Adams-Bashforth methods. Accordingly, investigating a second order Adam-Bashforth method could be beneficial since the larger stability region could enable researchers to choose larger time steps.

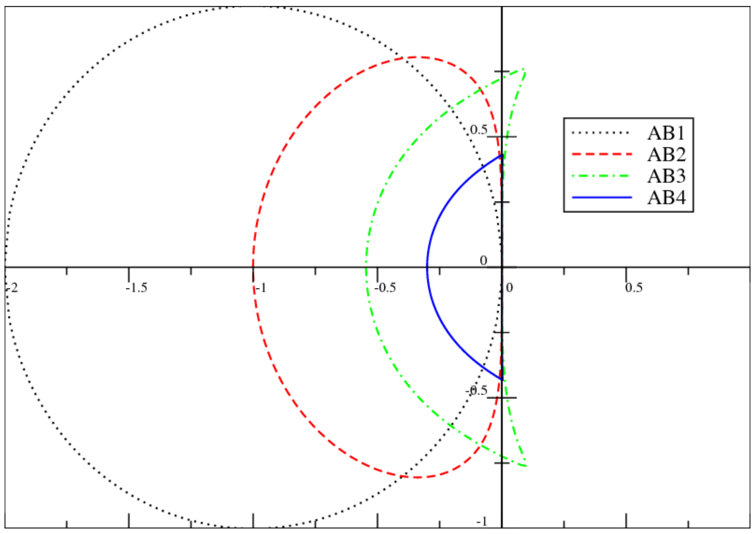


Figure 3.3: Stability regions Adams-Bashforth (Grandclément and Novak, 2009)

RK4 has a larger stability region compared to both the Euler-Forward and RK2 methods. However, RK4 is computationally more expensive than RK2, as it requires 4 stages per iteration. Additionally, due to the multiple stages in RK4, more memory is needed to store intermediate values.

In terms of computational complexity, the first-order Euler-Forward method requires only one evaluation of the function $\mathbf{f}$ per iteration. In contrast, second-order methods such as the midpoint, Heun, Ralston, and Adams-Bashforth schemes require two evaluations of $\mathbf{f}$. The third-order Adams-Bashforth scheme necessitates three function evaluations per iteration, while the fourth-order Runge-Kutta (RK4) method requires four evaluations. Therefore, the computational complexity per time step generally increases as higher-order numeric

Comparing Different Implicit Methods

Implicit methods have larger stability regions compared to explicit methods, as shown in Figure 3.2. Additionally, the CFL condition does not need to be satisfied. The Backward Euler method is a first-order method with a large stability region, and it can be applied even when the system has positive eigenvalues. In contrast, the trapezoidal method, which is second-order, has a smaller stability region. For the trapezoidal method, every eigenvalue of the time-dependent ODE must have a negative real part to ensure stability.

For each iteration, both the Euler Backward and the Trapezoidal method require solving a nonlinear system. Additionally, the Trapezoidal method requires one extra function evaluation. However, the extra computational cost of this function evaluation is negligible compared to the complexity of solving a nonlinear system. Therefore, when solving a system with negative eigenvalues, using the Trapezoidal scheme instead of the Euler Backward method reduces the error by one order, with only a minimal increase in computational complexity.

The BDF2 method is second-order accurate and features a large stability region. Similar to CN and BE methods, the BDF2 method is A-stable, meaning it remains stable for any time step size as long as all eigenvalues of $\mathbf{f}$ have non-positive real parts. This makes BDF2 well-suited for stiff systems and ensures numerical stability under such conditions. Note, if we select a BDF method with an order higher than 2, A-stability no longer holds. Therefore our study only includes the second-order BDF-method. The stability region of BDF2 is similar to that of BE, However the unstable circle on the right side of the complex plane is larger. Similar to the Backward Euler and trapezoidal methods, each BDF2 iteration requires solving a single system of equations. The BDF2 method is included in our study due to its larger stability region compared to the trapezoidal method and its higher order of accuracy compared to Backward Euler.

Memory Complexity

It may be useful to analyze the memory intensity of different time-integration schemes, especially since computations are performed on a GPU, where memory access is expensive compared to computations. While the GPU fetches a variable from memory, it can perform many floating-point operations (FLOPS) (Räss et al., 2023). Therefore, minimizing memory access is critical for maximizing computational efficiency on GPUs.

Furthermore, a useful memory throughput-based performance evaluation measure for testing different methods on a GPU is T_p in GB/s. To compute this value, we define A_f, which represents the amount of gigabytes accessed per iteration. The value of A_f is calculated as the sum of the number of values that are both read and modified per iteration, and once the number of values that are only read, multiplied by the floating-point precision (16, 32, 64, etc.) (Räss et al., 2023). The memory throughput, T_p, is then given by:

$$T_p = \frac{A_f}{t_i},$$

where t_i is the time per iteration.

Different explicit methods have varying memory complexities. For example, the memory complexity of the Euler Forward method is lower than that of RK4. In the RK4 method, four stages must be read and stored for each iteration, whereas in the Euler Forward method, only one stage needs to be read and stored. Therefore, one could expect the Euler Forward method to have better memory throughput per iteration.

In general, implicit time-integration methods are more memory-intensive, as they require solving a system of equations. Solving this system often necessitates storing a large matrix. A compromise between implicit and explicit methods to reduce matrix size is the use of semi-implicit or semi-explicit methods.

A semi-implicit time-integration method is a way to combine both implicit and explicit methods. According to Buwalda (2023), time splitting methods are an efficient compromise. In the next section, we give a examples of semi-implicit solvers. According to Goede (2019), time splitting methods with ”splitting into direction” are also often used. The ADI-method is used in Delft3D and Simona.

3.5.1. IMEX Solvers

In an implicit-explicit solver, the ODE problem is split as follows (Rackauckas and Nie, 2017):

$$\frac{d\mathbf{V}}{dt} = \mathbf{f}_1(t, \mathbf{V}(t)) + \mathbf{f}_2(t, \mathbf{V}(t)),$$

where $\mathbf{f}_1$ represents the stiff part and $\mathbf{f}_2$ is the non-stiff part. Using the `SplitODEProblem` function, one can solve a time-dependent ODE by integrating one part of the problem implicitly and the other explicitly.

The `SplitODEProblem` function provides methods such as `IMEXEuler` and `CNAB2`. For testing the accuracy of the splitting approach, the `SplitEuler` option is also available.

IMEXEuler

The `IMEXEuler` command applies both first-order explicit and implicit Euler methods (Rackauckas and Nie, 2017). At each time step, the following system must be solved:

$$\mathbf{V}^{n+1} - \Delta t \mathbf{f}_1(\mathbf{V}^{n+1}) = \mathbf{V}^n + \Delta t \mathbf{f}_2(\mathbf{V}^n).$$

CNAB2

The `CNAB2` option employs the Crank-Nicolson method to integrate the implicit components, while the second-order Adams-Bashforth method is used for explicit integration. At each time step, the following system must be solved:

$$\mathbf{V}^{n+1} - \Delta t \frac{1}{2} \mathbf{f}_1(\mathbf{V}^{n+1}) = \mathbf{V}^n + \frac{3\Delta t}{2} \mathbf{f_2}(\mathbf{V}^n) - \frac{\Delta t}{2} \mathbf{f_2}(\mathbf{V}^{n-1}) + \frac{\Delta t}{2} \mathbf{f}_1(\mathbf{V}^n).$$

The `CNAB2` method offers higher-order accuracy compared to IMEX-Euler. However, it may be more computationally expensive due to the increased number of floating-point operations required on the right-hand side of the equation.

Solving SWEs with Semi-Explicit & Semi-Implicit Methods

Different choices of IMEX schemes can be applied to various terms in Equations (2.7)-(2.9). In Buwalda et al., 2023, several IMEX schemes are compared, using backward and forward Euler methods for different splitting operators. Buwalda (2023) examines methods based on time-integration schemes by Silecki (1968), Hansen (1956), and Backhaus (1983). For very low water depths, he found the Hansen scheme to be the most efficient. When modeling SWEs with very low water depth, Buwalda observed that explicit schemes outperformed the semi-implicit scheme on a GPU. Additionally, for semi-implicit schemes, he investigated various solvers and found the repeated red-black (RRB) solver to be highly efficient.

Table 3.2 presents various IMEX schemes, along with their implicit and explicit components. In the Silecki scheme, the Coriolis component in the x-direction is integrated explicitly, while the y-direction component is integrated implicitly.

Table 3.2: Different IMEX Schemes.

IMEX-scheme	Implicit-terms	explicit-terms
Silecki, 1968	mass-conservation, bottom friction, Coriolis	Coriolis ,advection, pressure
Hansen, 1956	mass-conservation, bottom friction	Coriolis , advection, pressure
Backhaus, 1983	bottom-friction, pressure, mass-conservation	Coriolis, advection

3.5.2. The ADI-Method

Another very commonly used method for solving the SWEs using operator splitting is the alternating direction implicit (ADI) method. With the ADI-method a different time-integration technique is used. The main idea behind the ADI-method is that the discretizations in different spatial directions of a 2-dimensional PDE, in one time step the PDE is solved explicitly in the y-direction and implicitly in the x-direction, and for another time step the two schemes are swapped. Firstly, we have a problem with spatial discretization (van Kan et al., 2019, p. 115):

$$\frac{d\mathbf{V}}{dt} = (A_x + A_y)\mathbf{V} + \mathbf{F}^n,$$

such that A_x and A_y are matrices corresponding to spatial discretization in respectively the x and the y direction (Kan et. al.,2019) The ADI method is given by (van Kan et al., 2019, p. 115):

$$\begin{aligned}\bar{\mathbf{V}} &= \mathbf{V}^n + \frac{\Delta t}{2}\left(A_x\bar{\mathbf{V}} + A_y\mathbf{V}^n + \bar{\mathbf{F}}\right),\\ \mathbf{V}^{n+1} &= \bar{\mathbf{V}} + \frac{\Delta t}{2}\left(A_x\bar{\mathbf{V}} + A_y\mathbf{V}^{n+1} + \bar{\mathbf{F}}\right).\end{aligned}$$

That means the following linear system has to be solved (van Kan et al., 2019, p. 115):

$$(1 - \frac{\Delta t}{2}A_x)(1 - \frac{\Delta t}{2}A_y)\mathbf{V}^{n+1} = (1 + \frac{\Delta t}{2}A_x)(1 + \frac{\Delta t}{2}A_y)\mathbf{V}^n + \Delta t\bar{\mathbf{F}}.$$

Note that, if the matrices A_x and A_y are commutative, then the ADI-method is unconditionally stable.

The ADI-method is very often used for time integration on a CPU. The ADI-scheme is the time-integration scheme used for Simona.

ADI-Solvers

For the ADI-method different ways of solving a system are available such as cyclic reduction and The Thomas algorithm. According to research by the (De Goede, 2020) Red-Black Jacobi is the most efficient solver on a CPU.

On a GPU the ADI-method also shows excellent results in terms of performance. Research by (Aackermann et al., 2013) shows that the ADI-method with parallel cyclical reduction is efficient on a GPU.

4 Solving Nonlinear Systems in Implicit Time-Integration Schemes

After spatially discretizing the SWEs, a time-dependent ODE is obtained:

$$\frac{d\mathbf{V}}{dt} + \mathbf{f}(\mathbf{V}) = 0, \tag{4.1}$$

The simplest implicit numerical scheme to solve (4.1) is the Backwards Euler scheme, since it only has one stage with $\mathbf{f}(\mathbf{V}^{n+1})$. Using the implicit Euler scheme makes the solution more stable compared to an explicit Euler scheme. However finding a solution requires solving a system of equations. If $\mathbf{f}$ is a linear function, we only need to solve a linear system. However, in our thesis the equations are generally nonlinear. Applying the the backwards Euler time-integration scheme yields:

$$\mathbf{V}^{n+1} = \mathbf{V}^n + \Delta t \mathbf{f}(\mathbf{V}^{n+1}), \tag{4.2}$$

For computing the new values of $\mathbf{V}^{n+1}$, the root of the following equation needs to be found:

$$\mathbf{G}(\xi) = \xi - \mathbf{V}^n - \Delta t \mathbf{f}(\xi). \tag{4.3}$$

Thus applying an implicit time-integration scheme to a spatially discretized SWEs yields a nonlinear system of equations that has to be solved for every time iteration. It is therefore beneficial to examine methods for solving nonlinear systems. The main objective of this chapter is to present various techniques for solving nonlinear problems in the context of solving SWEs.

In section 4.1 an approach for linearizing the nonlinear systems using Newton-Raphson techniques is explained. After that, In section 4.2 a multigrid-based solver for linear problems is presented. Finally, in section 4.3 a pseudo-transient approach is discussed.

4.1. Solving Nonlinear Systems

For solving a system of nonlinear equations the Newton method can be employed. For the following system :

$$\mathbf{G}(\xi) = 0, \tag{4.4}$$

the objective is to derive approximations the root by applying an iterative scheme. The Taylor expansion of $\mathbf{G}$ around $\mathbf{x}$ is equal to:

$$\mathbf{0} = \mathbf{G}(\xi) = G(\mathbf{x}) + J_G(\mathbf{x})(\xi - \mathbf{x}) + \mathcal{O}(||\xi - \mathbf{x}||^2), \tag{4.5}$$

where for values of $\mathbf{x}$ close to ξ, the following linearization around root can be defined (van Kan et al., 2019, p.92):

$$J_G(\mathbf{x}^k)(\mathbf{x}^{k+1} - \mathbf{x}^k) = -G(\mathbf{x}^k), \tag{4.6}$$

where k is the k-th iteration and $J_G(\mathbf{x}^k)$ is the Jacobian of $\mathbf{G}$ evaluated in the point $\mathbf{x}^k$. For convergence an initial estimate $\mathbf{x}^0$ had to be chosen such that it is close enough to the exact root ξ. Furthermore we continue repeating the iterations from equation (4.6) till a stopping criterion is met. The stopping criterion used in (van Kan et al., 2019,p.92) is:

$$||\mathbf{G}(\mathbf{x}^k)|| < \epsilon,$$

where ϵ is a small number.

4.1.1. An Example of Newton's Method for Iterative Solution

To illustrate the idea of Newton iterations, the following problem is presented:

$$G(x) = \arctan v = 0,$$

where the analytical root is $v_{ex} = 0$. The initial guess is denoted by v^0. With an Newton's method an approximation of the stationary solution can be found. First the Jacobian of G is computed:

$$J_G(v) = \frac{1}{1+v^2}.$$

The Newton approximation for the n-th iteration based on equation (4.6) is:

$$v^{n+1} = v^n - (1 + (v^n)^2) \arctan v^n.$$

For the first four iterations with $v^0 = 20$, the following approximations can be found:

iteration	$\lvert v^n - v_{ex}\rvert \approx$
v^0	20
v^1	589.8560
v^2	545349.20
v^3	4.67162777e11
v^4	3.42812246e23

Table 4.1: Newton iterations, with $v^0 = 20$

In table 4.1, one can observe that for the first few iterations the method already diverges.

Only if we choose an initial guess close enough to the exact root v_{ex}, the Newton methods will converge. For example if $v^0 = 0.5$, the method satisfies the stopping criterion, with tolerance $\epsilon = 10^{-10}$, within 3 iterations:

iteration	$\lvert v^n - v_{ex}\rvert \approx$
v^0	0.5
v^1	0.079559511
v^2	0.000335302204
v^3	2.51314736e-11

Table 4.2: Newton iterations, with $v^0 = 0.5$

4.1.2. Applying the Newton-Raphson Method to Implicit Schemes

For working with an implicit time-integration method, we choose a Newton's method for solving the nonlinear equations. The implicit system in equation (4.3) is solved with the Newton iterations from equation (4.6). The following Newton-iterations are necessary for solving an implicit time step at $t = t^n$:

$$J_G(\mathbf{V}^{n+1,k})(\mathbf{V}^{n+1,k+1} - \mathbf{V}^{n+1,k}) = -\left(\mathbf{V}^{n+1,k} - \mathbf{V}^n - \Delta t \mathbf{f}(\mathbf{V}^{n+1,k})\right), \tag{4.7}$$

where the jacobian of $\mathbf{G}$, $J_G(\mathbf{V}^{n,k})$ is:

$$(I - \Delta t \mathbf{f}_{,\mathbf{V}}(\mathbf{V}^{n+1,k})),$$

where $\mathbf{V}^{n+1,k}$ is the k-th Newton iterand for approximating the root ξ from system 4.3.

The $\mathbf{V}^{n+1,k}$ should converge towards to root of the system for initial approximations that are close enough to ξ. We choose the following initial estimation: $\mathbf{V}^{n+1,0} = \mathbf{V}^{n}$. If a stopping criterion is met for the j-th iteration, we decide: $\mathbf{V}^{n+1} = \mathbf{V}^{n+1,j}$.

For small values of Δt, the system of equations will become diagonally dominant. For small values of Δt the condition number is smaller and iterative solvers will have faster convergence. The big disadvantage for choosing small values of Δt, is needing a large number of time iterations. However for larger values of Δt, an iterative solver night need more iterations for convergence. To reduce the number of equations that have to be solved, we could coarsen the grid. On the coarser grid, solutions can be found. With those solutions we continue to find solutions on the original finer grid.

4.2. Multigrid Methods

Multigrid techniques can both be used as a preconditioner or as a solver. A multigrid method preconditioner can be applied to Krylov subspace methods. When a multigrid is used as a solver, systems are solved on different grids with different levels of coarseness.

Solving on coarser-grids reduces the degrees of freedom, resulting in less computational complexity. Moreover it reduces smooth error-components, that are hard to get rid of on a fine grid. Furthermore, very often the number of iterations needed till convergence is smaller on a coarser grid, compared to finer grids. Multigrid methods make use of the solution on a coarser mesh, to faster find a solution on a finer grid.

The two exisiting types of multigrid methods are called geometric-and algebraic multigrid methods. Since a structured grid forms the basis for our study, we only consider the geometric multigrid method.

In this section we introduce the idea of a geometric multigrid, by presenting a an two-grid algorithm for solving a linear system, how to switch between different grids and we discuss the damped Jacobi solver.

4.2.1. Two-Grid Method

In this section solving a system on a fine grid with two different grids is explained. In this section the finer grid, denoted by Ω_h has n_x by n_y number of meshes in the x and y direction respectively. Each element is a $\Delta x \times \Delta y$-mesh. The coarsened variant of Ω_h, is denoted by Ω_H. The number of elements of Ω_H in the $x-$ and $y-$ direction are $\frac{n_x}{2}$ and $\frac{n_y}{2}$, respectively. The gridsize of an element on the coarse grid is: $\Delta x \times \Delta y$.

The system for Newton iterations given by (4.7) that we try to solve on fine grid and has the following structure:

$$A^h \mathbf{w}^h = \mathbf{g}^h, \tag{4.8}$$

such that we are trying to solve for $\mathbf{w}^h$ and we have

$$\begin{aligned} A^h = J_G(\mathbf{V}^{n+1,k}), \quad \mathbf{w}^h &= (\mathbf{V}^{n+1,k+1} - \mathbf{V}^{n+1,k}), \\ \mathbf{g}^h &= -\left(\mathbf{V}^{n+1,k} - \mathbf{V}^{n} - \Delta t \mathbf{f}(\mathbf{V}^{n+1,k})\right). \end{aligned}$$

The vector $\mathbf{w}^h$ on a staggered grid is a component vector of $\mathbf{u}^h$, $\mathbf{v}^h$ and $\mathbf{h}^h$-components. The vectors $\mathbf{u}^h$ and $\mathbf{v}^h$ are the velocity components in the x- and y- direction, respectively. The vector $\mathbf{h}^h$ represents water level elevation.

In Figure 4.1 the $\mathbf{h}^h$ is located at the centre of the mesh. Also, the vectors $\mathbf{u}^h$, $\mathbf{v}^h$ are located at the middle points on the x- and y-edges, respectively.

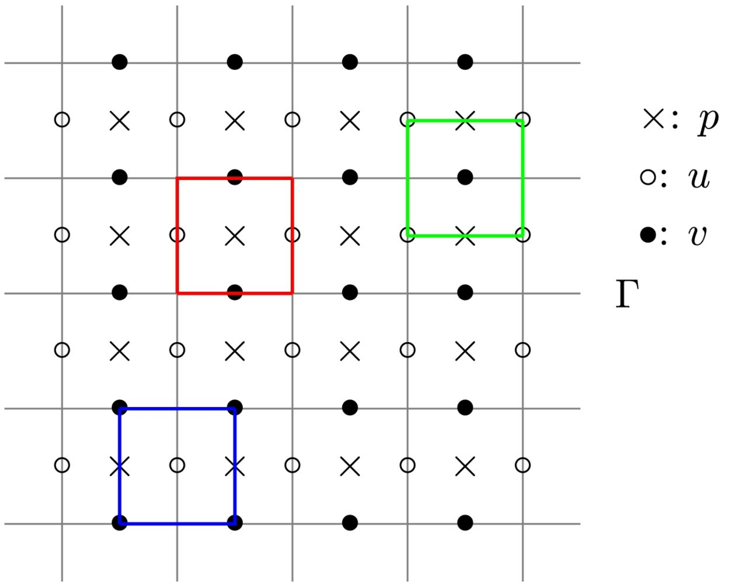


Figure 4.1: C-grid, (Luo et al., 2018)

Approximations on Coarse Grids For presenting the idea of a two-grid technique, general iteration based methods for finding approximations of solutions are introduced. Let $\mathbf{w}_n^h$ be the n-th approximation and $\mathbf{w}^h$ the exact solution. The error in the n-th iteration is (Trottenberg et al., 2001,p.35):

$$\mathbf{e}_n^h := \mathbf{w}^h - \mathbf{w}_n^h,$$

and a residual is defined as:

$$\mathbf{r}_n^h = A^h \mathbf{e}_n^h = \mathbf{g}^h - A^h \mathbf{w}_n^h,$$

and is equivalent with:

$$\mathbf{w}^h = \mathbf{w}_n^h + \mathbf{e}_n^h.$$

One can define a simplified operator of A^h such that the system can be solved easier. The approximation of A^h is given by $\tilde{A}^h$. The residual equation for an approximation $\tilde{\mathbf{e}}_n^h$ is:

$$\tilde{A}^h \tilde{\mathbf{e}}_n^h = \mathbf{r}_n^h,$$

such that a new approximation for $\mathbf{w}_{n+1}^h$, iteratively (Trottenberg et al., 2001,p.37):

$$\mathbf{w}_{n+1}^h = \mathbf{w}_n^h + \tilde{\mathbf{e}}_n^h.$$

Furthermore, it is also possible to solve residual equations for an approximation of A^h on a coarser grid. This coarse-grid approximation is given by $\mathbf{A}^H$. The residual equation in that case is:

$$A^H \mathbf{e}_n^H = \mathbf{r}_n^H,$$

and $\mathbf{e}_n^H$, $\mathbf{r}_n^H$ are the error and the residual on the course grid. Subsequently solve for $\mathbf{e}_n^H$ and transfer it to a fine grid error $\tilde{\mathbf{e}}_n^h$. Likewise a new approximation can be computed:

$$\mathbf{w}_{n+1}^h = \mathbf{w}_n^h + \tilde{\mathbf{e}}_n^h.$$

Note that the operators for transferring vectors between different grids are denoted as I_H^h and I_h^H. In Section 4.2.2 a more detailed explanation can be found.

Two-grid Cycle For the defined problem, we employ a two-grid method. To solve the linear system using this approach, the system is coarsened once and then solved on the coarse grid. The 2-level coarsening procedure is illustrated in Figure 4.2 (Nikolopoulos et al., 2024).

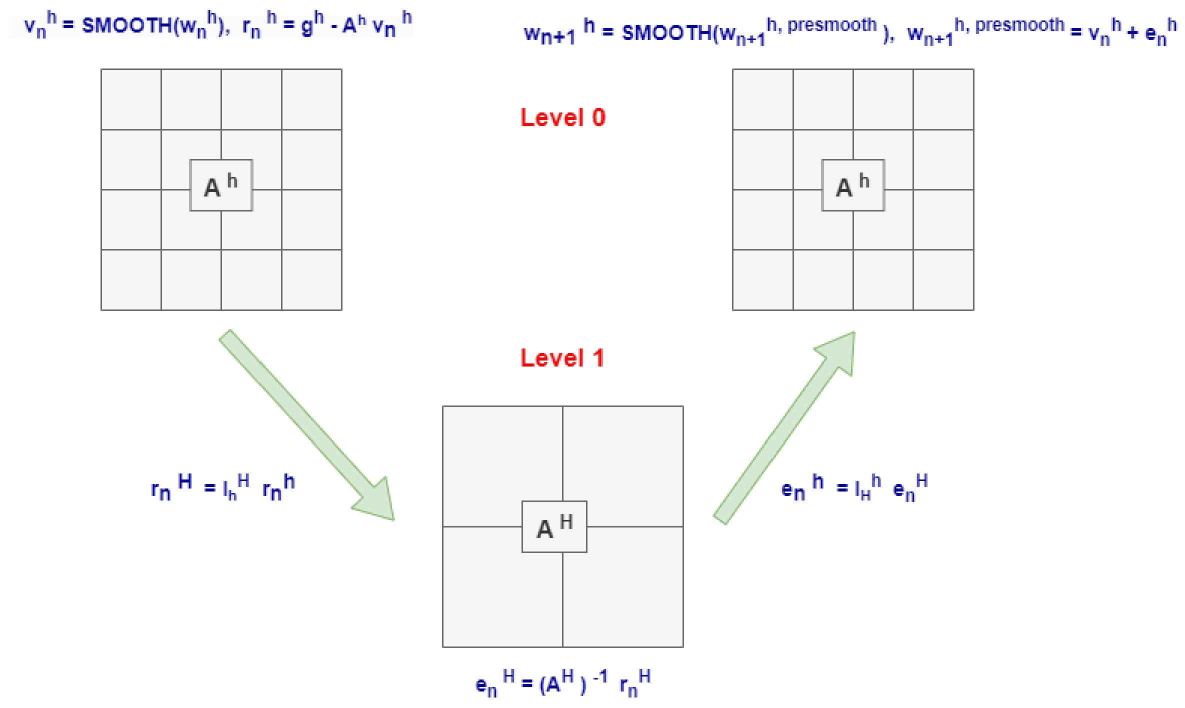

v_n^h = SMOOTH(w_n^h), $r_n^h = g^h - A^h v_n^h$
w_{n+1}^h = SMOOTH($w_{n+1}^{h,\text{ presmooth}}$), $w_{n+1}^{h,\text{ presmooth}} = v_n^h + e_n^h$
Level 0
A^h
A^h
Level 1
$r_n^H = I_h^H r_n^h$
A^H
$e_n^h = I_H^h e_n^H$
$e_n^H = (A^H)^{-1} r_n^H$

Figure 4.2: 2-level multigrid overview

For solving a system of the form (4.8), first a pre-smoother is applied (Briggs et al., 2000,p.37). For a pre-smoother we can apply ν_1 iterations of a basic iterative method (BIM). Secondly, we compute the residual on the fine grid and restrict the residual to the coarser grid (Briggs et al., 2000,p.37). On the coarser grid, we solve the system (4.9). Note, If we have a very large number of grid-points, solving on a coarse system is still very computationally expensive. The error on the coarse grid is transferred to the fine grid and added to the the solution (Briggs et al., 2000,p.37):

$$\mathbf{w}_{n+1}^h = \mathbf{w}_n^h + I_H^h \mathbf{e}_n^H.$$

Finally, a post-smoothing procedure is realized (Briggs et al., 2000,p.37). For post-smoothing, a BIM with ν_2 and with $\mathbf{w}^h$ as initial guess can be selected.

4.2.2. Intergrid Transfer Operators for Staggered Grids

In the previous sections, we switched between different grids. In this subsection, inter-grid transfer operators for relocating vectors between finer and coarser meshes are presented.

Restriction The restriction operator I_h^H transfers the fine grid in a staggered way to a coarse grid in a staggered way. In Figure 4.4 a fine staggered gird and coarse staggered grid are shown (Trottenberg et al., 2001, p.319). For approximating $\mathbf{w}^h$ on the coarse grid, we use mean values of neighboring points (Trottenberg et al., 2001, p.318):

$$u^H = \frac{1}{2}\begin{bmatrix}1\\ \cdot\\ 1\end{bmatrix} u^h = I^H_{u,h}u^h,$$
$$v^H = \frac{1}{2}\begin{bmatrix}1 & \cdot & 1\end{bmatrix} v^h = I^H_{v,h}v^h,$$
$$p^h = \frac{1}{4}\begin{bmatrix}1 & & 1\\ & \cdot & \\ 1 & & 1\end{bmatrix} p^h = I^H_{p,h}p^h,$$

where $I^H_{*,h}$, are the restriction operators for different variables.

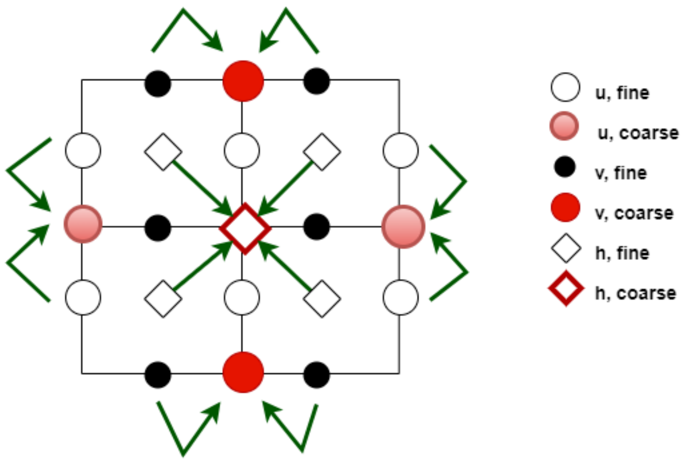


Figure 4.3: Coarse and Fine C-grids, Restriction from (Trottenberg et al., 2001,p.318).

The values for the coarse vector can also be approximated by selecting a weighted average of six adjacent points. Luo et al. (2018) presents the following restriction operators for staggered grids in stencil notation:

$$I^H_{u,h} = \frac{1}{8}\begin{bmatrix}1 & 2 & 1\\ & \cdot & \\ 1 & 2 & 1\end{bmatrix}, \quad I^H_{v,h} = \frac{1}{8}\begin{bmatrix}1 & & 1\\ 2 & \cdot & 2\\ 1 & & 1\end{bmatrix}, \quad I^H_{p,h} = \frac{1}{4}\begin{bmatrix}1 & & 1\\ & \cdot & \\ 1 & & 1\end{bmatrix}.$$

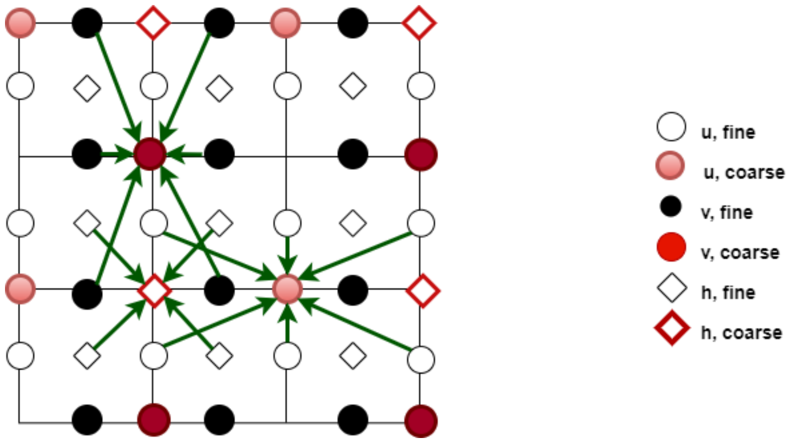


Figure 4.4: Coarse and fine C-grids, Restriction from (Luo et al., 2018).

Interpolation The interpolation transfer operator changes vectors on coarse grid to vectors on finer grid. The interpolation operator is given by I_H^h. In research by (Luo et al., 2018) the interpolation operator is chosen to be the adjoint of the restriction operator.

Galerkin condition For solving a coarse-grid problem the following notation is used:

$$A^H \mathbf{e}^H = \mathbf{r}^H \tag{4.9}$$

where $\mathbf{e}^H$ is the error and $\mathbf{r}^H$ is the residual on a coarse grid. with the interpolation and coarsening operators the vectors transferred between different grids. A coarse matrix system A^H needs to be derived to solve the problem on a coarse grid. To see how the coarse matrix is constructed, we can take the following steps:

$$A^H \mathbf{e}^H = \mathbf{r}^H = I_h^H \mathbf{r}^h = I_h^H A^h \mathbf{e}^h = I_h^H A^h I_H^h \mathbf{e}^H,$$

Hence the so- called Galerkin coarsening operator is (Trottenberg et al., 2001, p.42):

$$A^h = I_h^H A^h I_H^h.$$

Since we now have a well-defined way for transferring linear systems between fine grids, we can develop a correction scheme for two-grids.

Rediscretization The problem A^H on the coarse grid that is equivalent with the fine grid problem A^h can be also be acquired with a rediscretization approach (Briggs et al., 2000,p. 37). With a rediscretization technique the matrix A^H on a coarse mesh is found, by directly discretizing systems of equations on the coarse discrete domain Ω^H.

4.2.3. Damped Jacobi Solver & Smoother

The damped Jacobi solver can be used to solve a system as a basic iterative method (BIM) on a coarse grid. Also, the errors of a problem can be made smoother with a few damped Jacobi iterations.

Damped Jacobi Iterations The basic idea of an iterative method is solving a linear system of equations:

$$A\mathbf{u} = \mathbf{f},$$

with iterative steps. Let $\mathbf{u}^0$ be an initial guess and let $\mathbf{u}$ denote the exact solution. The error and residual after k iterations are defined by $\mathbf{e}^k = \mathbf{u} - \mathbf{u^k}$ and $\mathbf{r}^k = \mathbf{f} - A\mathbf{u}^k$, respectively. The matrix A can be split up as follows:

$$A = D - L - U,$$

where D is diag(A), $-L$ is the strictly lower triangular part of A and $-U$ is the strictly upper triangular part of A. The linear system can be rewritten as:

$$D\mathbf{u} = (L + U)\mathbf{u} + \mathbf{f}.$$

With this an iterative Jacobi method can be developed such that :

$$\mathbf{u}^{k+1} = D^{-1}(L + U)\mathbf{u}^k + D^{-1}\mathbf{f}.$$

Moreover the a damped Jacobi scheme is defined as:

$$\mathbf{u}^{k+1} = ((1 - \omega)I + \omega D^{-1}(L + U))\mathbf{u}^k + \omega D^{-1}\mathbf{f}, \tag{4.10}$$

where ω is the relaxation parameter. The damped Jacobi scheme is a weighted average between a previous iteration and a current Jacobi iteration. When $\omega = 1$, the damped Jacobi scheme is identical to the original Jacobi method. When using this method as a solver for a linear system, one continues to repeat the iterations till a stopping criterion is met. The following stopping criterion can be used:

$$\frac{||\mathbf{r}^k||}{||\mathbf{f}||} \leq \epsilon,$$

where we can choose small values for ϵ (Vuik and Lahaye, 2019, p.82).

From equation (4.10), one can deduce that the iteration matrix is $B = (I - \omega D^{-1}A)$. The error of the problem changes per iteration is :

$$\mathbf{e}^{k+1} = B\mathbf{e}^k.$$

Hence for convergence we need $||B|| < 1$ and also the spectrum of B needs to be smaller than one (Briggs et al., 2000, 17). When $\rho(B) << 1$, the BIM converges to a solution within very few iterations.

Smoothing Errors with Damped Jacobi If the problem matrix A is diagonalizable, the matrix can be rewritten as matrix product with an $A = P^{-1}DP$. The matrix D is a diagonal matrix with eigenvalues on the diagonals and P is a matrix with eigenvectors on the columns. The eigenmodes are either low or high frequency modes. The low frequency modes are grid vectors, that are linear combinations of eigenvectors corresponding to low eigenvalues (Briggs et al., 2000, p.19). The high frequency modes are grid vectors, that are linear combinations of eigenvectors corresponding to high eigenvalues (Briggs et al., 2000, p.19).

The error $\mathbf{e}^k$ can be written as a linear combination of the eigenvectors of A:

$$\mathbf{e}^k = \sum_{j=1}^{n-1} c_j \mathbf{w}_j^k,$$

where $\mathbf{w}_j$ are the eigenvectors of A and c_j are a few weights.

For the Jacobi-iteration this can be rewritten as :

$$\mathbf{e}^k = B^k\mathbf{e}^0 = \sum_{j=1}^{n-1} \lambda_j^k(B) c_j \mathbf{w}_j^0.$$

Since for damped Jacobi the eigenvalues of B and A are linked to each other such that $\lambda(B) = 1 - \frac{\omega}{2}\lambda(A)$ (Briggs et al., 2000, p.17), we have :

$$\mathbf{e}^k = \sum_{j=1}^{n-1} (1 - \frac{\omega}{2}\lambda_j(A))^k c_j \mathbf{w}_j^0. \tag{4.11}$$

From (4.11), one can observe that the damped Jacobi method decreases the high frequency errors the most. Hence the damped Jacobi method acts as a smoother.

The damped Jacobi scheme is very effective for removing the high frequency components of the error. In contrast the low error components decrease much slower, with a damping factor very close to 1. If the error vector is transferred to a coarser grid, the lower frequency components are downplayed while the high frequency components do not disappear. Hence the multigrid and Damped Jacobi method is a good combination. The two-gridmethod uses a combination of multigrid and a BIM.

Example Jacobi iterations We can also multiply Problem (4.3) by $\frac{1}{\Delta t}$ and apply Newton Raphson. In that case, the problem $A^h = \frac{1}{\Delta t} - \mathbf{f}_{,\mathbf{V}}$ is presented, such that $\mathbf{f}_{,\mathbf{V}}$, corresponds to the Jacobian of the problem in section 3.3.2. For this particular case we know that $\mathbf{f}_{,\mathbf{V}}$ has only zeros on the diagonal. Also on the diagonal all the matrix-entries are equal to $\frac{1}{\Delta t}$. Hence the iteration matrix in our problem is: $B = (I - \omega\Delta t(\frac{1}{\Delta t} - \mathbf{f}_{,\mathbf{V}}))$. The eigenvalues of B are equal to $1 - \omega(1 - \Delta t i\lambda_I)$, where $(i\lambda_I)$ are the purely imaginary eigenvalues of the Jacobian. The following condition for convergence needs to be satisfied.

$$|\rho(B)|^2 = (1-\omega)^2 + \omega^2 \Delta t^2 |\lambda_I|^2 < 1,$$

where for a one-dimensional problem we know:

$$|\lambda_I| \leq \frac{\sqrt{2gD}}{\Delta x}$$

Finally a condition for convergence is:

$$\Delta t \frac{\sqrt{2gD}}{\Delta x} < \sqrt{\frac{2-\omega}{\omega}}.$$

For $\omega = 1$, the convergence condition for the BIM is even more restrictive than the CFL-condition. Hence, we need to consider working with smaller values of ω.

Alternatives in the previous example, the damped Jacobi had poor convergence. However there are alternative iterative solvers. For example the relaxed Gauss Seidel method or Gauss-Seidel red black often have better convergence properties.

4.3. Pseudo-time-stepping

In the previous section we solve a nonlinear system with a Newton's method and a two-level multigrid. The Newton's method converges with a quadratic rate, if an initial guess is close enough to the actual solution (van Kan et al., 2019,p.94). The initial guess for ξ in equation (4.3) is chosen to equal to the value of $\mathbf{V}^n$, which is the solution from the previous time iteration. If the time steps are large and the difference : $\mathbf{V}^{n+1} - \mathbf{V}^n$ is large, the convergence of of the Newton's method is bad.

By applying Newton's method one still needs to still solve a linear system. This linear system could be solved with a BIM. in the example in section 4.2 one can find that the damped Jacobi method on a fine grid only converges if the w is very small or if a restriction to Δt is applied that is much stricter than the CFL-condition. With a multigrid method the computational cost of solving of applying damped Jacobi could be reduced again.

pseudo-time-stepping is often applied to implicit methods. In general pseudo-time-stepping methods are used for finding steady state solutions of a differential equation (Zandbergen et al., 2023). A pseudo-time-stepping method is also often called a pseudo-transient method.

4.3.1. Steady-state solution

By applying pseudo-transient methods, we can find steady state solution for an ODE. A pseudo-time-stepping solution is time independent. We present the following ODE:

$$\frac{d\mathbf{v}}{d\tau} = -\alpha \mathbf{G}(\mathbf{v}), \tag{4.12}$$

where α is chosen as $\frac{\beta}{\Delta t}$ and G is similar to equation (4.3). The value of β is a dimensionless constant. The Jacobian of $-\alpha\mathbf{G}(\mathbf{v})$ is:

$$\frac{\beta}{\Delta t}\left(\Delta t \mathbf{f}_{,\mathbf{V}}(\mathbf{V}) - I\right)$$

and the eigenvalues of this Jacobian are:

$$\mu = \frac{\beta}{\Delta t}(\Delta t \lambda - 1),$$

where λ is an eigenvalue of $\mathbf{f}_{,\mathbf{V}}(\mathbf{V})$. Thus this ODE is stable if the original system is stable. System (4.12) contains a damping term i.e. had eigenvalues with a negative component and is slightly more stable, which can be advantageous for oscillating systems. In the example in Section 3.3.2, the eigenvalues are purely imaginary. In this case, the eigenvalue μ has a negative real part, leading to faster convergence.

Using a time-integration scheme, we can discretize Equation (4.12) in the pseudo-time (τ) direction. After several pseudo-time iterations, a steady state is reached, which corresponds to the solution for the next outer time iteration.

For example, with a Forward Euler method we can find a steady state solution with the following iterations:

$$\mathbf{v}^{k+1} = \mathbf{v}^k - \Delta\tau\alpha\mathbf{G}(\mathbf{v}^k) = \mathbf{H}(\mathbf{v}^k).$$

The eigenvalues of the Jacobian of $\mathbf{H}(\mathbf{v}^k)$ are:

$$1 + \Delta\tau\mu = 1 + \Delta\tau\alpha(\Delta t\lambda - 1),$$

and a necessary condition for convergence is :

$$|1 + \Delta\tau\alpha(\Delta t\lambda - 1)| < 1.$$

For the problem for 3.3.2 the Jacobian of $\mathbf{f}$ has the following eigenvalues:

$$\lambda_{kl} = i\frac{2\sqrt{Dg}}{h}\sqrt{(1 - \frac{1}{2}\cos(\pi hk) - \frac{1}{2}\cos(\pi hl)}, \tag{4.13}$$

where the eigenvalue with the largest complex value is : $\lambda_{max} = i2\sqrt{2Dg}/h$. Hence the stability condition for applying Forward Euler to the pseudo-time ode is:

$$\left(1 - \frac{\Delta\tau\beta}{\Delta t}\right)^2 + \left(\Delta\tau\beta\frac{2\sqrt{2}\sqrt{Dg}}{h}\right)^2 < 1, \tag{4.14}$$

and for the time step the following stability condition can be found:

$$-\frac{h}{8D\beta g\Delta\tau}\left(-h + \sqrt{h^2 - 8\Delta\tau^2\beta^2 Dg}\right) < \Delta t < \frac{h}{8D\beta g\Delta\tau}\left(h + \sqrt{h^2 - 8\Delta\tau^2\beta^2 Dg}\right),$$

and also

$$\frac{8\Delta\tau^2\beta^2 Dg}{h^2} < 1.$$

There still exists a stability condition for the time step for solving a implicit method with a pseudo-time-stepping scheme. For $\Delta\tau$ multiplied with β, the CFL-condition still holds.

In order to reduce the upper limit for Δt, it is possible to select smaller values for $\Delta\tau$. A possible trade-off could be, that one might need to compute a larger number of pseudo-time-iterations, when the values of $\Delta\tau$ are small.

Multilevel methods can be useful for reducing the number of pseudo-time iterations. On a coarse grid, a larger inner time step could be used and faster convergence to a steady state could be realized. A steady-state solution can be obtained on the coarse grid, requiring fewer computations. This coarse grid solution could then be used as an approximation for a finer grid solution.

In comparison, if we were to solve the same problem with a Forward Euler scheme to the real time, we have the following stability condition (based on section 3.5, using ODE from section 3.3.2):

$$1 + \Delta t^2\frac{8Dg}{h^2} < 1.$$

This stability condition is never satisfied.

5 GPU Computing

Before 2003 microprocessors with a single central processing unit (CPU), have shown very fast increases in performance and decreases in cost for decades (Kirk and Hwu, 2017). However, the rise in clock frequency slowed down after 2003, due to issues with heat dissipation (Kirk and Hwu, 2017). As a result the growth in processing power on a single CPU slowed down. To continue increasing the number of flops per second, microprocessor manufacturers have made the decision produce microprocessors with multiple CPU-cores that perform tasks concurrently. Subsequently, computational programs need to be parallel to continue experiencing increases in performance. For these parallel programs multiple threads executing different parts of a program are mapped to different cores. For instance, the Intel® Core™ i9 processor 14900KS features a hybrid architecture with 8 performance cores (P-cores) and 16 efficient cores (E-cores), supporting a total of 32 threads (Intel, 2024a). The P-cores utilize hyper-threading, allowing each core to handle two threads concurrently, which boosts the performance of certain sequential or lightly threaded workloads.

The primary motivation for increasing the number of threads in a processor is to enhance its ability to execute multiple instructions simultaneously. Graphics processing units (GPUs) excel at this by supporting a massively parallel architecture. Even in 2016, GPUs, such as described in (Kirk and Hwu, 2017), operated with tens of thousands of threads.

In 2016, the performance ratio in terms of FLOPS between GPUs and multi-core CPUs was reported to be around 10 (Kirk and Hwu, 2017). However, for modern GPUs, such as the NVIDIA Hopper series, this ratio can be much higher. For example, the Intel® Xeon® Platinum 8470Q processor, a contemporary CPU, delivers a performance of 2.3296 TFLOPS (Intel, 2024b). In comparison, the NVIDIA H100 PCIe 80 GB GPU (Techpowerup, 2023) achieves a theoretical performance of 51.22 TFLOPS in single precision.

5.1. GPU Architecture

The differing performance between a CPU and a GPU can be attributed to their fundamentally distinct architectures. In Figure 5.1 one can observe the differences in terms of architecture between a GPU and CPU with multiple cores (Nvidia, 2024). The CPUs has much larger caches and thus larger memory. in contrast, The GPUs have a much larger number of cores. The CPU architecture is designed to optimize the performance of serial, non-parallel programs (Kirk and Hwu, 2017). With advanced control logic, a CPU can execute sequential instructions in parallel on a single thread (Kirk and Hwu, 2017). Moreover, a CPU features larger cache memories, which help reduce memory access latency in memory-intensive programs (Kirk and Hwu, 2017). However, for massively parallel algorithms that are computationally intensive, having massively parallel GPU-hardware is more beneficial compared to CPU with control logic and cache memories. If an algorithm can be divided into many tasks and performed by many threads in parallel on many different cores, running it on the large numbers of GPU-cores will result in a massive speedup.Thus, a GPU utilizes more transistors for data processing compared to a CPU (Nvidia, 2024). On a GPU, data access latencies can be hidden by performing concurrent computations, allowing for more efficient use of computational resources. (Nvidia, 2024). On a CPU, however, using large caches and complex control flows to reduce memory access latency requires a significant number of transistors(Nvidia, 2024).

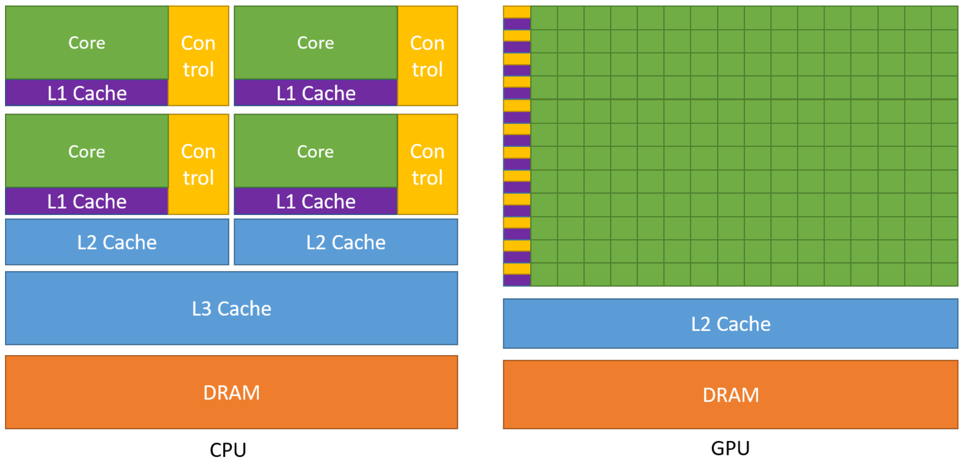


Figure 5.1: Different Designs for GPU and CPU (Nvidia, 2024)

A GPU features a SIMD (Single Instruction, Multiple Data) architecture (Vuik and Lemmens, 2018). In a SIMD-style architecture, multiple computation units execute the same instructions on different data elements, often in parallel (Maitre, 2013). Hence, a GPU, with its large number of processors, can perform computations where each core independently executes the same operations in parallel (Vuik and Lemmens, 2018). For example, if one GPU core has a clock rate of 1 GHz, then with a thousand cores working perfectly in parallel, a teraflop of compute power can be achieved (Vuik and Lemmens, 2018). A GPU node is often referred to as a device, and each device has a fixed number of processors. A processor on a device is called a streaming multiprocessor (SM). Each multiprocessor contains several streaming processors (SPs), and these SPs share control logic and an instruction cache within each SM(Kirk and Hwu, 2017). In figure 5.2 an general overview of a GPU device is given (Vuik and Lemmens, 2018).

A GPU is typically mounted on an internal board, and it is connected to the main memory via a PCIe (Peripheral Component Interconnect Express) connector (Maitre, 2013). Communication via the PCIe bus can be relatively slow compared to the high-bandwidth connections within the GPU itself, as the PCIe bus introduces latency and bandwidth limitations when transferring data between the GPU and the main memory. The NVIDIA A100 architecture incorporates PCIe gen 4 and NVIDIA H100 has PCIe gen 5 (Andersch et al., 2022) with 16 lanes. As a result the PCI-busses on H100 has bandwidth of 128 GB/s, compared to 64 GB/s for the PCI-busses on an A100 (Andersch et al., 2022). Many GPU versions also support NVLINK (Kirk and Hwu, 2017), which is a CPU–GPU and GPU–GPU interconnect.

On an NVIDIA GPU, the SMs are connected to an L2 cache and DRAM, which provides high memory bandwidth. Computations are performed on the SMs, and memory is accessed from the DRAM through the L2 cache. The global memory is stored in the DRAM.

Another component commonly found in GPU nodes is the tensor processing unit (TPU). TPUs are specialized hardware designed for accelerating tensor operations, which refer to high-dimensional linear algebra operations (Rackauckas, 2020a). To perform these operations, TPUs typically utilize BFloat16 data types (Rackauckas, 2020a). However, this reduced precision can lead to issues such as catastrophic cancellation in certain computations (Rackauckas, 2020a). In our study, we opted not to use TPUs, as the precision provided by BFloat16 is insufficient for our requirements.

5.1.1. Blocks, Warps and threads

The physical GPU architecture, encompassing devices, streaming multiprocessors (SMs), and cores, can be represented through a software layer (Vuik and Lemmens, 2018). Threads are organized into

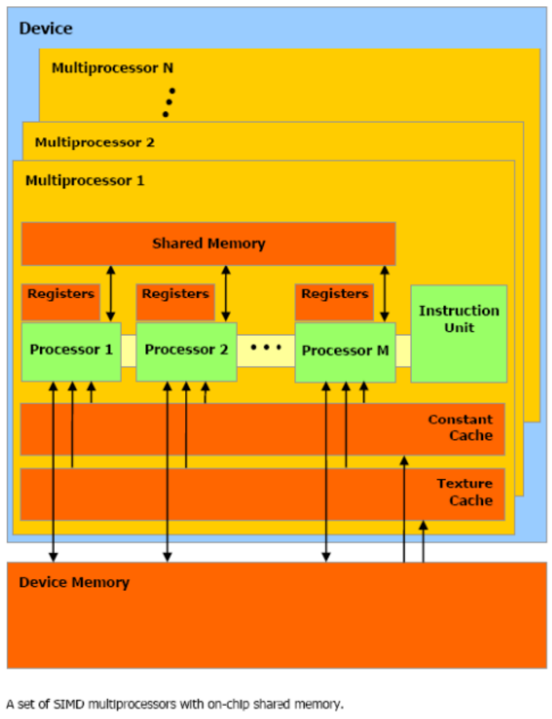


Figure 5.2: Architecture of a GPU (Vuik and Lemmens, 2018)

blocks, which are arranged in a grid. These blocks are then distributed across the available SMs (Vuik and Lemmens, 2018). Both threads and blocks are organized in a three-dimensional structure, with each thread or block assigned a unique index defined by x, y, and z coordinates (Vuik and Lemmens, 2018).

Warps A warp can be defined as a group of threads inside a block. A SM works on a number of blocks and the necessary group of threads for executing a program are selected. These groups of threads are scheduled such that they form groups of warps. A warp on an NVIDIA GPU consists of 32 threads running in parallel (Lin and Grover, 2018).

NVIDIA GPUs execute operations using a collection of threads called warps, in a SIMT (Single Instruction, Multiple Threads) manner (Lin and Grover, 2018). SIMT is similar to SIMD (Single Instruction, Multiple Data), but with a key difference: in SIMT, multiple threads execute the same instruction on different input data (Lin and Grover, 2018), whereas SIMD applies this principle to processor cores instead of threads.

Within a warp, each thread has its own registers to read and write data from varying memory addresses and to follow varying control flow paths (Lin and Grover, 2018). An example of varying control flow is an if/else statement. In that case different groups of threads in a warp perform different branches of the conditional statement. The CUDA compiler works together with the GPU to enhance performance by allowing threads within a warp to execute similar operations in parallel whenever feasible (Lin and Grover, 2018).

5.1.2. Memory on a GPU

Each streaming multiprocessor (SM) has access to different types of memory. These memory types are exclusive to the GPU, and if the CPU needs to access their data, communication via the PCI bus or NVLINK is necessary.

Register Memory Register memory is a high-speed memory type that is exclusive to a single thread-/core.

Shared Memory The shared memory can be accessed by all the threads inside a block (Vuik and Lemmens, 2018).In terms of memory bandwidth, shared memory is slower than registers but faster than other types of memory on the GPU. However, shared memory is typically limited in size. On a DTX H100 device, the shared memory can be configured up to 228 KB (Andersch et al., 2022)."

Texture Memory Texture memory is a form of read-only memory that can be accessed by every thread in every block on a single SM.

Global Memory Global memory is the largest memory type on the SM in terms of size, and it can be accessed by all threads on the SM. However, it is located furthest from the threads (Vuik and Lemmens, 2018). Accessing global memory can take up to 200 times longer than starting a shared memory operation (Vuik and Lemmens, 2018). Global memory resides on the DRAM component of the SM.

5.1.3. Comparison of different NVIDIA GPU Types

For our study different time-integration schemes will be tested on different versions of GPUs. The objective is to test on V100, A100 and H100 hardware. In this section, we compare the specifications of the different GPUs to gain a clearer understanding of the performance differences we can expect.

Note: In this subsection we compare the V100,A100,H100, with the PCI-bus versions for comparison. In table 5.1 different specifications for the different GPU versions are compared with a CPU-model.

Architecture Generally Speaking for the architecture on the chips there are a few differences: The total number of SMs is for V100, A100 and H100 are 80 108, 114, repspectively. the number of FP32 cores an on A100 and V100 are 64 per SM (Krashinsky et al., 2020) and 128 per SM on an H100 (Andersch et al., 2022). the total number of FP64 cores are for V100, A100 (Krashinsky et al., 2020) and H100: 32,32 and 64, respectively. The total number of cores on a device increases for every new version, in table 5.1.

Finally, with this in mind we can conclude that for every new GPU version the total number of cores increases. Furthermore, in Table 5.1 the total number of transistors also increases for every new version; however, the clock speed does not. This means that improvements in performance for newer GPU-versions are obtained by increasing parallelism rather than increasing the clock speeds.

Memory One of the fastest forms of on-chip memory is shared memory. On every SM the V100, A100 and H100 shared memory sizes can be configured up to 96, 164 and 228 KB, respectively (Andersch et al., 2022).

The V100, A100 and H100 are using the third, fourth and fifth PCI-e gen interconnect (Andersch et al., 2022) (Techpowerup, 2018). That means that the H100 has faster communication with the CPU compared to A100 and V100 has also faster communication with the CPU.

The memory bandwidths for the V100, A100 and H100 are 897.0 GB/s (Techpowerup, 2018), 1.94 TB/s (Techpowerup, 2021) and 2.04 TB/s TB/s (Techpowerup, 2023), respectively. Furthermore the (global) memory sizes are 32 GB (Techpowerup, 2018), 80 GB and 80 GB (Andersch et al., 2022), respectively. Also the L2 caches between DRAM (global memory) and SMs had the following sizes: 6MB (Techpowerup, 2018), 80 MB (Techpowerup, 2021) and 50 MB (Techpowerup, 2023), for respectively the V100, A100 and H100 models. Since the global memory is faster and larger for the newer GPU-models, we expect to be able to run complex models in less time.

Performance In terms of float32 and float 64 performance there are a few differences: The Peak FP32 in TFLOPS for non-tensor nodes is 14.13 TFLOPs, 19.49 TFLOPS and 51.22 TFLOPS for V100, A100 and H100. The Peak FP64 in TFLOPS for non-tensor nodes is V100, A100 and H100 7.066 TFLOPS , 9.746 TFLOPS and 25.61 TFLOPS.

Thus, for computationally intensive algorithms one can expect a faster performance on the H100.

Thermal Power Consumption We also consider the Thermal Design Power (TDP), which represents the maximum amount of heat, in watts, that a CPU or GPU can potentially generate (Harding, 2022). A higher TDP typically indicates more computing power, but it also leads to increased heat production and energy consumption. The TDP values for the V100, A100, and H100 are 250W, 300W, and 350W, respectively (Techpowerup, 2018), (Andersch et al., 2022). Thus, as each new GPU version is released, both compute power and energy consumption tend to increase

	V100	A100	H100	Xeon Platinum 8268
release date	Mar 27th, 2018	Jun 28th, 2021	Mar 21st, 2023	2 April 2019
TDP	250 W	300 W	350 W	205 W
Memory Size	32 GB	80 GB	80 GB	1 TB
Maximum memory bandwidth	897 GB/s	1.94 TB/s	2.04 TB/s	140.8 GB/s
Base Clock	1230 MHz	1065 MHz	1095 MHz	2.9 GHz
Number of SMs	80	108	114	-
number of cores	5120	6912	14592	24
Max threads	80×2048	108×2048	114×2048	48
total number of transistors	21,100 million	54,200 million	80,000 million	8,000 million

Table 5.1: Specs comparisons CPU vs GPU, the GPUs are PCI-versions, information from (Techpowerup, 2018,Techpowerup, 2021, Andersch et al., 2022) and Technical-City, n.d.)

5.1.4. GPU Programming Styles

A GPU must be programmed in a Single Program Multiple Data (SPMD) style (Rackauckas, 2020a), meaning that every thread executes the same instructions on different data (Rackauckas, 2020a). For example, in the case of an if statement with two branches, each GPU thread will perform operations from both branches on separate data (Rackauckas, 2020a). As a result, conditional operations on a GPU can be quite slow. Therefore, GPU programs must be highly parallel to achieve optimal performance.

For writing parallel SPMD programs on a GPU, CUDA is the most commonly used language. CUDA stands for Compute Unified Device Architecture. Only NVIDIA GPU-cores can be programmed with CUDA. The execution of a CUDA program is done by all the cores of the GPU, performing simultaneously the same kernel. CUDA is a C++-like programming language. CUDA is able to compile to .PTX kernels (Rackauckas, 2020a). PTX stands for parallel thread execution and is a low level virtual machine and instruction set architecture for carrying out instructions in parallel within the CUDA programming model (Wikipedia, 2024). The PTX instructions are written in assembly code and the compiler on the Nvidia driver translates this into binary code that can be run on Nvidea-GPU cores (Wikipedia, 2024).

An example of a CUDA.jl program that is programmed as a SPMD kernel:

```julia
using CUDA

N = 2^17

x_d = CUDA.fill(7.0f0, N)  # initialize a vector on GPU, with only 7.0 elements

function gpu_mul2!(x)
    index = threadIdx().x
    stride = blockDim().x
    for i = index:stride:length(x)
        @inbounds x[i] =x[i]*2
    end
    return nothing
end

@cuda threads=256 gpu_mul2!(x_d)
```

Kernels are designed to run in parallel by being assigned a certain chunk of the computation, such that the kernels are only supposed to perform computations on the given input. On every GPU-thread, the kernel is called and on every input block the instructions are executed in parallel (Rackauckas, 2020a).

Thus, for this SPMD programming kernel model, the programmer never writes a global algorithm, but instead only writes instructions for components of the algorithms performed on one CUDA core. Hence, we grant the compiler the flexibility to optimize overall execution (Rackauckas, 2020a).

Array-Based GPU Programming Model Another method for writing a program with CUDA.jl is performed on CUDA-arrays (Rackauckas, 2020a). By declaring that an array is a CUDA array type, the operations on this array are performed with CUDA.jl on different GPU-cores. This is the simplest way to perform very parallel operations on a GPU with CUDA.jl.

An example of an array-based CUDA.jl program for matrix-multiplication is:

```
    using CUDA
    A = rand(100,100) #initialize array with random values
    cuA = cu(A) #push to the GPU
    cuA2 = cuA1*cuA1 # Matrix mutliplications on GPU
    A2 = Array(cuA2) #push to the CPU.
```

A drawback of array-based parallelism is that operations on arrays are limited to specific functions that support GPU arrays.

5.2. Solving SWEs on GPUs

Having a fast and accurate numerical simulations for predicting ocean flows is crucial for flood and storm surge forecasts. Having a fast SWE solver, contributes to being able to predict floods early and thus being able to issue a warning in time. Hence, taking into account recent hardware developments is beneficial.

For state of the art GPU-hardware the number of floating point operations per second surpasses the number of bytes per second that can be read from memory. Hence for modern hardware computationally intensive algorithms are preferred.

For numerically solving SWEs on modern GPU-hardware it is relevant to investigate performance for different methods. In section 5.2.1 different GPU-based SWE-solvers with explicit time-integration schemes. for Flood modelling are listed. After that, In section 5.2.2 semi-implicit methods for solving SWEs on a GPU are described. In section 5.2.3 a case where the performance is compared on GPUs and CPUs for both Julia and Fortran is presented. Finally, in section 5.2.4. a number of Navier-stokes solvers for the GPU are compared.

5.2.1. Flood Inundation Modelling

Solving Shallow Water Equations on a GPU with an application to flood inundation models is very common. For flooding simulations the water depth is very low, hence the CFL-condition is easily met. Examples of research done for flood modelling or dam-break simulations on a GPU:

- In (Rak et al., 2024) parallel GPU-solvers are developed for improving flash-flood simulations. Explicit Leapfrog time-integration method in combination with Finite Volumes on a structured grid is used. In this model different real life scenarios are used to validate and test the GPU-implementation. Different cases are tested on different GPU-types: V100, A100 a GTX 1080 and RTX 3090. The GPU-performance on the A100 is the fastest for every test in the paper. The researchers conclude: ” Our implementation achieves a fourfold speedup in comparison to state-of-the-art approaches.” (Rak et al., 2024).
- In (J. Wang et al., 2024) SWes are discretized on a non-uniform grid with muti-time schales and a Finite Volume Method. The selected finite volume method is Harten-Lax-vanLeer-Contact (HLLC) approximate Riemann solution for approximating interface fluxes and for the Conservation Laws a Monotonie Upwind scheme is used. Second order RK-methods are used for time-integration. Different time steps are used on different parts of the grid. In this study, the speed of the GPU-implementation of this model is 4.52 times larger, compared to a traditional uniform grid model (J. Wang et al., 2024).
- In (Aureli et al., 2020), two-dimensional SWEs are numerically solved on a GPU with a finite volume method and a second order RK-scheme. This model was tested against a real life river

flooding scenario. For the real life test case the data obtained form the numerical simulation is very similar to the real life data. To simulate 20 seconds of physical time less than 1 second of simulation time is needed for some cases.

- In (Dazzi et al., 2018) a Local Time Stepping method for solving 2-dimensional SWEs on a GPU is presented. An explicit finite volume scheme is applied for discretization. This model is tested on non-uniform spatial configurations with an application to dam-break and flooding scenarios.
- In (García-Feal et al., 2018) a new parallezation for Iber+ is implemented, based on both CPUs and GPUs. Iber is a hydraulic model for solving 2-dimensional SWEs on an unstructured grid with a Finite Volume method. The new version of Iber was tested to a real life flash flood case that happened in the Pyrenees in 2012. A massive improvements was obtained for the GPU-based solver. According to (García-Feal et al., 2018): "Iber+ was able to simulate 24 h of physical time in less than 10 min using a numerical mesh of almost half a million elements. The same case run with the standard version needs more than 15 h of CPU time.".
- In (Smith and Liang, 2013), two-dimensional SWEs are solved with a second-order accurate Godunov-type MUSCL-Hancock scheme combined with an HLLC Riemann solver. On 1.8 million cells a dam-break scenario is simulated on both a CPU and GPU. Different simulations were done on the following hardware: AMD FirePro V7800 GPU with 2 GB of memory, an Intel Xeon E5-2609 2.40 GHz quad-core CPU with access to 24 GB of memory, and an NVIDIA Tesla M2075 GPU with 6 GB of memory. Also both simulations with 32-bit and 64-bit accuracy were compared. For flooding simulations with 32-bit floats, cause significant errors on a local level. For simulations with 64-bit floats, the simulation on a GPU are faster in terms of performance compared to the CPU-versions.

5.2.2. GPU-solvers for SWEs with IMEX-schemes

The previous GPU-solvers for flood inundation models are mostly using explicit time-integration schemes. Buwalda (2023) implemented SWE solvers for both semi-implicit and semi-explicit schemes. For the semi-implicit method, a pentadiagonal system has to be solved. The system is solved with a CG-method with a RRB (repeated red black) preconditioner (Buwalda et al., 2023). Generally, on a GPU the solvers were often 25 to 75 times faster, compared to solvers on a CPU (Buwalda et al., 2023). Buwalda (2023) tested the numerical methods with cases with shallow water depth. He ultimately, found that explicit methods had better performance on a GPU for low water depths.

Furthermore, an ADI-scheme can also be seen as an IMEX-scheme. In Aackermann et al., 2013 and Zhang and Jia, 2013, ADI-solvers for GPUs were built. For these SWE-solvers tridiagonal linear systems had to be solved. Methods such as cyclic reduction (CR) and parallel cyclic reduction (PCR), were implemented to solve these linear systems on a GPU. Aackerman, (2013) compared both the Thomas algorithm with the CR/PCR-algorithms and found that for small problems the CR/PCR algorithms are faster. He found large speedups of both algorithms on problems with gridsizes of at most 5120 gridponts.

5.2.3. Soving SWEs with Julia vs Fortran

Research done by (Bishnu et al., 2023) had the main objective of comparing the performance of Julia and Fortran on both CPUs and GPUs for solving SWEs. The equations of the MPAS-ocean model are spatially discretized on an unstructured grid with a Finite Volume Method, called TRiSK. For time-integration a first order Forward-Backwards scheme is applied by the researchers. While comparing a Julia-MPI, Fortran-MPI and Julia-GPU implementation, the authors stated the following conclusions:

- The first sequential model implementation of the SWE-solver was 13 times faster in Julia compared to Python with NumPy;
- Implementing Julia optimizations, by removing implicit memory allocations and adding static typing static typing and removing implicit made the sequential implementation 10-20 times faster;
- The performance of Julia-MPI is identical to Fortran-MPI if a small number of CPU-cores is used. Julia-MPI is slower compared to Fortan-MPI if a very high number of cores is used. If a midrange number of cores is selected, Julia-MPI outperforms Fortran-MPI;

- The speed of Julia-GPU on a single node is very close to the speed of Julia-MPI and Fortran-MPI on 64 CPU-cores.

5.2.4. Solving Navier-Stokes

Since the SWEs are derived from the Navier Stokes Equations, investigating how Navier Stokes Equations are solved on GPUs is also interesting. In the following papers a variety of methods can be used:

- In (Onodera et al., 2021), a Conjugate Gradient Solver with a multigrid preconditioner is used for solving the pressure-terms in the Navier Stokes equations. With a geometric multigrid, consisting of a three stage v-cycle the system is preconditioned. A red-black SOR smoother and a cache-reuse optimziation SOR are used for every grid-stage. The solver is tested for for a case concerning two-phase flows in a fuel bundle of a nuclear reactor (Onodera et al., 2021). A major finding in the research is : ”In the strong scaling test, the MG-CG solver with the CR-SOR smoother is accelerated by 2.1 times between 64 and 256 GPUs.” (Onodera et al., 2021).
- In (Zolfaghari and Obrist, 2021), Navier-Stokes equations are spatially discretized with 6-th order finite differences and for time-integration an explicit low-storage third-order RK-scheme is selected. The discretized equations are solved with a Schur-complement. For the solvers on the CPU and GPU a BiCGstab method is used in combination with a preconditioner. A geometric grid with a V-cycle is selected as the preconditioner. This method is not that fast on a GPU. On a GPU a different poisson solver is used. A classical Jacobi method with SOR is selected.
- In (Ha et al., 2018), incompressible Navier-Stokes equations are solved with a semi-implicit scheme. For the semi-implicit scheme, the convection terms are integrated explicitly with a Third order RK-scheme, while the viscous terms are integrated with a Crank Nickelson method. An ADI-method and Fourier-transformed direct solution methods are also used. Speedups are measured on both the Tesla K40 GPU and the Tesla P100 GPU. A central conclusion of this report: ”An overall speedup of 20 times is achieved using a Tesla K40 GPU in comparison with a single-core Xeon E5-2660 v3 CPU in simulations of turbulent boundary-layer flow over a flat plate conducted on over 134 million grids. Enhanced performance of 48 times speedup is reached for the same problem using a Tesla P100 GPU.”(Ha et al., 2018).
- In (Aissa et al., 2017), Navier Stokes is discretized with a Finite Volume Method and solved on a GPU with both explicit and implicit time-integration schemes. For explicit time-integration RK-schemes with four stages are used. For implicit solvers an implicit RK-method with four stages is used with a ILU-preconditioned GMRES solver. On different GPUs and CPUs the methods are implemented and compared. The following CPUs and GPUs are used: E3-1240 (Intel), E5-2640 (Intel), Geforce GTX 780 (NVIDIA) and Tesla K40 (NVIDIA). For this paper the explicit methods are tested on the GTX 780, while the implicit methods are tested on TeslaK40. The K40 has larger global memory compared to GTX780. Finally, on a testcase applied to a subsonic turbine, explicit methods on a GPU, massively outperform implicit methods on a GPU in terms of speedup.

6 Comparing Explicit Time-Integration Methods for SWEs on GPUs and CPUs

In the previous chapter one could notice that many SWEs were solved on a GPU with an explicit time-integration scheme. On a GPU explicit time-integration methods often outperform implicit time-integration solvers. This is caused by the fact that performance on a GPU is more restricted by memory complexity compared to computational complexity. Furthermore, selecting an implicit method requires solving a system of equations for every time-step and hence a larger memory complexity is often needed. However, explicit methods time-step sizes are often severely restricted by the CFL-condition and other stability conditions, consequently increasing the computational complexity.

Different explicit time-integration schemes have different restrictions and properties for solving systems of ODEs. Hence, having a comparison of different explicit time-integration techniques for solving SWEs on a GPU could be beneficial for researchers, who need to choose what method works best their shallow water flow application.

In this chapter the main objective is to compare accuracy, stability and performance trade-offs for different explicit-time integration solvers for SWEs on both GPUs and CPUs. In section 6.1 of this chapter A Shallow Water Model will be derived and its stability is analyzed. This section also contains a derivation for the maximum sizes of time-steps for different time integration techniques such that stability holds. Furthermore, in section 6.2 experiments are presented for comparing computational complexity, errors and times for different time-integration techniques with a variety of time-step sizes. In the following section 6.3 the time, accuracy, computational complexity and errors are compared for different methods on different grid-sizes. Finally, in section 6.4 parallelism on a GPU is investigated, by analyzing memory usage and run-times on different grid-sizes.

6.1. 2-dimensional Shallow Water Model

In this section the SWEs are shown and analyzed. First, in subsection 6.1 a general 2-dimensional shallow water model is presented. Secondly, in subsection 6.1.1 a simplified model for North Sea Tidal-dynamics is derived from the general set of SWEs. Thirdly, in subsection 6.1.2 the spatial discrimination of the simplified Tidal Model is shown. After that, in section 6.1.3 the stability of the spatially discretized model is studied by analyzing its eigenvalues. Furthermore, in section 6.1.4, stability for different explicit time-integration techniques is analyzed and largest time-step sizes are found such that the time-integration methods are stable.

Modelling Shallow Water Flows

A set equations of describing Shallow Water flows is given by the following equations:

$$\frac{\partial h}{\partial t} = -\frac{\partial (Hu)}{\partial x} - \frac{\partial (Hv)}{\partial y}, \tag{6.1}$$

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial x} - v\frac{\partial u}{\partial y} + fv - g\frac{\partial h}{\partial x} - \frac{c_f}{H}u\sqrt{u^2+v^2}, \tag{6.2}$$

$$\frac{\partial v}{\partial t} = -u\frac{\partial v}{\partial x} - v\frac{\partial v}{\partial y} - fu - g\frac{\partial h}{\partial y} - \frac{c_f}{H}v\sqrt{u^2+v^2}, \tag{6.3}$$

where $H = D+h$, where D is water depth relative to the geoid and H is the total water column. The free surface level elevation is given by h. The variables u and v correspond to depth averaged velocities. The variable f is equal to the Coriolis parameter. The bottom friction coefficient is presented by c_f.

Substituting $p = Hu$ and $q = Hv$ for the conservation of volume in Equation (6.1) yields:

$$\frac{\partial h}{\partial t} = -\frac{\partial p}{\partial x} - \frac{\partial q}{\partial y}.$$

Subsequently, substituting $p = Hu$ and $q = Hv$ in the momentum equations of given by (6.2)-(6.3) results in:

$$H\frac{\partial u}{\partial t} = -Hu\frac{\partial u}{\partial x} - Hv\frac{\partial u}{\partial y} + fHv - Hg\frac{\partial h}{\partial x} - H\frac{c_f}{H}u\sqrt{u^2+v^2}, \tag{6.4}$$

$$H\frac{\partial v}{\partial t} = -Hu\frac{\partial u}{\partial x} - Hv\frac{\partial u}{\partial y} - fHu - Hg\frac{\partial h}{\partial y} - H\frac{c_f}{H}v\sqrt{u^2+v^2}. \tag{6.5}$$

Rewriting Equations (6.4)-(6.5), yields:

$$\frac{\partial p}{\partial t} = \frac{p}{H}\frac{\partial h}{\partial t} - p\frac{\partial}{\partial x}(\frac{p}{H}) - q\frac{\partial}{\partial y}(\frac{p}{H}) + fq - Hg\frac{\partial h}{\partial x} - H\frac{cf}{H^2}p\sqrt{p^2+q^2}.$$

$$\frac{\partial q}{\partial t} = \frac{q}{H}\frac{\partial h}{\partial t} - p\frac{\partial}{\partial x}(\frac{q}{H}) - q\frac{\partial}{\partial y}(\frac{q}{H}) - fp - Hg\frac{\partial h}{\partial y} - \frac{c_f}{H^2}q\sqrt{p^2+q^2}.$$

The set of SWEs given by Equations (6.1)-(6.3), rewritten fully in terms of p and q is given by:

$$\frac{\partial h}{\partial t} = -\frac{\partial p}{\partial x} - \frac{\partial q}{\partial y}, \tag{6.6}$$

$$\frac{\partial p}{\partial t} = -\frac{\partial}{\partial x}\left(\frac{p^2}{H}\right) - \frac{\partial}{\partial y}\left(\frac{pq}{H}\right) + fq - gH\frac{\partial h}{\partial x} - c_f H^{-2}p\sqrt{p^2+q^2}, \tag{6.7}$$

$$\frac{\partial q}{\partial t} = -\frac{\partial}{\partial x}\left(\frac{pq}{H}\right) - \frac{\partial}{\partial y}\left(\frac{q^2}{H}\right) - fp - gH\frac{\partial h}{\partial y} - c_f H^{-2}q\sqrt{p^2+q^2}. \tag{6.8}$$

6.1.1. Simplified Model for North Sea Tides

Using the set of general SWEs from Equations (6.6)-(6.8), a simplified test-case for modeling tides in a North Sea model is found.

For defining the simplified test-case, the following spatial domain is

$$D = \{(x,y) \in \Re \times \Re : 0 \le x \le L_x, 0 \le y \le L_y\},$$

where L_x and L_y are the horizontal and vertical length, respectively. The variables L_x and L_y are both equal to 500000 meters, that is 500 kilometers.

On the boundary of domain D the following conditions are defined for the test case scenario:

$$p(0,y,t) = p(L_x,y,t) = 0, \quad q(x,0,t) = q(x,L_y,t) = 0. \tag{6.9}$$

(6.10)

In equation (6.9), the boundary conditions are defined on the p and q variables, and set to zero. However, for real life simulations on a North Sea model, the boundary conditions on p and q are not zero, since the North Sea has open boundaries. Often for researchers define non-zero boundary conditions on h for North sea Models and obtain the boundary values via measurements.

In addition to defining the boundary conditions, initial conditions are chosen equal to zero for the test case:

$$h(x,y,0) = p(x,y,0) = q(x,y,0) = 0. \tag{6.11}$$

Furthermore, the Equations (6.6-6.8) are altered such that they represent ocean flows. The advection therms in the Equations (6.7-6.8) are removed. Also to Equation (6.7) an artificial tidal potential term is added such that the model generates a rotating oscillation that roughly resembles the tides in the North Sea.

The tide potential is given by $\alpha\sin(m_2 t)$, where α and m_2 are tidal amplitude and period, respectively. The m_2-tide has a a cycle for every 12.4 hours and therefore $m_2 = \frac{2\pi}{3600*12.4}$. For α a value of 10^{-5} is selected.

Finally the particular SWEs for a Simplified North-Sea test-case are given by the Equations (6.12)-(6.14).

$$\frac{\partial h}{\partial t} = -\frac{\partial p}{\partial x} - \frac{\partial q}{\partial y}, \tag{6.12}$$

$$\frac{\partial p}{\partial t} = fq - gH\frac{\partial h}{\partial x} - c_f H^{-2} p\sqrt{p^2+q^2} - H\alpha\sin(m_t t), \tag{6.13}$$

$$\frac{\partial q}{\partial t} = -fq - gH\frac{\partial h}{\partial y} - c_f H^{-2} q\sqrt{p^2+q^2}, \tag{6.14}$$

where the gravitational acceleration g is equal to $9.81\ m/s^2$ and the bottom friction constant C_f chosen to be equal to 0.003. The Coriolis parameter is given by $f = 2\Omega sin(\phi)$. The Ω represents the angular speed of the earth and is equal to $\frac{2\pi}{3600\times 24}\ m/s$. the degrees longitude given by ϕ is approximated to be $\frac{\pi}{3}$. Also we select an ocean depth equal to $D = 60$ m, for the test-case.

6.1.2. Spatial Discretization

The variables h, p and q are discretized on a C-arakawa grid. Our grid has $n_x \times n_y$ grid-cells, where n_x and n_y are the number of grid-cells in the horizontal and vertical direction, respectively. In Figure 6.1 the grid-ordering is shown. Every grid cell has magnitude of Δx by Δy, given by:

$$\Delta x = \frac{L_x}{n_x}, \quad \Delta_y = \frac{L_y}{n_y}.$$

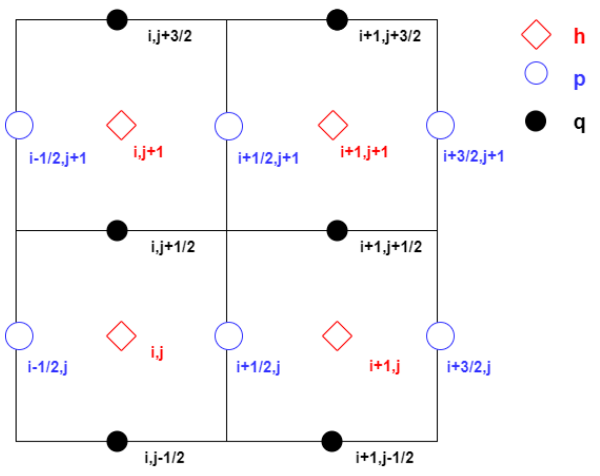


Figure 6.1: The variables h, p and q on a C-arakawa grid.

For spatially discretizing the mass conservation, central differences are used. Equation (6.12) is spatially discretized as follows:

$$\frac{\partial h_{i,j}}{\partial t} \approx -\frac{p_{i+1/2,j} - p_{i-1/2,j}}{\Delta x} - \frac{q_{i,j+1/2} - q_{i,j-1/2}}{\Delta y} \quad i = 1, \dots, n_x, j = 1, \dots, n_y. \tag{6.15}$$

To discretize Equation (6.13) on the grid points corresponding for p at location $(i + \frac{1}{2}, j)$, including the Coriolis term fq requires the values of q on the mesh-point $(i + \frac{1}{2}, j)$. However, the variable q is not defined in those grid points. Hence the values of q are approximated in grid points corresponding to p, by averaging the values of q in the four neighboring points of q.

$$\tilde{q}_{i+1/2,j} \approx \frac{1}{4}(q_{i,j+1/2} + q_{i,j-1/2} + q_{i+1,j-1/2} + q_{i+1,j+1/2}). \tag{6.16}$$

Likewise, Equation (6.14) is discretized at the gridpoint $(i, j + 1/2)$ of the q-variable on the staggered grid. Also, the Coriolis term fp requires the values of p on the grid-point locations corresponding to q. For that reason the values of p are approximated on grid points linked to q, by averaging the values of p in adjacent nodes.

$$\tilde{p}_{i,j+1/2} \approx \frac{1}{4}(p_{i-1/2,j+1} + p_{i+1/2,j+1} + p_{i-1/2,j} + p_{i+1/2,j}) \tag{6.17}$$

For evaluating H in various grid points one more averages are computed in the following way:

$$\tilde{H}_{i+1/2,j} = \frac{1}{2}(H_{i+1,j} + H_{i,j}), \tag{6.18}$$

$$\tilde{H}_{i,j+1/2} = \frac{1}{2}(H_{i,j} + H_{i,j+1}), \tag{6.19}$$

where the discretized variables H^p and H^q correspond to the averages in grid points relating to the p and q-components, respectively.

For spatially discretizing the spatial derivatives in Equations (6.13) and (6.14) central differences are applied. After that, substituting the averages of the discretized p, q and H, from equations (6.16)-(6.19) in Equation (6.13) and (6.14), yields the following spatially discretized equations for momentum conservation.

$$\begin{aligned} \frac{\partial p_{i+1/2,j}}{\partial t} &= f\tilde{q}_{i+1/2,j} - g\tilde{H}_{i+1/2,j}\frac{h_{i+1,j} - h_{i,j}}{\Delta x} - \\ c_f(\tilde{H}_{i+1/2,j})^{-2} p_{i+1/2,j}&\sqrt{p^2_{i+1/2,j} + \tilde{q}^2_{i+1/2,j}} - \tilde{H}_{i+1/2,j} A \sin(m_t t), \end{aligned} \tag{6.20}$$

$$\begin{aligned} \frac{\partial q_{i,j+1/2}}{\partial t} &= -f\tilde{p}_{i,j+1/2} - g\tilde{H}_{i,j+1/2}\frac{h_{i,j+1} - h_{i,j}}{\Delta y} - \\ &c_f(\tilde{H}_{i,j+1/2})^{-2} q_{i,j+1/2}\sqrt{q^2_{i,j+1/2} + \tilde{p}^2_{i,j+1/2}}. \end{aligned} \tag{6.21}$$

The boundary conditions in Equation (6.9) are given on a discretized spatial grid by:

$$p_{1/2,j} = p_{n_x+1/2,j} = 0, \quad j = 1, \dots, n_x, \tag{6.22}$$

$$q_{i,1/2} = q_{i,n_y+1/2} = 0, \quad i = 1, \dots, n_y. \tag{6.23}$$

6.1.3. Stability Analysis

The spatially discretized equations in (6.15)-(6.21) can be presented as an ordinary differential equation:

$$\frac{d}{dt}\begin{bmatrix}\mathbf{h}\\ \mathbf{p}\\ \mathbf{q}\end{bmatrix} = \mathbf{f}(\mathbf{h}, \mathbf{p}, \mathbf{q}), \tag{6.24}$$

where the entries of $\mathbf{h}$, $\mathbf{p}$ and $\mathbf{q}$ correspond to spatially discretized variables h, p and q on a numerical grid.

For comparing stability properties of different time-integration solvers applied to SWEs, firstly the eigenvalues of the Jacobian of the SWEs are computed. Subsequently, using these eigenvalues one can examine for which time-steps sizes different explicit time-integration schemes are stable.

In this subsection, the tests for stability are carried out on a grid with $n_x = 20$ by $n_y = 20$ grid points in the horizontal and vertical directions, respectively.

the Jacobian of the function $\mathbf{f}$ from Equation (6.24) is computed and evaluated in the initial condition. To examine the properties of the system of equations in (6.24) eigenvalues of the Jacobian are computed. In Figure 6.2 the real and imaginary parts of the eigenvalues of the Jacobian are shown. In this figure one can observe that for linearizations evaluated in the initial condition the eigenvalues are equal to zero or purely imaginary values. Hence in this point there is no damping of waves and the solutions are periodic.

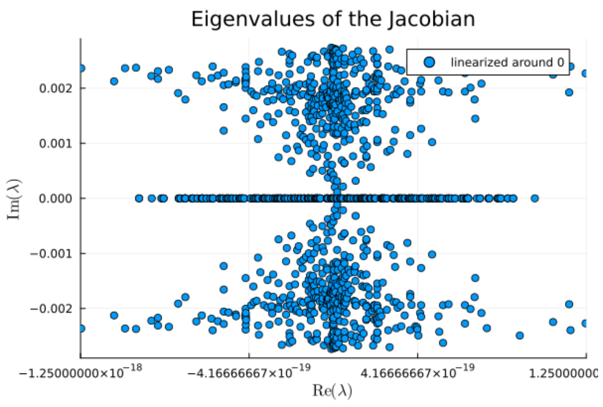


Figure 6.2: Eigenvalues of the Jacobian, linearized around 0.

Furthermore a simulation-point is computed by solving the ODE using a Runge Kutta 4 method with a time-step of 1000 seconds for 10 days. The Jacobian is evaluated in this endpoint and eigenvalues of this Jacobian are computed. In Figure 6.3 the real and imaginary parts of the eigenvalues of the Jacobian are shown. One can observe that for linearizations around a simulation-points some of the eigenvalues have negative real parts. Hence a small amount of damping exists in our simulation, which is a result of a nonzero linearized bottom fiction evaluated in a simulation point.

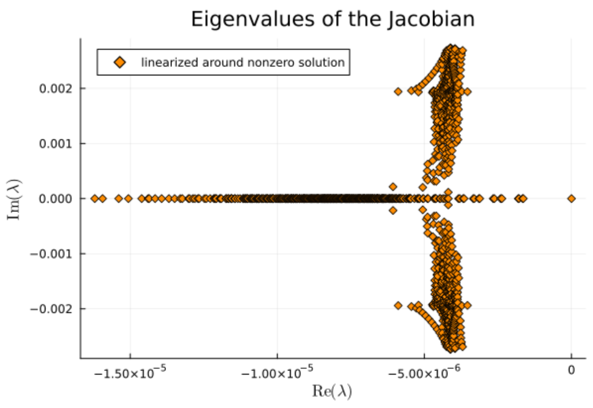


Figure 6.3: Eigenvalues of the Jacobian, linearized around a simulation-point.

In example 3.3.2 from chapter 3 the eigenvalues of a simplified non-conservative SWE with only pressure terms were computed and shown to be purely imaginary. The imaginary eigenvalues have the following maximum imaginary values:

$$\lambda = i\frac{2\sqrt{2Dg}}{\Delta x} \approx i0.002744827863455193$$

In Figure 6.3, the eigenvalues with the largest imaginary values is equal to approximately $\lambda = i0.0027363665806953144$. Furthermore, for the linearization around the non-zero vector, the eigenvalue with the largest imaginary values is equal to: $\lambda = -4.114776714106743e-6 \pm i0.002738650757621181$. Hence, one see the eigenvalues with the largest imaginary components imposing a CFL-like stability condition. Note, that the eigenvalue of the simplified SWE problem represented in section 3.3.2, containing only linear pressure terms, has also an purely imaginary eigenvalue with at most $\lambda = i\frac{2\sqrt{2Dg}}{\Delta x}$. Hence, we can say that linearized pressure terms are in our study, have the biggest impact on the spectrum of the system.

The bottom friction term in Equation (6.14) is equal to:

$$\tau_q = \frac{c_f q}{H^2}\sqrt{p^2+q^2}.$$

The derivative of τ_q is:

$$\frac{\partial \tau_q}{\partial q} = \frac{c_f}{H^2}\sqrt{p^2+q^2} + \frac{c_f q^2}{H^2}(\sqrt{p^2+q^2})^{-1}.$$

If we linearize τ_q around the zero point, the derivative of this value is equal to 0:

$$\frac{\partial \tau_q}{\partial q}\Big|_{q=0,p=0,h=0} = \lim_{q\to 0}\frac{c_f}{D^2}\frac{q^2}{|q|} = 0,$$

Furthermore, the eigenvalues in Figure 6.2 have a real part in the interval $(-1.25e-18, 1.25e-18)$, which smaller than the rounding errors for Float64 precision given by $2.22e-16$. Hence the real parts of these eigenvalues are also equal to zero.

In contrast, when we linearize the system around nonzero values (see Figure 6.3), the real parts of the eigenvalues are often negative. For the nonzero point, the maximum values of variables $|p|$, $|q|$, are approximately 10.39 and 10.87. For the value of H, we have $H = D + h$. Since, in our application we assume $h << D$, we approximate $H \approx D = 60m$. Therefore for bottom friction we approximate:

$$\tau_q = q\frac{c_f}{D^2}\sqrt{p^2+q^2} \approx q \times \frac{0.003}{60^2} \times \sqrt{10.39^2+10.87^2} \times 10^{-5} \approx (1.25e-5)q,$$
$$\tau_p = p\frac{c_f}{D^2}\sqrt{p^2+q^2} \approx (1.25e-5)p.$$

Since the the maximum negative value of the real parts in Figure 6.3, is equal to 1.6e-5. Hence we can say that the bottom friction, is the main cause of damping of errors in our stability analysis.

6.1.4. Numerical Stability for Time-Integration Methods

If we multiply eigenvalues of linearized SWEs, with the chosen time-step size, one can examine whether stability exists for a time-integration scheme. For $z = \lambda dt$, where dt is the time-step and λ is the eigenvalue of the linearized system of equations. The values of z have to exist within a stability region corresponding to the selected time-integration technique. In chapter (4), different stability regions for different time-integration methods are given. In this subsection the largest values of dt were found such that the values of z still stay within the stability region.

For the Runge Kutta 4, a time-step size of 1000s is selected. In Figure 6.4, all the eigenvalues multiplied by time-step are shown. The amplification factor for the RK4-scheme is denoted by |RK4| in Figure 6.4. The red line displays for which z-values close to the z-values of the lineariszation, |RK4| are equal to one. For stability the z-values need to exists within the red lines, to avoid having a amplification factor larger than one. One can observe that for dt $= 1000$ s, all our z-values are located within or on the solid red line. If we increase the time-step sizes with 50 seconds or more, the Runge Kutta 4 method is no longer stable in our case.

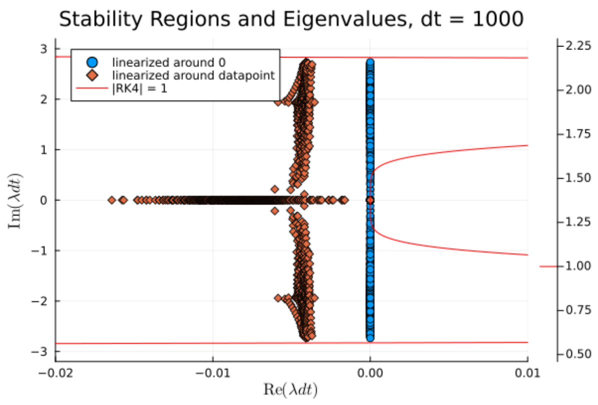


Figure 6.4: Stability region for Runge-Kutta 4.

Methods such as the third order Ralston or the SSPRK33 (third order, three stage, strong stability preserving Runge Kutta) are third order Runge Kutta methods. For these third order methods, a time-step size of 600 seconds is selected. In Figure 6.5, the amplification factor |RK3| for the RK3-methods, and z-values when $dt = 600$s are shown. One can observe that for $dt = 600$s, all our $z-$values are located within or on the solid red line. If we increase the time-step sizes with 50 seconds or more, the z-values will end up outside the red-line and our methods are no longer stable.

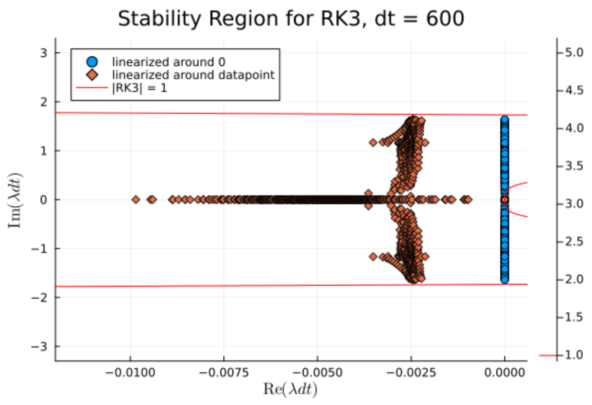


Figure 6.5: Stability region for Runge-Kutta 3

The second order Ralston method and Heun scheme are both second-order Runge Kutta (RK2) time-integration techniques. For obtaining stability while using a RK2 methods, the z-values should have at least a negative real part. In Figure 6.6, the stability region for a second order Runge-Kutta is displayed, together with the $z - values$, when $dt = 75$s. The red line represents the z-values such that the amplification factor |RK2| for RK2-methods is equal to one. If time-independent components of the SWEs are linearized around the simulation point, the eigenvalues with imaginary components have a negative real part at with the size of $\mathcal{O}(10^{-6})$. The z-values (denoted by orange diamonds) exist within the stability region, when $dt = 75$s. Nevertheless, for the purely imaginary nodes (denoted by blue circles) it is impossible to find a time step small enough such that that the stability conditions are met.

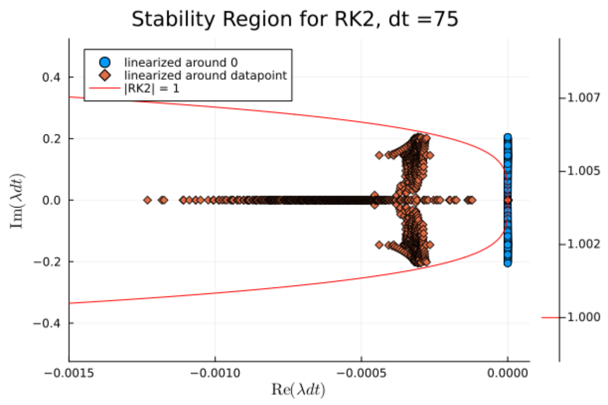


Figure 6.6: Stability region for Runge-Kutta 2

Similarly to the stability criteria of RK1-methods, Euler Forward method requires the z-values to at least have a negative real part for stability. In Figure 6.7, the stability region for Euler forward and the z-values when $dt = 0.9$ are displayed. If our system is linearized around the simulation-point, the eigenvalues have a negative real part. For the time step $dt = 0.9$, the z-values (denoted by orange diamonds) indeed fall within the stable region. However for the z-values corresponding to eigenvalues of the system linearized around the zero-vector (see blue circles), the z-values do not fall in stability region. Since the eigenvalues linearized around zero are purely imaginary, choosing smaller time-steps does not yield numerical stability. In subsequent sections, we chose not to set-up experiments with the Euler Forward methods, since we have to use incredibly small time-steps, resulting in a very high computational complexity due to many time iterations.

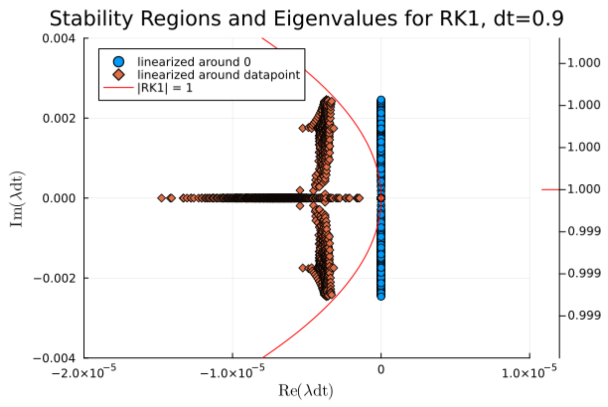


Figure 6.7: Stability region for Runge-Kutta 1

For the Runge Kutta 5, a time-step size of 350 seconds is picked. In Figure 6.8, the stability region with z-values is displayed. In this case, all our z-values are located within or on the red line. For step sizes between 350-500 seconds, only the z-values (given by the red diamonds) corresponding to the system linearized around the non-zero point remain in the stability region.

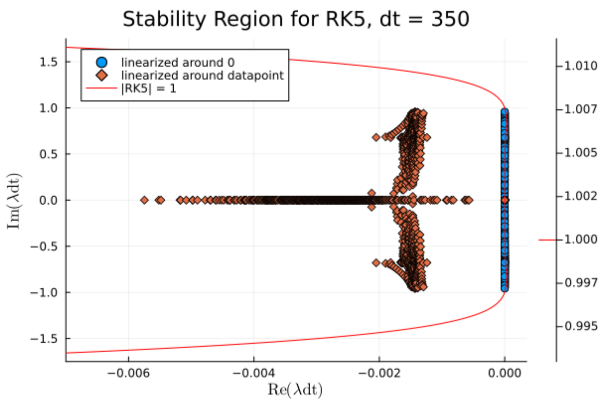


Figure 6.8: Stability region for Runge-Kutta 5

For the second order Adams-Bashforth,the stability region with z-values is displayed in Figure 6.9. For purely imaginary eigenvalues (corresponding to the blue circles) numerical stability can never be realized for the second order Adams-Bashforth. However for the z-values corresponding to eigenvalues of the system linearized around a nonzero vector (see orange diamonds), numerical stability exists for a time-step equal to 60 seconds.

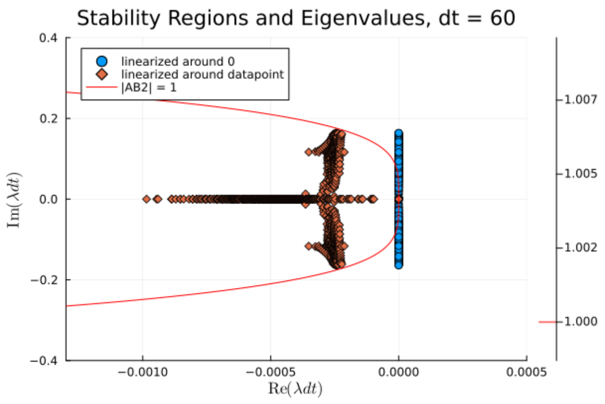


Figure 6.9: Stability region for Adams-Bashforth 2

The shape of the stability region of a time-integration method in combination with the spectrum of the linearized SWEs has a large impact on the size of the maximum allowed time-step. The spectrum of the linearized SWEs changes as, we linearize around different input-values. Hence the numerical solutions, computed in previous time iterations, change the stability properties of the system. If these solutions are close to 0, these solutions do not have a negative eigenvalue, and a stabile time-step for RK1, RK2 and AB2-methods can not be found. Only if numerical solutions in the simulations are moving away from 0, we can find a time-step such that stability holds and errors from previous time iterations in 0-values can be damped. Furthermore, for the RK5- method a time-step size of $500s$ is stable in case the system results in non-zero solutions, while at most a time-step of 350s is stable in the case the solutions are very close to zero. Hence, during the numerical simulation different numerical solutions at different time-steps change the stability properties of the system.

Stability Condition Finally, we find that the stability conditions on time step size in our study are mostly limited by the complex z-values, the largest imaginary z-value is approximately:

$$z_{im} = \pm \Delta t \frac{i2\sqrt{2Dg}}{\Delta x},$$

And to remain in a stability region we need:

$$|z_{im}| = \Delta t \frac{2\sqrt{2Dg}}{\Delta x} < K,$$

where K is a constant, that differs per time-integration scheme. A stability condition on Δt is:

$$\Delta t < K \frac{\Delta x}{2\sqrt{2Dg}}.$$

We have computed the maximum allowable time-steps on a 20×20-grid. If we were to increase the grid in both x- and y-direction by a factor R, we need to divide the maximum allowable time step by a factor R. Hence, when we determine the maximum allowable time step on a $n_x \times n_x$ grid,we use the following trick to approximate a stable time step:

$$\Delta t_{fine} = \Delta t_{coarse} \frac{20}{n_x},$$

where Δt_{coarse} is the maximum allowable time step, found for the 20×20 grid problem and Δt_{fine} is the maximum allowable time-step for the fine-grid problem.

6.2. Experiments on GPUs and CPUs for varying time-step sizes

In this subsection, we present the initial experiments and results comparing various time-integration methods tested across different time-step sizes.

6.2.1. Experiments with different time-integration schemes

For the explicit Runge-Kutta and Adams-Bashforth methods, the time-step size is chosen to be very close to the maximum value that ensures stability. Table 6.1 lists the time-step sizes for the different methods. The approximations for the RK2, RK3, RK4, RK5, and AB2 methods from the previous sections are used. For ORK256, AB3, and AB4, the maximum time-step sizes were determined through a trial-and-error approach. Using these largest time-step sizes, experiments are conducted on a 20×20 computational grid over a 10-day simulation period.

Four different time-step sizes are selected by successively dividing the largest time-step by 2, four times. Errors are estimated using Richardson interpolation between the numerical solutions obtained with these time-step sizes. The error is approximated as follows:

$$\begin{aligned} \mathbf{H}^{\Delta t} &= \frac{(2^d \mathbf{h}^{\frac{\Delta t}{2}} - \mathbf{h}^{\Delta t})}{2^d - 1}, \\ \mathbf{E}^{\Delta t} &= ||\mathbf{h}^{\Delta t} - \mathbf{H}^{\Delta t}||_2, \end{aligned}$$

where $\mathbf{h}^{\frac{\Delta t}{2}}$ and $\mathbf{h}^{\Delta t}$ are the numerical solutions at the last time iteration with time-step sizes equal to $\Delta t/2$ and Δt, respectively. The order of a time-integration method is given by d. The variables $\mathbf{H}^{\Delta t}$ and $\mathbf{E}^{\Delta t}$, represent a higher-order approximation of a numerical solution and the error-approximation for time-step sizes equal to Δt, respectively.

The GPU experiments were conducted on an H100 node (NVIDIA H100 PCIe 80 GB), while the CPU experiments were performed on an Intel® Xeon® Platinum 8462Y+ node. These experiments were carried out on the Deltares Cluster. Note, that in our implementation the cpu experiments are always done serially.

Table 6.1: Largest time-step approximations for different time-integration schemes.

method	dt [s]	function evaluations/iteration	order
RK5	350	5	5
RK4	1000	4	4
Ralston3/SSPRK33	500	3	3
Heun/Ralston2	70	2	2
ORK256	1000	5	2
AB2	60	1	2
AB3	250	1	3
AB4	150	1	4

Comparisons between experiment results on CPU and GPU

In this subsection, we compare the results obtained using different methods. First, the errors associated with each method are analyzed. Next, the performance of various explicit schemes is examined. Additionally, the computational complexity is evaluated. Finally, the trade-offs between errors and performance are discussed.

Errors for different methods Figures 6.10 and 6.12 present the results of the experiments conducted on a CPU using double precision and single precision, respectively. The findings from the GPU experiments are presented in Figures 6.12 and 6.13, for double precision and single precision, respectively.

In subplot (a) of each of these four figures, the approximated errors are plotted as a function of the time-step size. In the figures, it can be observed that, with the exception of the ORK256 method, the lines corresponding to higher-order methods are positioned to the right of those corresponding to lower-order methods. Furthermore, the methods the steeper lines in the plot correspond to lines with a higher order accuracy,i.e. the lines corresponding to AB4 and Ralston3 are steeper compared to lines corresponding to AB2 and Ralston2. Hence, it is clear that for higher-order methods, reducing the time-step size results in greater improvements in accuracy compared to applying the same time-step size reductions to lower-order time-integration techniques. Furthermore, when the selected time-steps and the order of the errors are the same, the errors for the various RK methods are smaller compared to those of the linear multi-step methods. Finally, it is worth noting that some higher-order methods, such as RK4, allow for much larger time-steps compared to lower-order methods.

In Figures 6.10(a) and 6.12(a), the approximated error sizes are shown for the experiments conducted on the CPU using double precision and single precision, respectively. For experiments conducted with double precision, the errors decrease as the time-step size becomes smaller. In contrast, for single precision, it can be observed that the errors increase when the smallest time-step size is selected for the second-order Adams-Bashforth and second-order Heun and Ralston methods. For RK methods with a fifth order, the errors increase as the time-steps become smaller.Moreover, selecting smaller time-steps for the AB4 method does not lead to a reduction in errors when the time-step size is already small in the single precision case.

In Figures 6.11(a) and 6.13(a), the approximated error sizes are shown for experiments on the GPU with both double and single precision, respectively. For experiments done with double precision the errors are reduced as the time-step size becomes smaller. In contrast one can observe that if smaller step-sizes are selected for experiments done with single precision, errors do always not decrease or do even increase as the time-step size becomes smaller. Similar to the CPU experiments, when selecting the smallest time-step during the single precision tests with the second-order Adams-Bashforth, Heun, and Ralston methods, the errors increase. Also, for selecting smaller time-steps for fifth order RK, errors will increase. Again, for single precision tests of the fourth order Adams-Bashforth on a GPU, smaller errors are not obtained as we decrease the time-step size. Finally, in terms of differences between the results for double and single precision experiments, there are significant similarities between the CPU and GPU experiments.

Rounding errors can become important. For Runge-Kutta 5, in the single precision case, the time step

$\Delta t = 350s$, yields the smallest error with size $\mathcal{O}(1e-6)$. In the 32-bit case the errors explode for RK5, if we increase the time-step size. This shows that the RK5 method is not entirely robust. This can be explained by the fact that shown in Figure 6.8, the z-values for a numerical solution equal to zero, is exactly on the stability line, which means that the amplification factor for the errors is exactly equal to 1. As, a consequence the rounding errors for single precision are not reduced by the numerical time-integration scheme, and will aggregate over time. The RK4-methods seems more robust. In Figure 6.4, one can observe that there are always a few z-values that fall within the stability region, and not on the boundary. That means that, the amplification factor for RK4, is smaller than one and as a result very small rounding errors are damped by the numerical method itself.

For second order methods such as the Ralston, Heun and Adams-Bashforth schemes, rounding errors are also relevant, as errors increase in the smallest time-step. This might be caused by the rounding errors in combination with possibly some instabilities. In the Figures 6.6 and 6.9 we observed that second order methods were only stable in a nonzero simulation-point, with maximum allowable time-steps of order $\mathcal{O}(10)$. Hence at some point during the simulation the amplification factor of the time-integration scheme is larger or equal than 1 and rounding errors are not always damped.

When working with single precision (32-bit), rounding errors are often different compared to (64-bit). The rounding error for 32-bit floating-point numbers is of order (10^{-8})), which results in computations become less precise. Methods that require smaller time-steps seem to be more affected by these rounding errors. When the time-step becomes too small, rounding errors can dominate, leading to poor results or instability. On the other hand, methods like RK4 and ORK256 appear to be more robust. These methods allow for larger time-steps.

Performance for Various Explicit Schemes In subplot (b) of the figures, the simulation time is plotted against the time-step size. One can observe that the linear multi-step methods have the same time-step versus simulation time trade-off for different orders. A first observation is that the lines corresponding to the second, third and fourth order Adams-Bashforth methods are to the left of all the lines in the graph. Hence, Adams-Bashforth methods have the best performance when using small time-steps. For Runge-Kutta methods with more stages, such as ORK256 and RK5, and RK4, the lines are to the right in the figures compared to Runge-Kutta methods with fewer stages, such as the Heun scheme. One can thus observe that as the number of stages increases for a Runge Kutta method, more simulation time is needed for the same time-step size. However, since for the Runge-Kutta 4 and 5 methods the error is fourth order, the RK4-method still has a smaller simulation time to error relation compared to lower order RK-methods such as Heun.

In Figures 6.10(b) and 6.12(b), the simulation time on a CPU is shown for different time-steps in both double and single precision, respectively. The required simulation time for different time-steps in double and single precision is similar.

In Figures 6.13(b) and 6.14(b) the simulation time on the GPU is shown for multiple time-steps, with double and single precision respectively. The necessary amount of time for running a variety of time-integration schemes for different time-steps on double precision is larger compared to single precision on a GPU.

By boldly comparing simulation times for GPU and CPU-experiments, the results in Figures 6.10(b) with 6.12(b), and Figures 6.11(b) with 6.13(b) can be set side by side. From those figures we can observe that it takes $2^4 - 2^5$-times more simulation time to run experiments on a coarse grid on a GPU compared to a CPU. Hence, for problems on a 20×20 grid, solving on a GPU instead of a CPU results in reduced performance. This can be explained by the fact that on a GPU extra overhead for setting up parallelism is needed. There is an overhead for starting a GPU-kernel and the amount of work is limited on a coarse grid. Hence the parallelism will be small. For the experiments on a GPU only a fraction of the available GPU-cores are utilized. Only for experiments on finer grids, where the computational work outweighs the costs of parallelism, one can expect the performance on a GPU be superior to the performance on CPU.

Computational Complexity In Figures 6.10(c), 6.11(c), 6.12(c), and 6.13(c), it can be observed that for every time integration method, the slope of time to function evaluations is the same, except for the second-order Adams-Bashforth method. The second-order Adams-Bashforth exhibits a similar slope compared to the other curves. Since we implemented our own AB2() function, compatible with the DifferentialEquations.jl solve functions, the AB2() method must be retrieved from a separate file, resulting in some initial overhead.

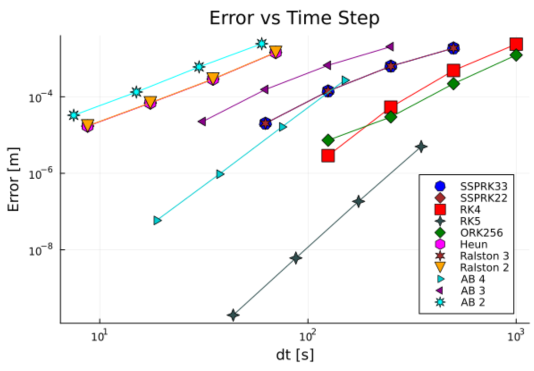
Trade-offs Between Errors and Time In the Figures (d), the errors are plotted against the computational time. In these figures, the lines for the fifth-order, fourth-order, and second-order ORK256 methods are positioned to the left of the AB2, AB3, Heun, Ralston, and SSPRK33 methods. Hence, using fourth- and fifth-order methods or an ORK256 scheme will yield the best accuracy improvements as the running time increases. Furthermore, if one compares the lines corresponding to third order SSPRK33, Ralston 3, and Adams-Bashforth with the second order Adams-Bashforth, Ralston and Heun methods in figures 6.10(d) and 6.12(d) the third order methods are to the left of the second order methods. Therefore, selecting a third-order technique yields better accuracy improvements as the running time increases compared to choosing a second order method.

Furthermore, if the error-time-trade offs in figures 16.10(a) and 16.11(a) are compared, one can observe that for double precision there is a direct time-error trade-off. For the single precision case this is not the case for second order methods with very small time-steps and RK5 methods in general. This can be explained by the fact that if small time-step sizes are selected, errors do not decrease for single precision, while simulation time does increase.

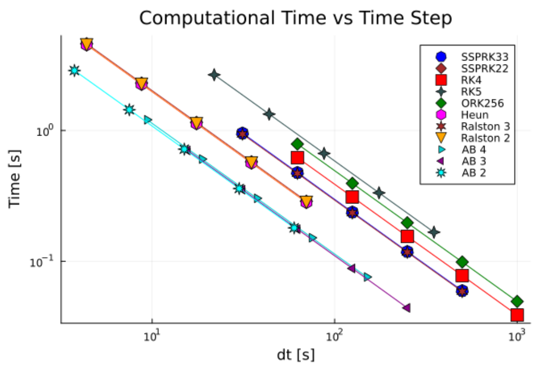
In Figures 6.12 and 6.13 the results are displayed for the experiments on a GPU with double and single precision. Furthermore, if the error-time trade-offs in figures 16.10(a) and 16.11(a) are compared, that for double precision experiments errors reduce for longer experiments, while for single precision this is not the case.

The RK4 method performs best at low accuracies of size $10^{-2} - 10^{-4}$, alongside ORK256. The AB4 method requires smaller time steps but only one function evaluation per time step, whereas RK4 requires four function evaluations per time step. As a result, AB4 is slightly more accurate at intermediate accuracies of $10^{-4} - 10^{-6}$. For higher accuracies of size $10^{-6} - 10^{-8}$, the RK5 method is preferred.

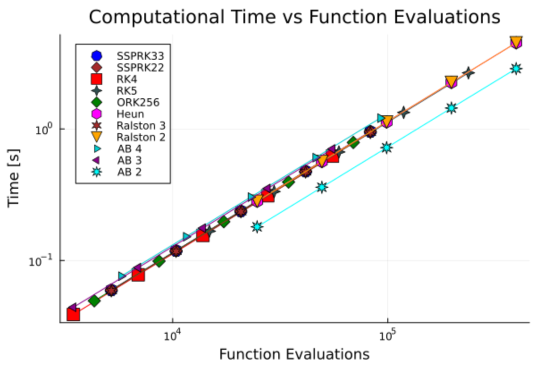
The RK4 method performs best at low accuracies in the range of 10^{-2} to 10^{-4}, alongside ORK256. The AB4 method requires smaller time steps but only one function evaluation per time step, whereas RK4 requires four function evaluations per time step. As a result, AB4 performs slightly better at intermediate accuracies between 10^{-4} and 10^{-6}. For higher accuracies in the range of 10^{-6} to 10^{-8}, the RK5 method is preferred.



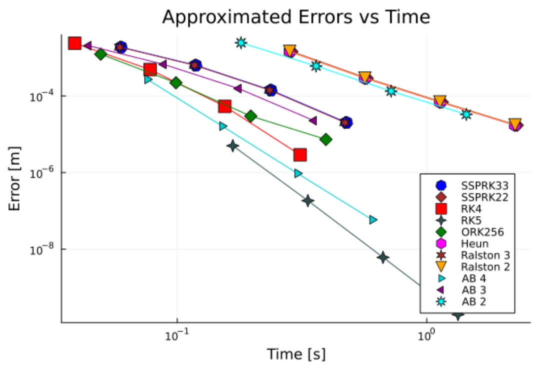
(a)



(b)

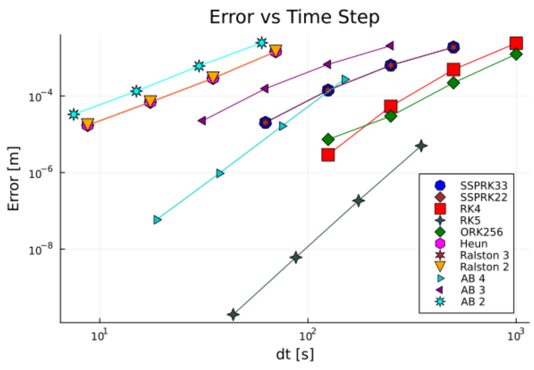


(c)

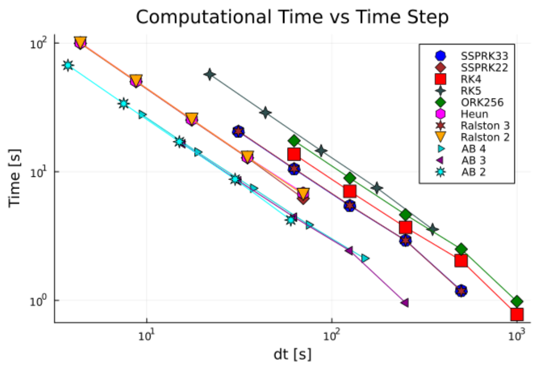


(d)

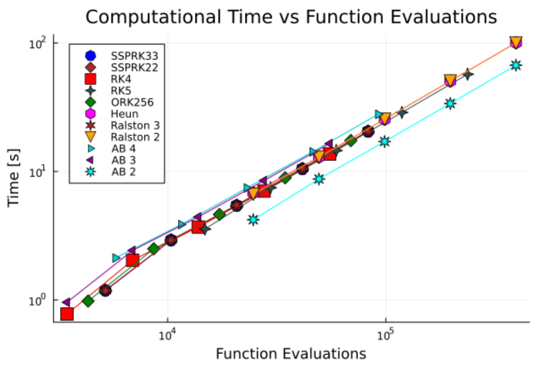
Figure 6.10: Solving SWEs on a CPU numerically, double precision.



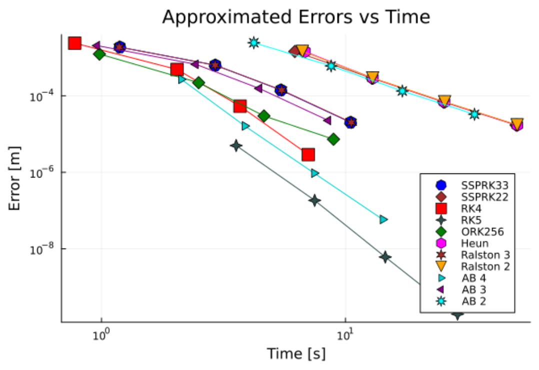
(a)



(b)



(c)



(d)

Figure 6.11: Solving SWEs on a GPU numerically, double precision.

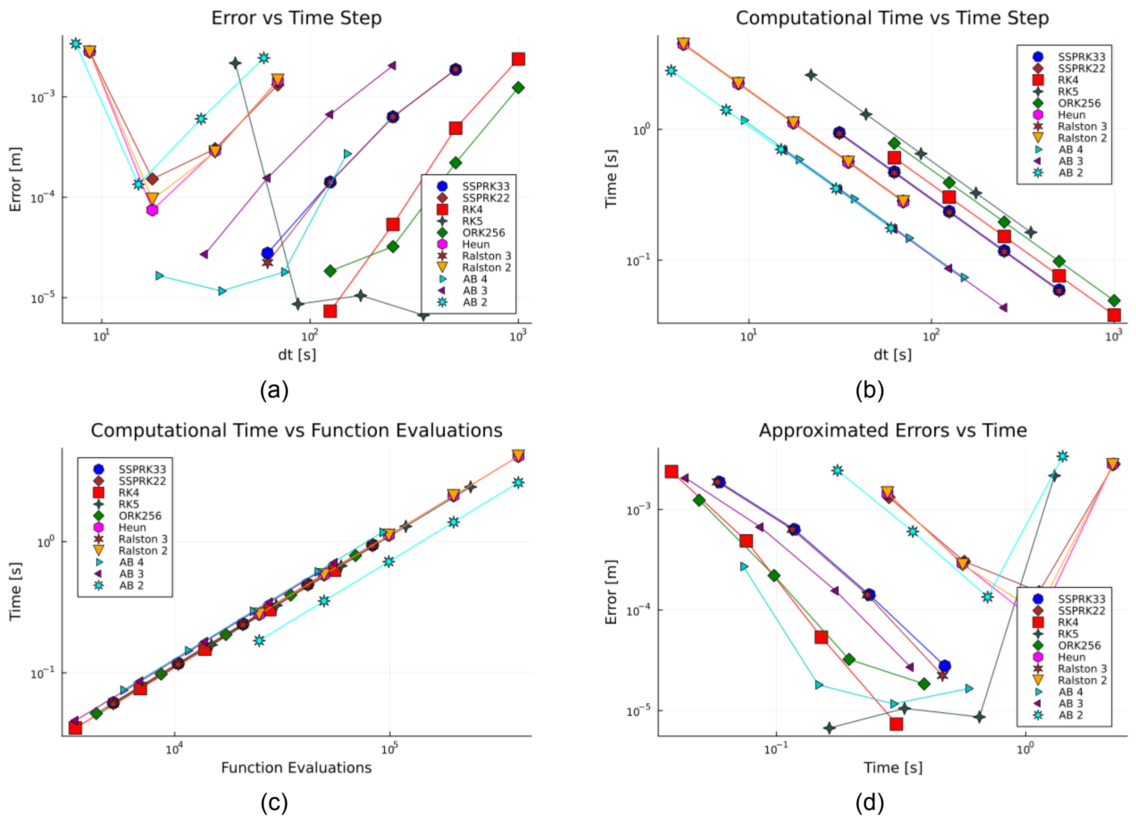


Figure 6.12: Solving SWEs on a CPU numerically, single precision.

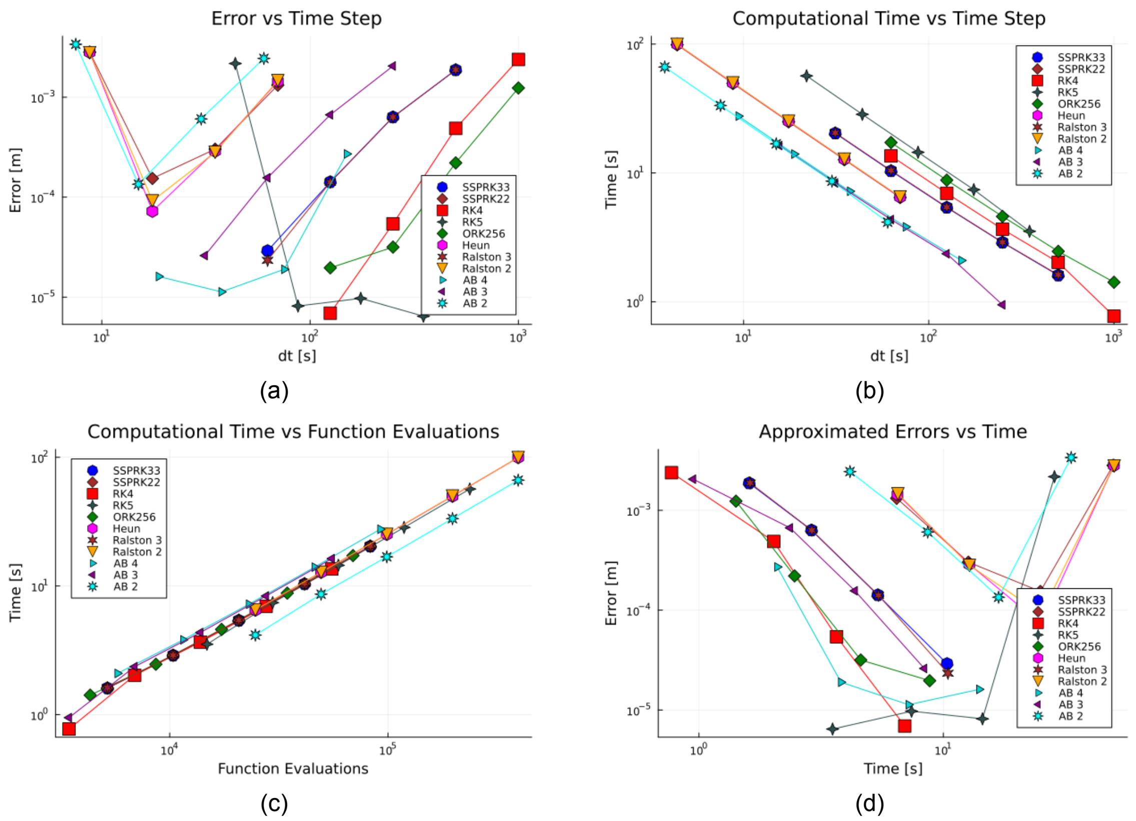


Figure 6.13: Solving SWEs on a GPU numerically, single precision.

6.3. Comparing performance for varying grid-sizes

In the previous subsection, different explicit time-integration schemes were compared for various time-steps on a 20×20 grid. In this subsection, a comparison of different methods in terms of accuracy and performance is presented, testing a variety of time-integration techniques on different grid sizes.

6.3.1. Experiment set-up

The time-dependent ODE, which we solve again for this problem using different time-integration schemes on various grid sizes, is described in Section 6.1.2. For the explicit RK and AB methods, a time-step size is chosen such that it is very close to the largest time step for which stability is maintained. In Table 6.1, these time-step sizes are listed for different methods on a 20×20 grid. The experiments will initially be conducted on a 30×30 grid. Therefore, the time steps for a 30×30 grid can be found by dividing the time-steps in Table 6.1 by 1.5. However, for the second-order Ralston and Adams-Bashforth methods, we eventually found that much smaller time-steps had to be used, employing a trial-and-error approach.

Errors are estimated using Richardson interpolation by computing the differences between numerical solutions on different grids with varying mesh sizes. The error is approximated as follows:

$$\mathbf{H}^{\Delta x} = \frac{(3^2 \mathbf{h}^{\frac{\Delta x}{3}} - \mathbf{h}^{\Delta x})}{3^2 - 1},$$
$$\mathbf{E}^{\Delta x} = \frac{1}{\sqrt{n_x n_y}} ||\mathbf{h}^{\Delta x} - \mathbf{H}^{\Delta x}||_2,$$

where the variables $\mathbf{H}^{\Delta x}$ and $\mathbf{E}^{\Delta x}$, represent a higher-order approximation of a numerical solution and the error-approximation for spatial step-sizes equal to Δx, respectively. Also, the variables $\mathbf{h}^{\frac{\Delta h}{2}}$ and $\mathbf{h}^{\Delta h}$ are the numerical solutions at the last time iteration with spatial-step sizes equal to $\Delta x/2 = \Delta y/2$ and $\Delta x = \Delta y$, respectively. The spatial discretization is second order, due to the use of central differences. Since experiments are done on different grid sizes, the error is divided by the number of total grid points, to only obtain the average error in one grid point.

Furthermore, the experiments are set up in the following way:

- A time span of 3 days is selected;
- For every problem, the simulation starts on a 30×30 grid with the maximum possible time-step. To compare performance, we use 30×30, 90×90, 270×270, and 810×810 grids, i.e., each time the number of grid points in both the x- and y-directions is multiplied by 3. The refinement factor of 3 is chosen so that the points of the fine and coarse grids overlap, eliminating the need for interpolation.
- In the previous point the mesh-sizes Δx and Δy are divided by 3. In addition to dividing the spatial-step by 3, it is necessary to divide the time step by 3 for satisfying the stability-condition. Thus, for a simulation with a fixed time span, the computational workload will increase $27 = 3 \times 3 \times 3$ times. An certain case is where for every computation on a numerical grid solution within an iteration, all the computations on the different grid points can be done in parallel and still a few GPU cores remain idle. In that case where all the GPU cores are not fully occupied, every individual time iteration does not need more time if the grid size increases, since a greater number of GPU cores are working on the larger problem. In that case, he amount of time for the perfect parallel case will only increase by three, since the number of necessary time iterations increases by three. For serial computations on a CPU, the needed time will increase 27 times.
- In Table 6.2 time-steps are given such that the problem is stable on a 30-by-30 grid. These time-steps are computed using the stability conditions. For the second-order AB2 and Heun schemes, instabilities can occur because the eigenvalues of the linearized system around 0 might not lie within the stability regions of AB2 and Heun. Therefore, for these second-order schemes, stability is experimentally verified.
- The GPU experiments were conducted on an H100 (NVIDIA H100 PCIe 80 GB) device. For the CPU experiments, an Intel® Xeon® Platinum 8462Y+ node was used. The Deltares cluster was utilized for these experiments.

- If the interpolated Richardson error is below 1×10^{-3} m, we choose to stop refining the computational grid.

Table 6.2: Largest time-step approximations for different time-integration schemes.

method	dt [s]
RK5	200
RK4	667
Ralston3	300
Ralston2	30
ORK256	667
AB2	15
AB3	140
AB4	100

When the time-step and spatial step sizes are reduced by a factor of 3, the computational workload increases by a factor of 3^3. For serial computations on a CPU, this results in approximately a 27-fold increase in computation time, as both the time-step size and spatial step size are refined by a factor of 3.

In an ideal parallel computing scenario, the computational time per iteration remains constant even as the grid size increases. The total computational time increases only by a factor of 3, corresponding to the threefold increase in the number of required time iterations.

6.3.2. Comparing Performance on a GPU versus CPU for Experiments with Varying Grid-sizes

In Figures 6.14(c), 6.15(c), 6.16(c), and 6.17(c), the simulation time is plotted against the grid size in the x-direction for CPU and GPU experiments in both double and single precision, respectively. The dashed line in these figures displays the simulation time for the experiment on the smallest 30-by-30 grid multiplied by 3^3, for every time the grid is refined by a factor 3 in x- and y-direction. For the cases on the GPU in Figures 6.14 (c) and 6.16(c), the simulation time measurements are below the dashed lines. This can be explained by the parallelism on a GPU. For the serial experiment on a CPU in Figure 6.15 (c), the measurement data is close to the dashed line. Hence, it can be confirmed that for serial CPU-runs the needed time will increase by p^3 times each time we refine the spatial grid by a factor p in both the horizontal and vertical direction.

Moreover, in Figures 6.15 (c) and 6.16 (c) the simulation time is plotted against grid-size for both single precision experiments and double precision experiments on a GPU, respectively. The simulations on a GPU take more time for the double case compared to the single case, as the grid-size increases. To show this more clearly, the running-times of Runge-Kutta 4 for single precision and double precision are shown in table 6.3. The decrease in computational time when switching to single precision is significant, but not enormous.

Table 6.3: Simulation time on a GPU for RK4.

nx	Float32 time [s]	Float64 time [s]
30	0.3375573	0.34060815
90	2.7012186	3.84364
270	5.3289766	6.544757
810	16.20453	19.689924

Figures 6.14 (f), 6.15 (f), 6.16 (f), and 6.17 (f) display the memory throughput as a function of grid size for both CPU and GPU, with results shown for double precision and single precision separately. If Figures 6.14 (f) and 6.16 (f), we do not see an increase in memory throughput on the CPU as, the grid-size increases. In contrast, in Figures 6.15 (f) and 6.17 (f), one can observe that the memory throughput increases as the grid-size increases. As more computations are performed in parallel on a GPU, a

larger amounts of read and write operations are performed simultaneously. Furthermore, a GPU has often a higher bandwidth compared to a CPU. The H100 PCIe 80 GB offers a maximum bandwidth of 2.04 TB/s, while the Xeon Platinum 8268 CPU has a maximum bandwidth of 140.9 GB/s. The CPU experiments in our study, have a much lower memory throughput. However, in our experiments one can still observe that GPU experiments have for high resolution grids a larger bandwidth.

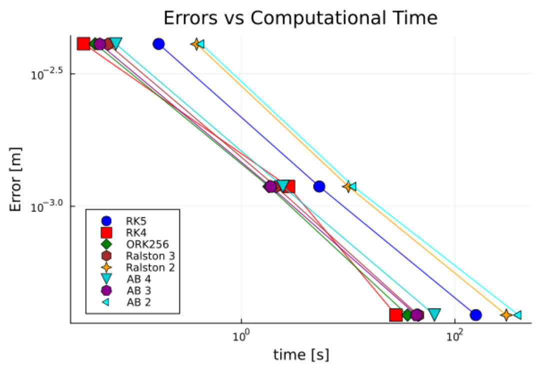
Remark: The memory bandwidth is approximated as follows. We count the number of read and write operations (for dense matrices) per function evaluation. Also we estimate the number of read and write operations within a time-step function corresponding to a scheme from within OrdinaryDiffEq.jl (Rackauckas and Nie, 2017). Furthermore we compute the total number of read and write operations and multiply this by the number of bytes per floating point number. For single precision, for every number we need four bytes, while for double precision, we need 8 bytes. Furthermore, we compute the total number of iterations: $N_t = \frac{\textbf{time span}}{\Delta t}$ and multiply this with the memory access [GB] per iteration. Finally, to compute memory bandwidth, we divide memory access by the total computational time.

Figures 6.14(d), 6.15(d), 6.16(d), and 6.17(d) display the computational time versus the number of function evaluations for both CPU and GPU experiments, with results shown for double precision and single precision. In the figures for the CPU experiments (6.14(d) and 6.16(d)), the computational time increases linearly with the total number of function evaluations in the simulation. In the figures for the GPU experiments (6.15(d) and 6.17(d)), the computational time also increases as the number of function evaluations increases. However, this increase is not as linear as in the CPU case.

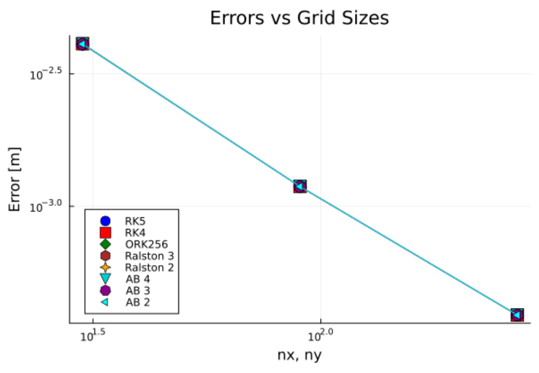
Accuracy and Robustness Figures 6.14 (b), 6.15 (b), 6.16 (b), and 6.17 (b) display the estimated error as a function of grid size for both CPU and GPU-experiments, for double precision and single precision. In Figures 6.14 (b) and 6.15(b), it is evident that increasing both n_y and n_x with a factor 3 results in similar error reductions across different methods. Apparently, spatial errors dominate. In contrast, for single precision experiments shown in Figures 6.16 (b), and 6.17 (b), the second-order methods increase in error as grid size increases (except for ORK256). However, for the single precision experiments depicted in Figures 6.16(b) and 6.17(b), the second-order methods show an increase in error as the grid size grows (except for ORK256). Additionally, the error reductions for third-order methods are smaller compared to those achieved by fourth- and fifth-order schemes. This demonstrates that the ORK256, RK5, RK4, and AB4 methods are the most robust explicit schemes. These methods allow for larger time steps compared to the second- and third-order methods, which can be advantageous in minimizing rounding errors when using Float32 precision. Furthermore, since the stability region for second-order methods is small, these methods are not always stable in our case. As a result, both slight instabilities and accumulating rounding errors contribute to large errors.

Error vs computational time In Figures 6.15(a) and 6.16 (a), 6.17(a) and 6.18(a) the error-approximation is plotted against simulation for both single precision experiments and double precision experiments on a GPU and CPU. For double precision, as simulation time increases, errors do decrease. However, to obtain the same error-reduction with second order Ralston and Adams-Bashforth we need more than four times the amount of simulation time too achieve the same error reduction with, third and fourth order methods. For reducing errors with RK5 we need more than twice the running time compared to ORK256, RK4, RK3, AB3 and AB4-methods. For single precision experiments in Figure 6.15 (a), the same error reductions are not achieved for AB3, RK3, RK2 and AB2-methods as time increases. This is due to rounding errors. For those third and second order methods, an increased simulation time does yields smaller error reductions compared to the double precision experiments in Figure 6.16(a).

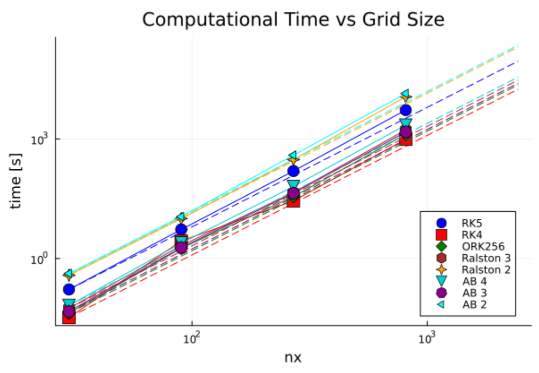
For experiments with double precision on a GPU, the lines of the graph in 6.15 (a) corresponding to RK4-, RK3-, AB3- and ORK256-schemes are the most to the left. Hence, selecting those methods results in the most optimal time-to-accuracy trade-off. However, for single-precision experiments in 6.17(a) and 6.17(b), we observe that for RK3 and AB3 on course grids errors do not converge similar to double precision. Thus, choosing RK4- and ORK256- methods yields the best time versus accuracy trade-off.



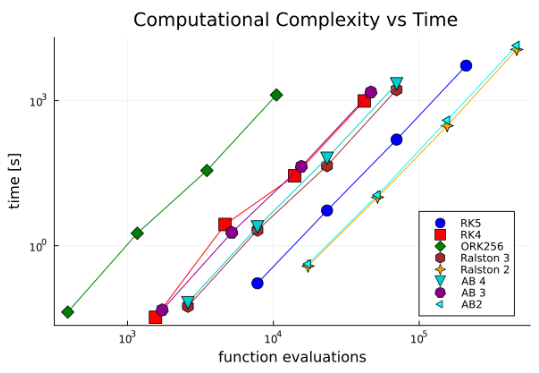
(a)



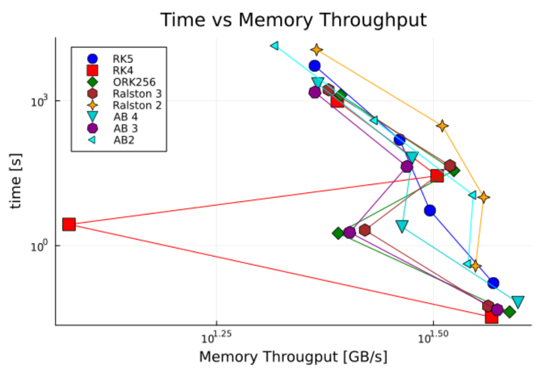
(b)



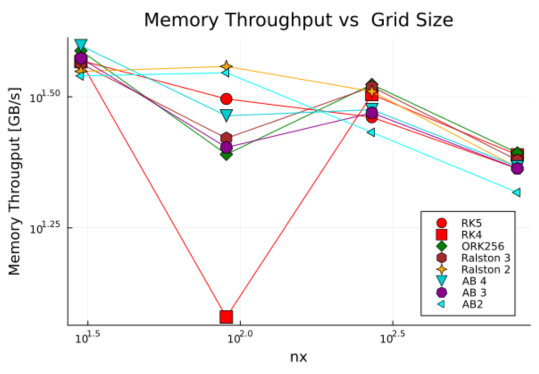
(c)



(d)

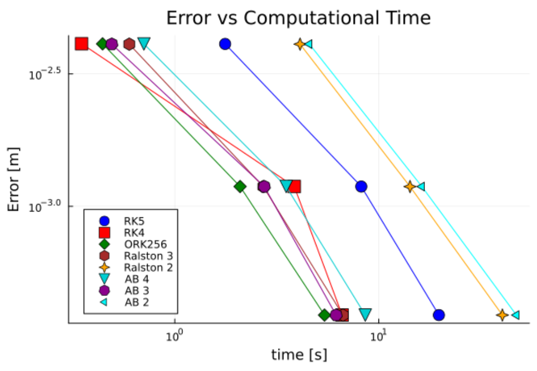


(e)

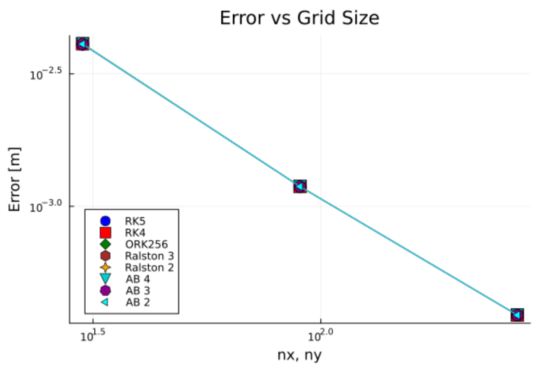


(f)

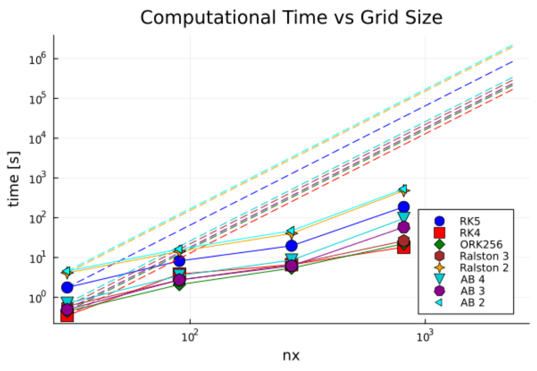
Figure 6.14: Solving SWEs on a CPU numerically, double precision



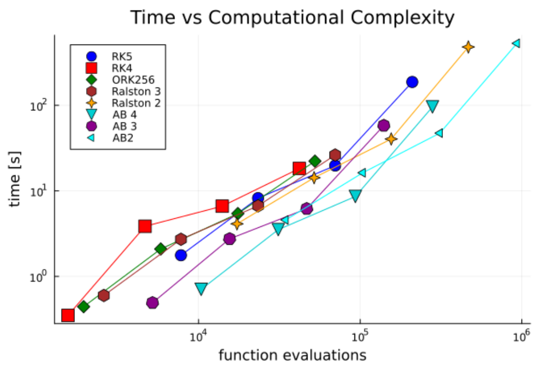
(a)



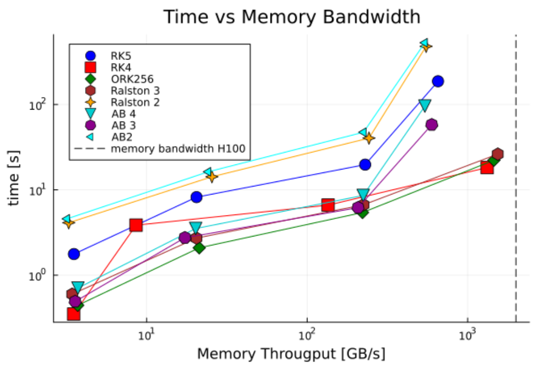
(b)



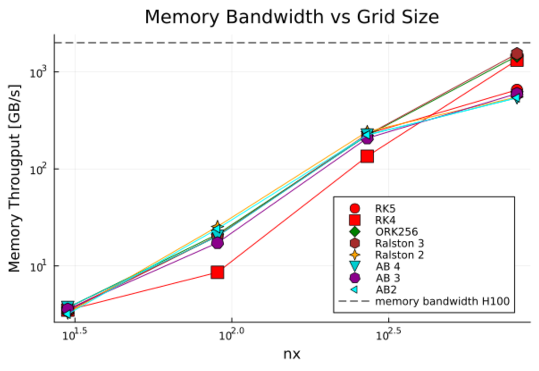
(c)



(d)



(e)



(f)

Figure 6.15: Solving SWEs on a GPU numerically, double precision

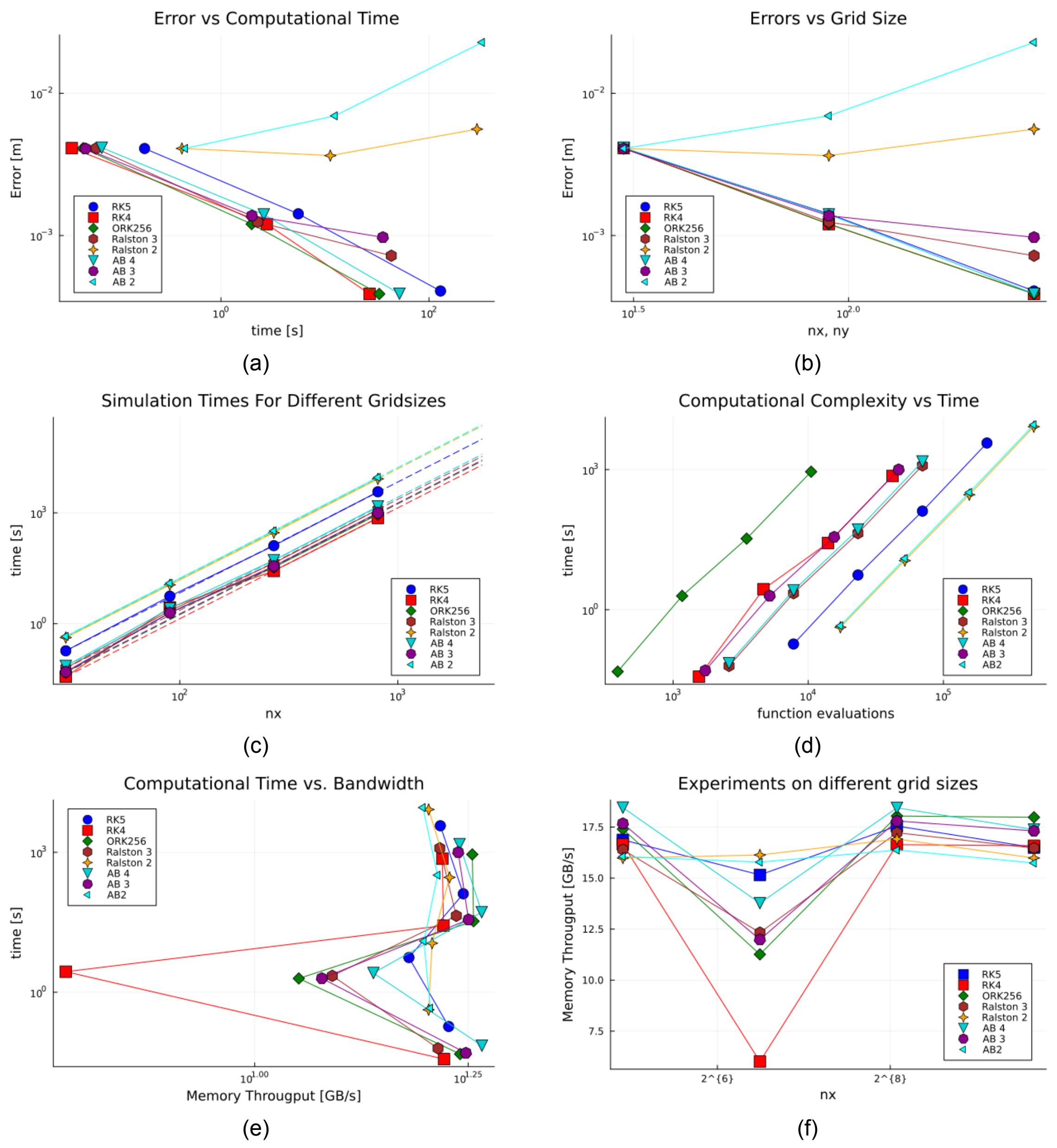
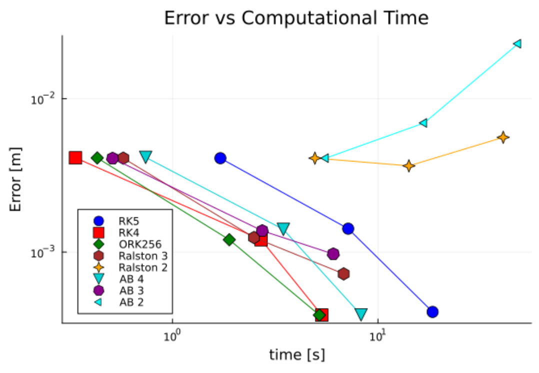
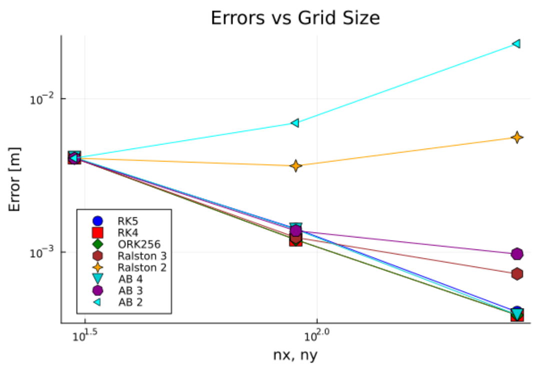


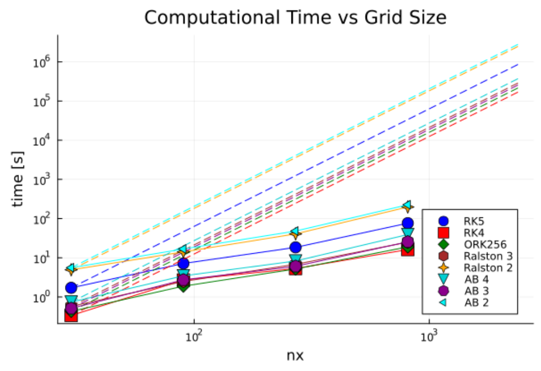
Figure 6.16: Solving SWEs on a CPU numerically, single precision



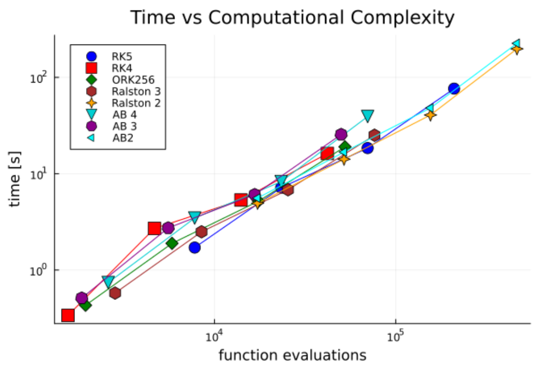
(a)



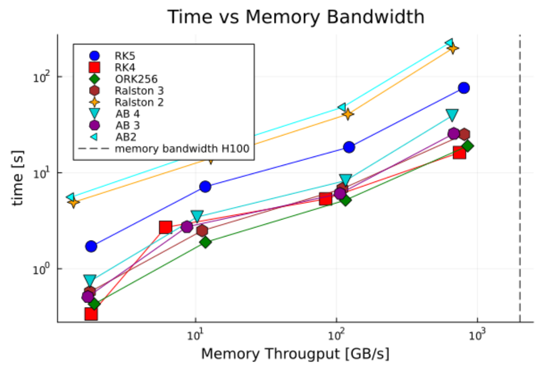
(b)



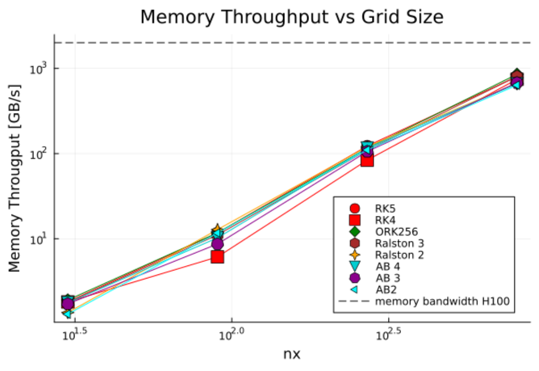
(c)



(d)



(e)



(f)

Figure 6.17: Solving SWEs on a GPU numerically, single precision

6.4. Investigating Parallelism and Memory Usage

To investigate parallelism, experiments were conducted with different grid sizes. Instead of focusing on a specific time span, we tested 10,000 time-integration iterations. Since different methods use different time step sizes, varying time spans were applied for each method. These tests are solely focused on examining parallelism. The RK4-method was selected for these tests.

The previous experiments did not fully use the GPU-memory, due to not having large array sizes i.e. very high resolution grids. In the previous experiments switching to a very high resolution, would have lead to very small time steps and hence we needed more iterations to compute the same time-span. Subsequently, run-time would increase. Hence, for practicality we also choose to perform experiments with a fixed number of iterations, rather than a fixed time span.

These experiments were conducted on a 1/4 A100 card with 40 GB of memory, part of the Snellius cluster. Using the `nvtop` command, we measured both memory usage and the percentage of GPU cores utilized.

We observe the following for different grid sizes:

- For grid sizes with n_x and n_y equal to 30, 90, and 270, the time required for 10,000 iterations remains constant. However, for grid sizes between 405 and 1215, the time per iteration increases sublinearly with the grid size.
- For 1215×1215 grids the GPU capacity is used fully, which entails that all the GPU cores are working to solve the problem. Increasing the grid from that point on will lead in a linear increase of the necessary simulation time per iteration, since the extra work can not be be assigned to additional gpu-cores.
- If one attempts to solve the SWE with a grid size of 14580×14580 or higher in our experiment, we exceed the maximum 40GB memory capacity of the GPU, causing the attempt to solve the ODE to fail.

Furthermore, there is a significant, though not enormous, difference between single and double precision in terms of performance. For example, for a grid with $7290^2 \approx 53144100$ points, it takes 1072 seconds for double precision and 708 seconds for single precision to perform 10,000 iterations.

For double precision, a 9477×9477 grid uses 95% of the GPU memory. In theory, a 9975×9975 grid would fit in memory at most for double precision. For single precision, a 13122×13122 grid uses 91% of the GPU memory, and in theory, a 14419×14419 grid would fit at most. Switching from double to single precision allows us to accommodate a larger grid size on the GPU, with an increase by a factor of 1.44. Hence, switching to single precision allows us to fit a larger grid into the GPU.

For solving the problem on a 7290×7290 grid, a time step of $\Delta t \approx 2.7$ is required, resulting in a simulated time span of approximately $2.7 \times 10000 \approx 7.5$ hours. The computational time is 18 minutes for double precision and 12 minutes for single precision.

Table 6.4: Experiments with RK4 using double precision.

RK4	# nx=ny	dt [s]	time [s]		memory [Mib]	%gpu	%mem gpu
Float64	30	667	15.292088		496	76	1
	90	667/3	15.412181		496	81	1
	270	667/9	15.600637		496	84	1
	405	667/(9*1.5)	16.350397		560	91	1
	540	667/18	18.633648		592	91	1
	810	667/27	25.62598		752	93	2
	1215	667/(27*1.5)	40.494373		1072	98	3
	2430	667/81	133.52763		2992	99	7
	7290	667/243	1072.6831		23184	99	57
	9477	667/(243*1.3)	1840.736		38864	99	95

Table 6.5: Experiments with RK4 using single precision.

RK4	nx=ny	dt [s]	time [s]	memory [Mib]	%gpu	%mem gpu
Float32	30	667	15.315307	496	73	1
	90	667/3	15.642945f0	496	75	1
	270	667/9	15.321502f0	496	86	1
	405	667(9*1.5)	15.461562	528	87	1
	540	667/18	17.214737	528	90	1
	810	667/27	22.043463	624	92	2
	1215	667/(27*1.5)	31.04757	784	96	2
	2430	667/81	89.33011	1744	99	4
	7290	667/243	707.95294	11824	99	29
	13122	667/(243*1.8)	2286.8215	37264	99	91

Discussion Finally, the parallelization of explicit methods for solving on the GPU is effective. For large grids, explicit methods demonstrate significantly faster performance when implemented on the GPU compared to serial implementations. However, a major limitation are the stability conditions, which necessitate very small time steps for fine grid resolutions. As a result more iterations, and therefore more work is needed to simulate a fixed time span.

7 Applying Pseudo Time-Stepping to Implicit Time Iterations

In the previous chapter, SWEs were solved with a variety of different explicit time-integration schemes. The results of the experiments with explicit schemes can be found in Section 6.4. As the computational mesh becomes finer, the computational complexity increases. On one hand, this increase in computational complexity can be explained by the larger number of unknowns that have to be solved. On the other hand, as the grid becomes finer, the cell size decreases and a smaller time step has to be selected to satisfy the stability conditions. Consequently, on fine grids (e.g. 810x810), the maximum allowable time step is often equal to a couple of seconds.

The stability conditions for the time step sizes for explicit methods are often much more restrictive compared to the stability conditions for implicit methods. For example, If we look at different stability profiles in Section 3.5, the stability region for first order implicit Runge-Kutta consist of the entire complex plane, with an exception for a set of z-values with at most a real part and imaginary part equal to 1. In contrast, the first order explicit Euler forward method, the modules of the z-values should at most be equal to one, for stability. Since the stability restrictions on Δt are less strict for implicit methods, we would like to examine how SWEs can be solved, with implicit times-integration schemes.

Applying an implicit time-integration scheme to the SWEs, yields a nonlinear system that must be solved for every time step. This nonlinear system is commonly solved with a Newton-method, which requires a linear system to be solved for every Newton-iteration. However, solving a nonlinear system with a Newton-method can be expensive.

For solving nonlinear systems on a GPU, we would like to have an algorithm with a lower memory cost. Therefore, instead of applying the Newton method for solving the nonlinear ODE, we regard the nonlinear problem at every time step as a stationary problem, and then solve it with the pseudo time-stepping method. To the nonlinear problem a pseudo-time derivative will be added and explicit integration with respect to pseudo-times is used to iterate until an equilibrium solution is found. Because explicit time-integration is not expensive, memory wise, implementing pseudo-time-stepping with an explicit pseudo-time integration scheme, we would like to develop and test a pseudo-time integration scheme on a GPU.

The main objective of this chapter is to incorporate a pseudo-time-stepping technique within an implicit time-stepping scheme, such that it is less time-expensive than explicit methods. In Section 7.1, pseudo time-stepping techniques are applied to different implicit time-integration schemes. In Section 7.2, stability properties are presented for different pseudo-time solvers. Finally, in Section 7.3, the performance and computational costs of different pseudo-time solvers are compared.

7.1. Applying Pseudo-Time-Stepping to Implicit Methods

In this section, we present the process of implementing different pseudo-time techniques to solve a nonlinear system within a time iteration corresponding to various implicit time-integration schemes.

In Section 7.1.1, different implicit time-integration schemes are presented. In Section 7.1.2, a new pseudo-time approach is presented. In Section 7.1.3 the ODEs resulting from the pseudo time-stepping approach, are analyzed with respect to damping.

7.1.1. Implicit Time-Integration Schemes

With implicit time integration schemes an ordinary differential equation similar to Equation (7.1) can be solved.

$$\frac{d\mathbf{x}}{dt} = \mathbf{F}(\mathbf{x}, t) \tag{7.1}$$

In our study four different implicit methods will be examined: the Backward Euler (BE), the Crank-Nickelson (CN), the second-order backward differencing formula (BDF2)-scheme and the second order singly-diagonal implicit Runge-Kutta (SDIRK2)-method.

The Backward Euler Scheme The Backward Euler (BE) method is an implicit first-order method, with a LTE of $\mathcal{O}(\Delta t)$. At time t^{n+1} we have the following solution:

$$\mathbf{x}^{n+1} = \mathbf{x}^n + \Delta t \mathbf{F}(\mathbf{x^{n+1}}, t^{n+1}). \tag{7.2}$$

The Backward Euler (BE) method is A-stable, meaning it remains stable for systems with eigenvalues having non-positive real parts, regardless of the time step size. Additionally, it is L-stable, which ensures that as the scaled eigenvalue (z) becomes very large (and goes to infinity), the stability function approaches zero. This property is particularly advantageous for stiff systems, as it effectively damps out high-frequency components and ensures numerical stability. The combination of A-stability and L-stability makes Backward Euler a robust and reliable method for solving stiff differential equations.

The Crank-Nickelson Method A very common second order method is the Crank-Nickelson (CN) method, shown in (7.3). This scheme requires an additional function evaluation in the previous time step for every time iteration compared to BE (see Equation (7.2)).

$$\mathbf{x}^{n+1} = \mathbf{x}^n + \frac{\Delta t}{2}\left(\mathbf{F}(\mathbf{x^{n+1}}, t^{n+1}) + \mathbf{F}(\mathbf{x}^n, t^n)\right). \tag{7.3}$$

The Crank-Nicolson method is also A-stable. However, if the eigenvalues of $\mathbf{F}$ have a positive real part, CN is always unstable. Therefore, this method is not L-stable.

The BDF2-Method The BDF2-method is an implicit linear multi-step method. At time t^{n+1} the numerical solution is given by Equation (7.4).

$$\mathbf{x}^{n+1} = \frac{4}{3}\mathbf{x}^n - \frac{1}{2}\mathbf{x}^{n-1} + \frac{2\Delta t}{3}\mathbf{F}(\mathbf{x}^{n+1}, t^{n+1}). \tag{7.4}$$

The BDF2-method is A-stable and L-stable. In our implemenation we compute the first time-step using Crank-Nicolson.

The SDIRK2 method The SDIRK2-method is an implicit Runge-Kutta method with 2 stages and has a triangular Butcher tableau. Hence, in every time iteration, two nonlinear systems have to be solved for the SDIRK2-method. The numerical solution $\mathbf{x}^{n+1}$, in the next time step is obtained in three steps. The SDIRK2 method used in our study is described in Kennedy and Carpenter, 2016 (p. 72).

Firstly, Equation (7.5) is solved for the stage-variable $\mathbf{k}_1$.

$$\mathbf{k}_1 = \mathbf{F}(\mathbf{x}^n + \gamma \Delta t \mathbf{k}_1, t + \gamma \Delta t). \tag{7.5}$$

Secondly, Equation (7.6) is solved for variable $\mathbf{k}_2$.

$$\mathbf{k}_2 = \mathbf{F}(\mathbf{x}^n + \Delta t\left((1-\gamma)\mathbf{k}_1 + \gamma\mathbf{k}_2\right), t + \Delta t). \tag{7.6}$$

Finally, in Equation (7.7) a weighted average is computed between the two stages to obtain the numerical solution at the next time step.

$$\mathbf{x}^{n+1} = \mathbf{x}^n + \Delta t((1-\gamma)\mathbf{k}_1 + \gamma\mathbf{k}_2), \tag{7.7}$$

where constant γ is chosen to be equal to $1 - \frac{1}{\sqrt{2}}$.

For this method both A- and L- stability holds.

Time-Dependent problem

Using the implicit time integration schemes, we would like to solve time-dependent problems for Shallow Water Flows. This problem is given by Equations (6.12)- (6.14) in the previous chapter. For the pressure terms we use the same central-difference discretization and for the Corliolis force we use the same averaging.

We define $H = D + h$, $p = Hu$ and $q = Hv$. For the sake of simplicity, we introduce the following notational convention to represent Equations (6.12)–(6.14):

$$\frac{\partial h}{\partial t} = f_h(p) + g_h(q), \tag{7.8}$$

$$\frac{\partial p}{\partial t} = f_p(h) + g_p(q) + v_p(h,p,q) + f_{tide}(h,t), \tag{7.9}$$

$$\frac{\partial q}{\partial t} = f_q(h) + g_q(p) + v_q(h,p,q). \tag{7.10}$$

where,

$$f_h(p) = -\frac{\partial p}{\partial x}, \quad g_h(q) = -\frac{\partial q}{\partial y},$$

$$f_p(h) = -gH\frac{\partial h}{\partial x}, \quad g_p(q) = fq, \quad v_p(h,p,q) = -c_f H^{-2} p\sqrt{q^2+p^2}, \quad f_{tide}(h,t) = -HAsin(m_2 t),$$

$$f_q(h) = -gH\frac{\partial h}{\partial y}, \quad g_p(p) = -fp, \quad v_p(h,p,q) = -c_f H^{-2} q\sqrt{q^2+p^2}.$$

The Equations (7.8)-(7.10) represent the test case that will be solved in this chapter.

7.1.2. Pseudo-Time-Stepping Techniques to Various Implicit Methods

In this subsection a pseudo-time method is applied to the implicit Backward Euler, Crank-Nicolson, BDF2 and SDIRK2 scheme, respectively. The unknowns in Equations (7.2)-(7.6) are made dependent on a time-variable τ and a τ-dependent ODE is defined such that the stationary solution of the ODE is an approximation of the unknowns.

In our study we use the abbreviations **BE**, **CN**, **BDF2** and **SDIRK2** to indicate that the Backwards Euler, Crank Nicolson, BDF2, and SDIRK2 methods were selected as outer time-schemes, respectively. Furthermore, we with **IM**, **IMEX**, and **FW**, we denote the inner time-integrations schemes: explicit, semi-explicit and forward-backward, respectively. With **FW**, we indicate that we only approximated h as a pseudo-time variable.

In the appendix B, we show how pseudo-time-stepping can be used to solve the nonlinear systems arising form BDF2- and CN- iterations.

Applying Pseudo-Time-Stepping to the Backward Euler Scheme

Applying the Backward Euler integration method to the test problem defined by Equations (7.8)–(7.10) results in:

$$h^{n+1} = h^n + \Delta t(f_h(p^{n+1}) + g_h(q^{n+1}), \tag{7.11}$$

$$p^{n+1} = p^n + \Delta t\left(f_p(h^{n+1}) + g_p(q^{n+1}) + v_p(h^{n+1}, p^{n+1}, q^{n+1}) + f_{tide}(h^{n+1}, t^{n+1})\right), \tag{7.12}$$

$$q^{n+1} = q^n + \Delta t(f_q(h^{n+1}) + g_q(p^{n+1}) + v_q(h^{n+1}, p^{n+1}, q^{n+1})). \tag{7.13}$$

where h^{n+1}, p^{n+1} and q^{n+1} are the values of h, p and q at time: $t^{n+1} = \Delta t(n+1)$. To obtain these unknowns, a nonlinear system has to be solved. Instead of using Newton's method to solve this nonlinear system, we apply pseudo-time-stepping to Equations (7.11)-(7.13). In this subsection this approach is outlined, by explaining how one can approximate the values of h^{n+1}, p^{n+1} and q^{n+1}, by redefining Problem (7.11)-(7.13).

One can approximate the values h^{n+1}, p^{n+1} and q^{n+1} in Equations (7.11)-(7.13), by introducing three pseudo-time variables ξ_h, ξ_p and ξ_q, respectively.

Hence, the root of the following functions has to be found.

$$G_1(\xi_h, \xi_p, \xi_q) = (f_h(\xi_p) + g_h(\xi_q)) - \frac{\xi_h - h^n}{\Delta t} = 0, \tag{7.14}$$

$$G_2(\xi_h, \xi_p, \xi_q) = \left(f_p(\xi_h) + g_p(\xi_q) + v_p(\xi_h, \xi_p, \xi_q) + f_{tide}(\xi_h, t^{n+1})\right) - \frac{\xi_p - p^n}{\Delta t} = 0, \tag{7.15}$$

$$G_3(\xi_h, \xi_p, \xi_q) = (f_q(\xi_h) + g_q(\xi_p + v_q(\xi_h, \xi_p, \xi_q)) - \frac{\xi_q - q^n}{\Delta t} = 0. \tag{7.16}$$

Furthermore, the variables ξ_h, ξ_p, and ξ_q are made dependent on the pseudo-time variable τ. Pseudo-time derivatives of ξ_h, ξ_p, and ξ_q are incorporated into the systems G_1, G_2, and G_3. The resulting pseudo-time system is a τ-dependent ODE, as presented in Equation (7.17).

$$\begin{bmatrix} \frac{\partial \xi_h}{\partial \tau} \\ \frac{\partial \xi_p}{\partial \tau} \\ \frac{\partial \xi_q}{\partial \tau} \end{bmatrix} = \begin{bmatrix} G_1(\xi_h, \xi_p, \xi_q) \\ G_2(\xi_h, \xi_p, \xi_q) \\ G_3(\xi_h, \xi_p, \xi_q) \end{bmatrix} = \mathbf{0}. \tag{7.17}$$

The stationary solution of Equation (7.17), is the root of the Equations (7.14)-(7.16) and represents the solutions at time t^{n+1}:

$$\lim_{\tau \to \infty} \xi_h(\tau) = h^{n+1}, \quad \lim_{\tau \to \infty} \xi_p(\tau) = p^{n+1}, \quad \lim_{\tau \to \infty} \xi_q(\tau) = q^{n+1}.$$

This equilibrium solution is obtained by numerically solving Equation (7.17). A time-integration technique with respect to pseudo-time is applied, and iteration continues until the roots of Equations (7.14)–(7.16) are found.

BE_IM Applying Explicit Pseudo-Time-Integration For solving Equation (7.17) numerically an explicit Forward Euler method is used. For finding the equilibrium state of the pseudo-ODE, we iterate till we have found τ-independent solutions.

Note, in our study we choose to implicitly integrate the $\frac{\xi_\mathbf{h}}{\Delta t}$, $\frac{\xi_\mathbf{p}}{\Delta t}$ and $\frac{\xi_\mathbf{q}}{\Delta t}$-components from Equations (7.14), (7.15) and (7.16), respectively.

The pseudo-time iterations for solving (7.17) are given by:

$$\xi_h^{k+1} = \xi_h^k + \Delta\tau \left(\left(f_h(\xi_p^k) + g_h(\xi_q^k)\right) - \frac{\xi_h^{k+1} - h^n}{\Delta t} \right),$$

$$\xi_p^{k+1} = \xi_p^k + \Delta\tau \left(\left(f_p(\xi_h^k) + g_p(\xi_q^k) + v_p(\xi_h^k, \xi_p^k, \xi_q^k) + f_{tide}(\xi_h^k, t^{n+1})\right) - \frac{\xi_p^{k+1} - p^n}{\Delta t} \right),$$

$$\xi_q^{k+1} = \xi_q^k + \Delta\tau \left(\left(f_q(\xi_h^k) + g_q(\xi_p^k) + v_q(\xi_h^k, \xi_p^k, \xi_q^k)\right) - \frac{\xi_q^{k+1} - q^n}{\Delta t} \right),$$

where $\Delta\tau$ represents the pseudo-time step, and ξ_h^{k+1}, ξ_p^{k+1}, and ξ_q^{k+1} are numerical approximations of $\xi_h(\tau)$, $\xi_p(\tau)$, and $\xi_q(\tau)$ at the pseudo-time $\tau = (k+1)\Delta\tau$, respectively.

The initial approximations, ξ_h^0, ξ_p^0, and ξ_q^0, are set to h^n, p^n, and q^n, respectively. These initial approximations correspond to the solutions computed in the previous outer time iteration at time $t^n = n\Delta t$.

Since our goal is to iterate till an equilibrium of (7.17) is found, we define the following stopping criterion:

$$||G_1(\xi_h^{k+1}, \xi_p^{k+1}, \xi_q^{k+1})||_\infty < 1e - 8.$$

If this stopping criterion is satisfied, the solutions for the next time step, i.e., at t^{n+1}, are given by:

$$h^{n+1} = \xi_h^{k+1}, \quad ,p^{n+1} = \xi_p^{k+1}, \quad q^{n+1} = \xi_q^{k+1}.$$

BE_IMEX Applying Semi-Explicit Pseudo-Time-Integration In the fully explicit pseudo-time solver, pseudo-time values are updated using function evaluations based on values computed in the previous pseudo-time step. To enhance the stability properties of the pseudo-time solver, an alternative approach can be employed: first, update the ξ_p^{k+1} variable; second, use this updated value to compute ξ_q^{k+1}; and finally, perform the update for ξ_h^{k+1} fully implicitly, utilizing function evaluations based on the most recent updates of ξ_p^{k+1} and ξ_q^{k+1}. In this procedure, ξ_p^{k+1} is updated first, followed by ξ_q^{k+1}.

To ensure symmetry between the updates of ξ_p and ξ_q, ξ_q^{k+2} is updated at the next pseudo-time step, $k+2$, using the values of ξ_h^{k+1}, ξ_p^{k+1}, and ξ_q^{k+1}. Subsequently, ξ_p^{k+2} is updated using the latest values of ξ_h^{k+1}, ξ_p^{k+1}, and ξ_q^{k+2}. Finally, ξ_h^{k+2} is computed fully implicitly.

the above procedures i.e. the semi-explicit iteration scheme for solving the Equation (7.17) has the following structure:

$$\xi_p^{k+1} = \xi_p^k + \Delta\tau \left(\left(f_p(\xi_h^k) + g_p(\xi_q^k) + v_p(\xi_h^k, \xi_p^k, \xi_q^k) + f_{tide}(\xi_h^k, t^{n+1})\right) - \frac{\xi_p^{k+1} - p^n}{\Delta t} \right),$$

$$\xi_q^{k+1} = \xi_q^k + \Delta\tau \left(\left(f_q(\xi_h^k) + g_q(\xi_p^{k+1}) + v_q(\xi_h^k, \xi_p^{k+1}, \xi_q^k)\right) - \frac{\xi_q^{k+1} - q^n}{\Delta t} \right),$$

$$\xi_h^{k+1} = \xi_h^k + \Delta\tau \left(\left(f_h(\xi_p^{k+1}) + g_h(\xi_q^{k+1})\right) - \frac{\xi_h^{k+1} - h^n}{\Delta t} \right),$$

$$\xi_q^{k+2} = \xi_q^{k+1} + \Delta\tau \left(\left(f_q(\xi_h^{k+1}) + g_q(\xi_p^{k+1}) + v_q(\xi_h^{k+1}, \xi_p^{k+1}, \xi_q^{k+1})\right) - \frac{\xi_q^{k+2} - q^n}{\Delta t} \right),$$

$$\xi_p^{k+2} = \xi_p^{k+1} + \Delta\tau \left(\left(f_p(\xi_h^{k+1}) + g_p(\xi_q^{k+2}) + v_p(\xi_h^{k+1}, \xi_p^{k+1}, \xi_q^{k+2}) + f_{tide}(\xi_h^{k+1}, t^{n+1})\right) - \frac{\xi_p^{k+2} - p^n}{\Delta t} \right),$$

$$\xi_h^{k+2} = \xi_h^{k+1} + \Delta\tau \left(\left(f_h(\xi_p^{k+2}) + g_h(\xi_q^{k+2})\right) - \frac{\xi_h^{k+2} - h^n}{\Delta t} \right).$$

BDF2_IMEX: Applying Semi-Explicit Pseudo-Time-Integration In the fully explicit pseudo-time solver we updated the pseudo-time values by function evaluations with values computed in the previous pseudo-time step. To improve the stability properties of the solver, one approach is to first update only the ξ_p^{k+1} variable and use this value to update ξ_q^{k+1}. Subsequently, the update for ξ_h^{k+1} can be performed fully implicitly, relying solely on function evaluations of ξ_p^{k+1} and ξ_q^{k+1}.

To ensure symmetry between the updates of ξ_p and ξ_q, the value of ξ_q^{k+2} is updated first, using the values of ξ_h^{k+1}, ξ_p^{k+1}, and ξ_q^{k+1}. Subsequently, ξ_p^{k+2} is updated with the most recent updates of ξ_h^{k+1}, ξ_p^{k+1}, and ξ_q^{k+2}. Finally, the update for ξ_h^{k+2} is performed fully implicitly. The schemes are presented below:

$$\xi_p^{k+1} = \xi_p^k + \Delta\tau \left(\frac{2}{3} F_p(\xi_h^k, \xi_p^k, \xi_q^k, t^{n+1}) - \frac{\xi_p^{k+1} - \frac{4}{3}p^n + \frac{1}{2}p^{n-1}}{\Delta t} \right), \tag{7.18}$$

$$\xi_q^{k+1} = \xi_q^k + \Delta\tau \left(\frac{2}{3} F_q(\xi_h^k, \xi_p^{k+1}, \xi_q^k) - \frac{\xi_q^{k+1} - \frac{4}{3}q^n + \frac{1}{2}q^{n-1}}{\Delta t} \right), \tag{7.19}$$

$$\xi_h^{k+1} = \xi_h^k + \Delta\tau \left(\frac{2}{3} \left(F_h(\xi_p^{k+1}) + G_h(\xi_q^{k+1}) \right) - \frac{\xi_h^{k+1} - \frac{4}{3}h^n + \frac{1}{2}h^{n-1}}{\Delta t} \right). \tag{7.20}$$

$$\xi_q^{k+2} = \xi_q^{k+1} + \Delta\tau \left(\frac{2}{3} F_q(\xi_h^{k+1}, \xi_p^{k+1}, \xi_q^{k+1}) - \frac{\xi_q^{k+2} - \frac{4}{3}q^n + \frac{1}{2}q^{n-1}}{\Delta t} \right), \tag{7.21}$$

$$\xi_p^{k+2} = \xi_p^{k+1} + \Delta\tau \left(\frac{2}{3} F_p(\xi_h^{k+1}, \xi_p^{k+1}, \xi_q^{k+2}, t^{n+1}) - \frac{\xi_p^{k+2} - \frac{4}{3}p^n + \frac{1}{2}p^{n-1}}{\Delta t} \right), \tag{7.22}$$

$$\xi_h^{k+2} = \xi_h^{k+1} + \Delta\tau \left(\frac{2}{3} \left(F_h(\xi_p^{k+2}) + G_h(\xi_q^{k+2}) \right) - \frac{\xi_h^{k+2} - \frac{4}{3}h^n + \frac{1}{2}h^{n-1}}{\Delta t} \right). \tag{7.23}$$

$$\tag{7.24}$$

Applying Pseudo-Time-Stepping to the SDIRK2-Scheme

In this subsection the two-stage SDIRK time-integration scheme given by Equations (7.5)-(7.7) will be used to solve the SWEs given by Equations (B.1)-(B.3). The two nonlinear stages in the SDIRK-method will be solved using a pseudo-time approach.

Applying a second-order implicit SDIRK2 scheme to Equations (B.1)–(B.3) involves the following two steps. First, the stages $k_{1,h}$, $k_{1,p}$, and $k_{1,q}$ are defined. These stage values are computed by solving the following nonlinear system:

$$k_{1,h} = F_h(p^n + \gamma\Delta t k_{1,p}) + G_h(q^n + \gamma\Delta t k_{1,q}), \tag{7.25}$$

$$k_{1,p} = F_p(h^n + \gamma\Delta t k_{1,h}, p^n + \gamma\Delta t k_{1,p}, q^n + \gamma\Delta t k_{1,q}, t^n + \gamma\Delta t), \tag{7.26}$$

$$k_{1,q} = F_q(h^n + \gamma\Delta t k_{1,h}, p^n + \gamma\Delta t k_{1,p}, q^n + \gamma\Delta t k_{1,q}), \tag{7.27}$$

where constant γ is selected such that L-stability holds and is equal to $\frac{(2-\sqrt{2})}{2}$.

Secondly, the second stages $k_{2,h}$, $k_{2,p}$, and $k_{2,q}$ are defined by the nonlinear system given in Equations (7.28)–(7.31). To solve this system, we first substitute the first-stage values, which are obtained by solving Equations (7.25)–(7.27). The stages $k_{2,h}$, $k_{2,p}$, and $k_{2,q}$ can then be computed by solving the following system:

$$k_{2,h} = F_h(p^n + \Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p})) + G_h(q^n + \Delta t((1-\gamma)k_{1,q} + \gamma k_{2,q}))), \tag{7.28}$$

$$k_{2,p} = F_p(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma k_{2,h}))), \tag{7.29}$$

$$p^n + \Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p})), q^n + \Delta t((1-\gamma)k_{1,q} + \gamma k_{2,q}), t^n + \Delta t), \tag{7.30}$$

$$k_{2,q} = F_q(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma k_{2,h}))), p^n + \gamma\Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p})), q^n + \Delta t((1-\gamma)k_{1,q} + \gamma k_{2,q}),). \tag{7.31}$$

Finally, the value for the next time step iteration can be computed by adding the stages:

$$\begin{aligned}
h^{n+1} &= h^n + \Delta t((1-\gamma)k_{1,h} + \gamma k_{2,h})),\\
p^{n+1} &= p^n + \Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p})),\\
q^{n+1} &= q^n + \Delta t((1-\gamma)k_{1,q} + \gamma k_{2,q})).
\end{aligned}$$

Approximating the First SDIRK2 stage with a Pseudo-Time Approach One can find the unknown values $k_{1,h}$, $k_{1,p}$ and $k_{1,q}$ in Equations (7.25)-(7.27), by introducing three pseudo-time variables. We can approximate $k_{1,h}$, $k_{1,p}$ and $k_{1,q}$, by $\xi_{1,h}$, $\xi_{1,p}$ and $\xi_{1,q}$ in the Equations (7.25)-(7.27), respectively. Additionally we scale the equations with $\frac{1}{\Delta t}$ in (7.25)-(7.27), to have the same unit as the pseudo-time equations corresponding to the Backward Euler, Crank-Nicolson and BDF2-methods. Furthermore, we let the values of $\xi_{1,h}$, $\xi_{1,p}$ and $\xi_{1,q}$, depend on τ and add derivatives of the pseudo-time dependent variables to the equations for the stages.

$$\frac{\partial \xi_{1,h}}{\partial \tau} = \frac{1}{\Delta t}F_h(p^n + \gamma\Delta t\xi_{1,p}) + \frac{1}{\Delta t}G_h(q^n + \gamma\Delta t\xi_{1,q}) - \frac{1}{\Delta t}\xi_{1,h}, \tag{7.32}$$

$$\frac{\partial \xi_{1,p}}{\partial \tau} = \frac{1}{\Delta t}F_p(h^n + \gamma\Delta t\xi_{1,h}, p^n + \gamma\Delta t\xi_{1,p}, q^n + \gamma\Delta t\xi_{1,q}, t^n + \gamma\Delta t) - \frac{1}{\Delta t}\xi_{1,p}, \tag{7.33}$$

$$\frac{\partial \xi_{1,q}}{\partial \tau} = \frac{1}{\Delta t}F_q(h^n + \gamma\Delta t\xi_{1,h}, p^n + \gamma\Delta t\xi_{1,p}, q^n + \gamma\Delta t\xi_{1,q}) - \frac{1}{\Delta t}\xi_{1,q}. \tag{7.34}$$

The stationary solution of Equations (7.32)-(7.34) are the solutions of the stages in Equations (7.25)-(7.27):

$$\lim_{\tau\to\infty}\xi_{1,h}(\tau) = k_{1,h}, \quad \lim_{\tau\to\infty}\xi_{1,p}(\tau) = k_{1,p}, \quad \lim_{\tau\to\infty}\xi_{1,q}(\tau) = k_{1,q}.$$

To find the equilibrium solutions of (7.32)-(7.34), are found by utilizing a numerical time integration scheme and performing pseudo-time iterations till an equilibrium state is found.

SDIRK2_IM: Solving the Pseudo-ODE for the first stage value with Forward Euler Pseudo-time Equations (7.32)-(7.34) are solved by applying the Euler forward scheme and implicitly integrating the $\frac{\xi}{\Delta t}$-terms. With the following pseudo-time iterations:

$$\xi_{1,h}^{k+1} = \xi_{1,h}^k + \frac{\Delta\tau}{\Delta t}(F_h(p^n + \gamma\Delta t\xi_{1,p}^k) + G_h(q^n + \gamma\Delta t\xi_{1,q}^k) - \xi_{1,h}^{k+1}), \tag{7.35}$$

$$\xi_{1,p}^{k+1} = \xi_{1,p}^k + \frac{\Delta\tau}{\Delta t}(F_p(h^n + \gamma\Delta t\xi_{1,h}^k, p^n + \gamma\Delta t xi_{1,p}^k, q^n + \gamma\Delta t\xi_{1,q}^k, t^n + \gamma\Delta t) - \xi_{1,p}^{k+1}), \tag{7.36}$$

$$\xi_{1,q}^{k+1} = \xi_{1,q}^k + \frac{\Delta\tau}{\Delta t}(F_q(h^n + \gamma\Delta t\xi_{1,h}^k, p^n + \gamma\Delta t\xi_{1,p}^k, q^n + \gamma\Delta t\xi_{1,q}^k) - \xi_{1,q}^{k+1}). \tag{7.37}$$

where $\Delta\tau$ is the pseudo-time step and $\xi_{1,h}^{k+1}$, $\xi_{1,p}^{k+1}$ and $\xi_{1,q}^{k+1}$ are numerical approximations of $\xi_{1,h}(\tau)$, $\xi_{1,p}(\tau)$ and $\xi_{1,q}(\tau)$ at $\tau = (k+1)\Delta\tau$, respectively.

The first approximations for $\xi^0_{1,h}$, $\xi^0_{1,p}$ and $\xi^0_{1,q}$ are the stages $\bar{k}_{1,h}$,$\bar{k}_{1,p}$,$\bar{k}_{1,q}$, that were computed in the previous outer time iteration at time t^n, while solving the systems:

$$\begin{aligned}\bar{k}_{1,h} &= F_h(p^{n-1} + \gamma\Delta t\bar{k}_{1,p}) + G_h(q^{n-1} + \gamma\Delta t\bar{k}_{1,q}),\\ \bar{k}_{1,p} &= F_p(h^{n+1} + \gamma\Delta t\bar{k}_{1,h}, p^{n+1} + \gamma\Delta t\bar{k}_{1,p}, q^{n+1} + \gamma\Delta t\bar{k}_{1,q}, (t^n - \Delta t) + \gamma\Delta t),\\ \bar{k}_{1,q} &= F_q(h^{n-1} + \gamma\Delta t\bar{k}_{1,h}, p^{n-1} + \gamma\Delta t\bar{k}_{1,p}, q^{n-1} + \gamma\Delta t\bar{k}_{1,q}).\end{aligned}$$

Since our objective is to iterate till an equilibrium of (7.32)-(7.34) is found, we define the following stopping criterion:

$$||F_h(p^n + \gamma\Delta t\xi^k_{1,p}) + G_h(q^n + \gamma\Delta t\xi^k_{1,q}) - \xi^{k+1}_{1,h}||_\infty < 1e-8.$$

If this stopping criterion is met, the first stage in Equation (7.25)-(7.27) is updated by:

$$k_{1,h} = \xi^{k+1}_{1,h}, \quad , k_{1,p} = \xi^{k+1}_{1,p}, \quad k_{1,q} = \xi^{k+1}_{1,q}.$$

Furthermore, in our implementation, we store this stage and use it as the initial approximation for the first-stage pseudo-time variable while computing the values in the $(n+2)$-th time iteration.

Approximating the second SDIRK2-stage with a Pseudo-Time Approach One can find the unknown values $k_{2,h}$, $k_{2,p}$ and $k_{2,q}$ in Equations (7.28)-(7.31), by introducing three pseudo-time variables. We can approximate $k_{2,h}$, $k_{2,p}$ and $k_{2,q}$, by $\xi_{2,h}$, $\xi_{2,p}$ and $\xi_{2,q}$ in the Equations (7.28)-(7.31), respectively. Similarly to the first stage, we divide (7.25)-(7.27) by Δt. Furthermore, we let the values of $\xi_{2,h}$, $\xi_{2,p}$ and $\xi_{2,q}$, depend on τ and add derivatives of the pseudo-time dependent variables to the equations for the stages.

This pseudo-time equation is defined by:

$$\frac{\partial\xi_{2,h}}{\partial\tau} = \frac{1}{\Delta t}F_h(p^n + \Delta t((1-\gamma)k_{1,p} + \gamma\xi_{2,p})) + G_h(q^n + \Delta t((1-\gamma)k_{1,q} + \gamma\xi_{2,q}))) - \frac{1}{\Delta t}\xi_{2,h}, \tag{7.38}$$

$$\frac{\partial\xi_{2,p}}{\partial\tau} = \frac{1}{\Delta t}F_p(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_{2,h}))), p^n + \Delta t((1-\gamma)k_{1,p} + \gamma\xi_{2,p})), q^n + \Delta t((1-\gamma)k_{1,q} + \gamma\xi_{2,q}), t^n + \Delta t) - \frac{1}{\Delta t}\xi_{2,p}, \tag{7.39}$$

$$\frac{\partial\xi_{2,q}}{\partial\tau} = \frac{1}{\Delta t}F_q(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_{2,h}))), p^n + \gamma\Delta t((1-\gamma)k_{1,p} + \gamma\xi_{2,p})), q^n + \Delta t((1-\gamma)k_{1,q} + \gamma\xi_{2,q})) - \frac{1}{\Delta t}\xi_{2,q}. \tag{7.40}$$

The values of $k_{1,h}$, $k_{1,p}$, and $k_{1,q}$ have already been computed using a pseudo-time approach in the previous subsection. Therefore, these stages are substituted into Equations (7.38)–(7.40).

The equilibrium solution of Equations (7.38)-(7.40) are the solutions of the stages in Equations (7.25)-(7.27):

$$\lim_{\tau\to\infty}\xi_{2,h}(\tau) = k_{2,h}, \quad \lim_{\tau\to\infty}\xi_{2,p}(\tau) = k_{2,p}, \quad \lim_{\tau\to\infty}\xi_{2,q}(\tau) = k_{2,q}.$$

The equilibrium solutions of Equations (7.38)–(7.40) are obtained by applying a numerical time integration technique and performing pseudo-time iterations until a stationary solution is found.

SDIRK2_IM: Solving the Pseudo-ODE for the Second Stage Variable with Forward Euler The semi-explicit pseudo time iterations for solving (7.38)-(7.40) are given by the following scheme:

$$\begin{aligned}
\xi_{2,h}^{k+1} &= \xi_{2,h}^{k} + \frac{\Delta\tau}{\Delta t}\left(F_h(p^n + \Delta t((1-\gamma)k_{1,p} + \gamma\xi_{2,p}^{k})) + G_h(q^n + \Delta t((1-\gamma)k_{1,q} + \gamma\xi_{2,q}^{k}))) - \xi_{2,h}^{k+1}\right),\\
\xi_{2,p}^{k+1} &= \xi_{2,p}^{k} + \frac{\Delta\tau}{\Delta t}(F_p(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_{2,h}^{k}))),\\
&\quad p^n + \Delta t((1-\gamma)k_{1,p} + \gamma\xi_{2,p}^{k})), q^n + \Delta t((1-\gamma)k_{1,q} + \gamma\xi_{2,q}^{k}), t^n + \Delta t) - \xi_{2,p}^{k+1}),\\
\xi_{2,q}^{k+1} &= \xi_{2,q}^{k} + \frac{\Delta\tau}{\Delta t}(F_q(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_{2,h}^{k}))), p^n + \gamma\Delta t((1-\gamma)k_{1,p} + \gamma\xi_{2,p}^{k+1})),\\
&\quad q^n + \Delta t((1-\gamma)k_{1,q} + \gamma\xi_{2,q}^{k})) - \xi_{2,q}^{k+1}).
\end{aligned}$$

where $\Delta\tau$ is the pseudo-time-step and $\xi_{2,h}^{k+1}$, $\xi_{2,p}^{k+1}$ and $\xi_{2,q}^{k+1}$ are numerical approximations of $\xi_{2,h}(\tau)$, $\xi_{2,p}(\tau)$ and $\xi_{2,q}(\tau)$ at $\tau = (k+1)\Delta\tau$, respectively.

The initial approximations for $\xi_{2,h}^{0}$, $\xi_{2,p}^{0}$, and $\xi_{2,q}^{0}$ are the stages $\bar{k}2, h$, $\bar{k}2, p$, and $\bar{k}_{2,q}$, which were computed in the previous outer time iteration when solving the following systems:

$$\begin{aligned}
\bar{k}_{2,h} &= F_h(p^{n-1} + \Delta t((1-\gamma)\bar{k}_{1,p} + \gamma\bar{k}_{2,p})) + G_h(q^{n-1} + \Delta t((1-\gamma)\bar{k}_{1,q} + \gamma\bar{k}_{2,q}))),\\
\bar{k_{2,p}} &= F_p(h^{n-1} + \Delta t(((1-\gamma)\bar{k}_{1,h} + \gamma\bar{k}_{2,h}))),\\
&\quad p^{n+1} + \Delta t((1-\gamma)\bar{k}_{1,p} + \gamma\bar{k}_{2,p})), q^{n+1} + \Delta t((1-\gamma)\bar{k}_{1,q} + \gamma\bar{k}_{2,q}), t^{n-1} + \Delta t),\\
\bar{k}_{2,q} &= F_q(h^{n-1} + \Delta t(((1-\gamma)\bar{k}_{1,h} + \gamma\bar{k}_{2,h}))), p^{n-1} + \gamma\Delta t((1-\gamma)\bar{k}_{1,p} + \gamma\bar{k}_{2,p})), q^{n-1} + \Delta t((1-\gamma)\bar{k}_{1,q} + \gamma\bar{k}_{2,q})).
\end{aligned}$$

Since our objective is to iterate till an equilibrium of (7.32)-(7.34) is found, we define the following stopping criterion:

$$||F_h(p^n + \Delta t((1-\gamma)k_{1,p} + \gamma\xi_{2,p}^{k+1})) + G_h(q^n + \Delta t((1-\gamma)k_{1,q} + \gamma\xi_{2,q}^{k+1}))) - \xi_{2,h}^{k+1}||_\infty < 1e-8.$$

If this stopping criterion is met, the first stage in Equation (7.28)-(7.31) is updated by:

$$k_{2,h} = \xi_{2,h}^{k+1}, \quad , k_{2,p} = \xi_{2,p}^{k+1}, \quad k_{2,q} = \xi_{2,q}^{k+1}.$$

Furthermore in our implementation we store this stage and use this as the first approximation for the first stage pseudo time variable, while computing the values in the $(n+2)$-th time iteration.

SDIRK2_IMEX Solving Pseudo-ODEs for Two SDIRK Stages Using a Semi-Explicit Approach For the first stage:

$$\begin{aligned}
\xi_{1,p}^{k+1} &= \xi_{1,p}^{k} + \frac{\Delta\tau}{\Delta t}(F_p(h^n + \gamma\Delta t\xi_{1,h}^{k}, p^n + \gamma\Delta t\xi_{1,p}^{k}, q^n + \gamma\Delta t\xi_{1,q}^{k}, t^n + \gamma\Delta t) - \xi_{1,p}^{k+1}),\\
\xi_{1,q}^{k+1} &= \xi_{1,q}^{k} + \frac{\Delta\tau}{\Delta t}(F_q(h^n + \gamma\Delta t\xi_{1,h}^{k}, p^n + \gamma\Delta t\xi_{1,p}^{k+1}, q^n + \gamma\Delta t\xi_{1,q}^{k}) - \xi_{1,q}^{k+1}),\\
\xi_{1,h}^{k+1} &= \xi_{1,h}^{k} + \frac{\Delta\tau}{\Delta t}(F_h(p^n + \gamma\Delta t\xi_{1,p}^{k+1}) + G_h(q^n + \gamma\Delta t\xi_{1,q}^{k+1}) - \xi_{1,h}^{k+1}).
\end{aligned}$$

To ensure symmetry, we alternate between updating the variables corresponding to $\xi_{1,p}$ and $\xi_{1,q}$. First, the new updates are substituted into the scheme for updating $\xi_{1,q}$ and $\xi_{1,p}$, respectively. Therefore, the $(k+2)$-th pseudo-time iteration follows structure:

$$\begin{aligned}
\xi_{1,q}^{k+2} &= \xi_{1,q}^{k+1} + \frac{\Delta\tau}{\Delta t}(F_q(h^n + \gamma\Delta t\xi_{1,h}^{k+1}, p^n + \gamma\Delta t\xi_{1,p}^{k+1}, q^n + \gamma\Delta t\xi_{1,q}^{k+1}) - \xi_{1,q}^{k+2}),\\
\xi_{1,p}^{k+2} &= \xi_{1,p}^{k+1} + \frac{\Delta\tau}{\Delta t}(F_p(h^n + \gamma\Delta t\xi_{1,h}^{k+1}, p^n + \gamma\Delta t\xi_{1,p}^{k+1}, q^n + \gamma\Delta t\xi_{1,q}^{k+2}, t^n + \gamma\Delta t) - \xi_{1,p}^{k+2}),\\
\xi_{1,h}^{k+2} &= \xi_{1,h}^{k+1} + \frac{\Delta\tau}{\Delta t}(F_h(p^n + \gamma\Delta t\xi_{1,p}^{k+2}) + G_h(q^n + \gamma\Delta t\xi_{1,q}^{k+2}) - \xi_{1,h}^{k+2}).
\end{aligned}$$

To approximate the second stage, semi-implicit inner iterations are used once again, similar to the first stage. The following equations outline the procedure:

$$
\begin{aligned}
\xi_{2,p}^{k+1} &= \xi_{2,p}^{k} + \frac{\Delta\tau}{\Delta t}(F_p(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_{2,h}^{k}))),\\
&\quad p^n + \Delta t((1-\gamma)k_{1,p} + \gamma\xi_{2,p}^{k})), q^n + \Delta t((1-\gamma)k_{1,q} + \gamma\xi_{2,q}^{k}), t^n + \Delta t) - \xi_{2,p}^{k+1}),\\
\xi_{2,q}^{k+1} &= \xi_{2,q}^{k} + \frac{\Delta\tau}{\Delta t}(F_q(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_{2,h}^{k}))), p^n + \gamma\Delta t((1-\gamma)k_{1,p} + \gamma\xi_{2,p}^{k+1})),\\
&\quad q^n + \Delta t((1-\gamma)k_{1,q} + \gamma\xi_{2,q}^{k})) - \xi_{2,q}^{k+1}).\\
\xi_{2,h}^{k+1} &= \xi_{2,h}^{k} + \frac{\Delta\tau}{\Delta t}\left(F_h(p^n + \Delta t((1-\gamma)k_{1,p} + \gamma\xi_{2,p}^{k+1})) + G_h(q^n + \Delta t((1-\gamma)k_{1,q} + \gamma\xi_{2,q}^{k+1}))) - \xi_{2,h}^{k+1}\right),\\
\xi_{2,q}^{k+2} &= \xi_{2,q}^{k+1} + \frac{\Delta\tau}{\Delta t}(F_q(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_{2,h}^{k+1}))), p^n + \gamma\Delta t((1-\gamma)k_{1,p} + \gamma\xi_{2,p}^{k+1})),\\
&\quad q^n + \Delta t((1-\gamma)k_{1,q} + \gamma\xi_{2,q}^{k+1})) - \xi_{2,q}^{k+2}).\\
\xi_{2,p}^{k+2} &= \xi_{2,p}^{k+1} + \frac{\Delta\tau}{\Delta t}(F_p(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_{2,h}^{k+1}))),\\
&\quad p^n + \Delta t((1-\gamma)k_{1,p} + \gamma\xi_{2,p}^{k+1})), q^n + \Delta t((1-\gamma)k_{1,q} + \gamma\xi_{2,q}^{k+2}), t^n + \Delta t) - \xi_{2,p}^{k+2}),\\
\xi_{2,h}^{k+2} &= \xi_{2,h}^{k+1} + \frac{\Delta\tau}{\Delta t}\left(F_h(p^n + \Delta t((1-\gamma)k_{1,p} + \gamma\xi_{2,p}^{k+2})) + G_h(q^n + \Delta t((1-\gamma)k_{1,q} + \gamma\xi_{2,q}^{k+2}))) - \xi_{2,h}^{k+2}\right).
\end{aligned}
$$

7.1.3. Applying Pseudo-Times by only Approximating h^{n+1} as a Pseudo-Time Variable

In this subsection, an alternative approach using pseudo-time variables for solving the unknowns in different implicit time schemes is outlined. To solve the nonlinear systems at time step t^{n+1}, only the variable h^{n+1} is approximated as a pseudo-time variable ξ_h. The p and q variables are updated alternately in each pseudo-time iteration, using the latest updates of ξ_h.

This approach will be explained for the Backward Euler, Crank-Nicholson, and BDF2 methods. For the second-order SDIRK method, a slightly different approach is outlined, where two stages, $k_{1,h}$ and $k_{1,p}$, correspond to the updates of h^{n+1} and are solved using a pseudo-time scheme.

BE_FW: Approximating h^{n+1} in the Backward Euler-scheme

Integrating the Problem (7.8)-(7.10) with respect to time, by applying Backward Euler, results in the following nonlinear problem:

$$h^{n+1} = h^n + \Delta t(f_h(p^{n+1}) + g_h(q^{n+1}) \tag{7.41}$$

$$p^{n+1} = p^n + \Delta t\left(f_p(h^{n+1}) + g_p(q^{n+1}) + v_p(h^{n+1}, p^{n+1}, q^{n+1}) + f_{tide}(h^{n+1}, t^{n+1})\right) \tag{7.42}$$

$$q^{n+1} = q^n + \Delta t(f_q(h^{n+1}) + g_q(p^{n+1}) + v_q(h^{n+1}, p^{n+1}, q^{n+1})). \tag{7.43}$$

In the Backward Euler scheme given by Equations (7.41)-(7.43), substituting the updates for p^{n+1} and q^{n+1} in the updates of h^{n+1} yields:

$$
\begin{aligned}
h^{n+1} = h^n + \Delta t f_h\left(p^n + \Delta t(f_p(h^{n+1}) + g_p(q^{n+1}) + v_p(h^{n+1}, p^{n+1}, q^{n+1}) + f_{tide}(h^{n+1}, t^{n+1}))\right) +\\
g_h\left(q^n + \Delta t(f_q(h^{n+1}) + g_q(p^{n+1}) + v_q(h^{n+1}, p^{n+1}, q^{n+1}))\right).
\end{aligned} \tag{7.44}
$$

Furthermore h^{n+1} is approximated with ξ in Equation (7.44):

$$\xi \approx h^{n+1},$$

and we obtain an equation that has to be solved for ξ:

$$
\begin{aligned}
\xi = h^n + \Delta t f_h\left(p^n + \Delta t(f_p(\xi) + g_p(q^{n+1}) + v_p(\xi, p^{n+1}, q^{n+1})) + f_{tide}(\xi, t^{n+1})\right) +\\
g_h\left(q^n + \Delta t(f_q(\xi) + g_q(p^{n+1}) + v_q(\xi, p^{n+1}, q^{n+1})\right).
\end{aligned} \tag{7.45}
$$

Solving (7.45), for ξ, is similar to finding the root of the function G, given by Equation (7.46).

$$\begin{aligned} G(\xi) = -\frac{\xi - h^n}{\Delta t} + \\ (f_h(p^n + \Delta t(f_p(\xi) + g_p(q^{n+1}) + p_p(\xi, p^{n+1}, q^{n+1}) + f_{tide}(\xi, t^{n+1}))) + \\ g_h(q^n + \Delta t(f_q(\xi) + g_q(p^{n+1}) + v_q(\xi, p^{n+1}, q^{n+1})))). \end{aligned} \tag{7.46}$$

To which pseudo-time-stepping can be applied such that:

$$\frac{d\xi}{d\tau} = G(\xi).$$

This ODE will be solved with a first order Euler Forward time-integration method.

With the following procedure the approximations of the unknowns in the Backward Euler iterations in Equations (7.11)-(7.13) are found:

$$\begin{aligned} p^{n+1,k+1} &= p^n + \Delta t \left(f_p(\xi^k) + g_p(q^{n+1,k}) + v_p(h^{n+1}, p^{n+1,k}, q^{n+1,k}) + f_{tide}(\xi^k, t^{n+1}) \right), \\ q^{n+1,k+1} &= q^n + \Delta t(f_q(\xi^k) + g_q(p^{n+1,k+1}) + v_q(\xi^k, p^{n+1,k+1}, q^{n+1,k})), \\ \xi^{k+1} &= \xi^k + \Delta\tau \left(-\frac{\xi^{k+1} - h^n}{\Delta t} + (f_h(p^{n+1,k+1}) + g_h(q^{n+1,k+1})) \right), \end{aligned}$$

where $\Delta\tau$ is the pseudo-time step and $p^{n+1,k}$ and $q^{n+1,k}$ are the k-th approximations of p^{n+1} and q^{n+1}, respectively. In this procedure the values of p are updated firstly and secondly the values of q are updated. Finally, ξ is updated with the latest values of p and q.

In the previous inner time iteration the values of p are updated, firstly. After that, the values of q are updated. To enforce symmetry, we will update q in the next inner time iteration. The value of ξ, will always be updated as last within the inner time iterations using the latest updates of p and q.

$$\begin{aligned} q^{n+1,k+2} &= q^n + \Delta t(f_q(\xi^{k+1}) + g_q(p^{n+1,k+1}) + v_q(\xi^{k+1}, p^{n+1,k+1}, q^{n+1,k+1})), \\ p^{n+1,k+2} &= p^n + \Delta t \left(f_p(\xi^{k+1}) + g_p(q^{n+1,k+2}) + v_p(\xi_h^{k+1}, p^{n+1,k+1}, q^{n+1,k+2}) + f_{tide}(\xi^{k+1}, t^{n+1}) \right), \\ \xi^{k+2} &= \xi^{k+1} + \Delta\tau \left(-\frac{\xi^{k+2} - h^n}{\Delta t} + (f_h(p^{n+1,k+2}) + g_h(q^{n+1,k+2})) \right). \end{aligned}$$

Alternating the order in which p and q are updated during the inner time steps is a strategy that will be consistently employed in the subsequent paragraphs.

Moreover, The solutions from the previous outer time step are used as initial approximations:

$$p^{n+1,0} \approx p^n, \quad q^{n+1,0} \approx q^n, \quad \xi^0 \approx h^n.$$

In Appendix B, one can find a similar procedure for solving nonlinear systems arising from BDF2-and CN-iterations by pseudo-times by only approximating h as a pseudo-time variable.

SDIRK2_FW: Approximating only $k_{1,h}$ and $k_{2,h}$ as pseudo-time variables in the SDIRK2-scheme

Instead of approximating h as a pseudo-time variable, we choose to approximate the stage-variables corresponding to h in the SDIRK2-method. The variales $k_{1,h}$ and $k_{2,h}$ are approximated as pseudo-time variables.

Approximating the first stage For solving the values in the first stage the value of $k_{1,h}$ is approximated by ξ_1. The root of the following function has to be found:

$$G_1(\xi_1) = \frac{1}{\Delta t}\left(F_h(p^n + \gamma\Delta t(k_{1,p})) + G_h(q^n + \gamma\Delta t k_{1,q}) - \xi_1\right),$$

where $k_{1,p}$ and $k_{1,q}$ are:

$$\begin{aligned} k_{1,p} &= F_p(h^n + \gamma\Delta t\xi_1, p^n + \gamma\Delta t k_{1,p}, q^n + \gamma\Delta t k_{1,q}, t^n + \gamma\Delta t), \\ k_{1,q} &= F_q(h^n + \gamma\Delta t\xi_1, p^n + \gamma\Delta t k_{1,p}, q^n + \gamma\Delta t k_{1,q}). \end{aligned}$$

With pseudo-time-stepping the first stage can be computed, by solving the following ODE:

$$\frac{d\xi_1}{d\tau} = G_1(\xi_1).$$

Furthermore with the following inner-iterations a solution is found

$$\begin{aligned} k_{1,p}^{k+1} &= F_p(h^n + \gamma\Delta t\xi_{1,h}^k, p^n + \gamma\Delta t k_{1,p}^k, q^n + \gamma\Delta t k_{1,q}^k, t^n + \gamma\Delta t), \\ k_{1,q}^{k+1} &= F_p(h^n + \gamma\Delta t\xi_{1,h}^k, p^n + \gamma\Delta t k_{1,p}^{k+1}, q^n + \gamma\Delta t k_{1,q}^k, t^n + \gamma\Delta t), \\ \xi_{1,h}^{k+1} &= \xi_{1,h}^k + \frac{\Delta\tau}{\Delta t}(F_h(p^n + \gamma\Delta t k_{1,p}^{k+1}) + G_h(q^n + \gamma\Delta t k_{1,q}^{k+1}) - \xi_1^{k+1}), \\ k_{1,q}^{k+2} &= F_p(h^n + \gamma\Delta t\xi_{1,h}^{k+1}, p^n + \gamma\Delta t k_{1,p}^{k+1}, q^n + \gamma\Delta t k_{1,q}^{k+1}, t^n + \gamma\Delta t), \\ k_{1,p}^{k+2} &= F_p(h^n + \gamma\Delta t\xi_{1,h}^{k+1}, p^n + \gamma\Delta t k_{1,p}^{k+1}, q^n + \gamma\Delta t k_{1,q}^{k+2}, t + \gamma\Delta t), \\ \xi_{1,h}^{k+2} &= \xi_{1,h}^{k+1} + \frac{\Delta\tau}{\Delta t}(F_h(p^n + \gamma\Delta t k_{1,p}^{k+2}) + G_h(q^n + \gamma\Delta t k_{1,q}^{k+2}) - \xi_1^{k+2}). \end{aligned}$$

Approximating the second stages With a similar iteration scheme it is possible to solve for the second stages $k_{2,h}$, $k_{2,p}$ and $k_{2,q}$.

For solving the unknowns in the second stage, the value of $k_{2,h}$ is approximated by ξ_2. The root of the following function has to be found:

$$G_2(\xi_2) = F_h(p^n + \Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p})) + G_h(q^n + \Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p})) - \xi_2,$$

where $k_{2,p}$ and $k_{2,q}$ are:

$$\begin{aligned} k_{2,p} = F_p(h^n &+ \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_2))), \\ p^n + \Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p})), q^n &+ \Delta t((1-\gamma)k_{1,q} + \gamma k_{2,q}), t^n + \Delta t), \\ k_{2,q} = F_q(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_2))), p^n &+ \gamma\Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p})), q^n + \Delta t((1-\gamma)k_{1,q} + \gamma k_{2,q})). \end{aligned}$$

With pseudo-time-stepping the first stage can be computed, by solving the following ODE:

$$\frac{d\xi_2}{d\tau} = G_2(\xi_2).$$

With the following pseudo-time iterations the Equations (7.28)-(7.31) can be solved:

$$
\begin{aligned}
k_{2,p}^{k+1} &= F_p(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_{2,h}^{k}))), \\
&\quad p^n + \Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p}^{k})), q^n + \Delta t((1-\gamma)k_{1,q} + \gamma k_{2,q}^{k}), t^n + \Delta t), \\
k_{2,q}^{k+1} &= F_q(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_{2,h}^{k}))), p^n + \gamma\Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p}^{k+1})), q^n + \Delta t((1-\gamma)k_{1,q} + \gamma k_{2,q}^{k})), \\
\xi_{2,h}^{k+1} &= \xi_{2,h}^{k} + \frac{\Delta\tau}{\Delta t}(F_h(p^n + \Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p}^{k+1})) + G_h(q^n + \Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p}^{k+1})) - \xi_{2,h}^{k+1}), \\
k_{2,q}^{k+2} &= F_q(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_{2,h}^{k+1}))), p^n + \gamma\Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p}^{k+1})), q^n + \Delta t((1-\gamma)k_{1,q} + \gamma k_{2,q}^{k+1})), \\
k_{2,p}^{k+2} &= F_p(h^n + \Delta t(((1-\gamma)k_{1,h} + \gamma\xi_{2,h}^{k+1}))), \\
&\quad p^n + \Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p}^{k+1})), q^n + \Delta t((1-\gamma)k_{1,q} + \gamma k_{2,q}^{k+2}), t^n + \Delta t), \\
\xi_{2,h}^{k+2} &= \xi_{2,h}^{k+} + \frac{\Delta\tau}{\Delta t}(F_h(p^n + \Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p}^{k+2})) + G_h(q^n + \Delta t((1-\gamma)k_{1,p} + \gamma k_{2,p}^{k+2})) - \xi_{2,h}^{k+2}).
\end{aligned}
$$

7.1.4. Damping In the systems

In this subsection, we approximate the eigenvalues of the pseudo-time ODE's and analyze the negative eigenvalues for damping.

Damping in the pseudo-time ODEs

To further simplify (7.8)-(7.10), the system of SWEs is presented by the following equation:

$$\frac{\partial}{\partial t}\mathbf{x} = H(\mathbf{x}, t), \tag{7.47}$$

where H represents the right hand side of the test-case. After applying a time-integration scheme, we define a pseudo-time dependent ODE. The equilibrium state of this ODE is equal to the solution in the next time step.

Subsequently, The pseudo-time ODEs, for solving nonlinear systems within the iterations within the iterations for Backward Euler, Crank-Nicolson, BDF2-methods and SDIRK2-methods have the following kind of form:

$$\frac{d\xi}{d\tau} = \mathbf{G}(\xi).$$

To Determine the behavior of this nonlinear ordinary differential equation towards an equilibrium point, we linearize this system around the zero-value and compute the eigenvalues of this system. Note that $G(\xi)$ is a linear combination of $H(\xi, t^{n+1})$ and (ξ).

In Chapter 6, we found in Section (6.1.3) that the function right-side of (7.8)-(7.10), has a Jacobian with imaginary eigenvalues, which are approximately equal to $i\frac{2\sqrt{(2Dg)}}{\Delta x}$. Furthermore if we evaluate the Jacobian of (7.8)-(7.10) in the zero-vector, the eigenvalues are purely imaginary. In this subsection we only analyze the case where the eigenvalues are computed of the linearized system around the zero-vector.

Damping in the Backward Euler pseudo-ODE Applying Backward Euler time-integration to (7.47), rewriting the system as a pseudo-time-ODE yields:

$$\frac{\partial\xi}{\partial t} = H(\xi, t^{n+1}) - \frac{\xi - \mathbf{x}^n}{\Delta t} = \mathbf{G}(\xi) \tag{7.48}$$

where ξ denotes the τ-dependent approximation of $\mathbf{x}$ at t^{n+1} and $\mathbf{x}^n$ is the value of $\mathbf{x}$ at $t = t^n$. The Jacobian of the right-hand side of (7.48) is:

$$\frac{\partial G(\xi)}{\partial\xi} = \frac{\partial}{\partial\xi}H(\xi, t^{n+1}) - \frac{1}{\Delta t},$$

and the eigenvalue of G linearized around a 0-value with the largest imaginary part is:

$$-\frac{1}{\Delta t} \pm \frac{2\sqrt{2Dg}}{\Delta x} i,$$

since we know that the eigenvalues of $\frac{\partial}{\partial \xi} H(\xi, t^{n+1})\Big|_{\xi=0}$ are at most $\frac{2\sqrt{2Dg}}{\Delta x} i$ in the imaginary direction. The real component is negative and depends on $-\frac{1}{\Delta t}$ and this indicates that convergence to an equilibrium will happen. However, as Δt is chosen to be larger, the damping-component in the pseudo ODE will decrease.

In Subsection 6.1.3, Figure 6.3 presents the eigenvalues of H, linearized around a simulation point for a coarse grid case. The figure shows a small amount of damping, as indicated by eigenvalues with a negative real part of magnitude $\mathcal{O}(10^{-5})$. Since this damping is negligible compared to the $-\frac{1}{\Delta t}$ term in G, our analysis only considers the complex eigenvalues of H obtained from linearizing around zero.

Damping in the Crank-Nicolson pseudo-ODE Applying Crank-Nicolson to (7.47) and rewriting the system as a pseudo-time-ODE yields:

$$\frac{\partial \xi}{\partial t} = \frac{1}{2} H(\xi, t^{n+1}) + \frac{1}{2} H(\mathbf{x}^n, t^n) - \frac{\xi - \mathbf{x}^n}{\Delta t} = \mathbf{G}(\xi), \tag{7.49}$$

where ξ denotes the τ-dependent approximation of $\mathbf{x}$ at t^{n+1} and $\mathbf{x}^n$ is the value of $\mathbf{x}$ at $t = t^n$. The Jacobian of the right-hand side of (7.49) is:

$$\frac{\partial G(\xi)}{\partial \xi} = \frac{\partial}{\partial \xi} \frac{1}{2} H(\xi, t^{n+1}) - \frac{1}{\Delta t}.$$

and the eigenvalue of G linearized around 0 with the largest imaginary parts is:

$$-\frac{1}{\Delta t} \pm \frac{\sqrt{2Dg}}{\Delta x} i,$$

since we know that the eigenvalues of $\frac{\partial}{\partial \xi} H(\xi, t^{n+1})\Big|_{\xi=0}$ are at most $\frac{2\sqrt{2Dg}}{\Delta x} i$ in the imaginary direction. In addition, for the Crank-Nicolson case, the eigenvalue of $\frac{\partial}{\partial \xi} \frac{1}{2} H(\xi, t^{n+1})\Big|_{\xi=0}$ has at most an imaginary eigenvalue equal to $\frac{\sqrt{2Dg}}{\Delta x} i$. Similar to the Backward Euler case, the negative real eigenvalue of the Jacobian, $\frac{\partial}{\partial \xi} G(\xi)\Big|_{\xi=0}$ is equal to $-\frac{1}{\Delta t}$. The damping in the Crank-Nicolson pseudo-time ODE (7.49) is similar to the damping for the pseudo-time ODE resulting from applying pseudo-times to an Backward Euler implicit time iteration.

Damping in the BDF2 pseudo-ODE Applying a BDF2-scheme to (7.47) and rewriting the system as a pseudo-time-ODE yields:

$$\frac{\partial \xi}{\partial t} = \frac{2}{3} H(\xi, t^{n+1}) - \frac{\xi - \frac{4}{3}\mathbf{x}^n + \frac{1}{3}\mathbf{x}^{n-1}}{\Delta t} = \mathbf{G}(\xi) \tag{7.50}$$

where ξ denotes the τ-dependent approximation of $\mathbf{x}$ at t^{n+1} and $\mathbf{x}^n$ and $\mathbf{x}^{n-1}$ are the values of $\mathbf{x}$ at $t = t^n$ and $t = t^{n-1}$, respectively. The Jacobian of the right-hand side of (7.50) is:

$$\frac{\partial G(\xi)}{\partial \xi} = \frac{2}{3} \frac{\partial}{\partial \xi} H(\xi, t^{n+1}) - \frac{1}{\Delta t}.$$

and the eigenvalue of G linearized around 0 with the largest imaginary part is:

$$-\frac{1}{\Delta t} \pm \frac{2(\frac{2}{3})\sqrt{2Dg}}{\Delta x} i.$$

Similar to the Backward Euler and Crank-Nicolson cases, it can be observed that the $\frac{-\xi}{\Delta t}$-component contributes to damping, which ensures convergence to an equilibrium solution.

Damping in the SDIRK pseudo-ODE For the SDIRK methods, the pseudo-time equations are used to approximate the values of the stages and the pseudo-time ODE has the following shape:

$$\frac{\xi_1}{\partial\tau} = \frac{1}{\Delta t}H(x^n + \Delta t\gamma\xi_1) - \frac{1}{\Delta t}\boldsymbol{\xi_1} = G(\xi_1),$$

where ξ_1 corresponds to the approximation of the first stage. The Jacobian of $G(\xi_1)$ is:

$$\frac{\partial G(\xi_1)}{\partial \xi_1} = \frac{1}{\Delta t}(\frac{\partial H(\xi)}{\partial \xi})\gamma\Delta t - \frac{1}{\Delta t},$$

The eigenvalue of $G(\xi_1)\Big|_{\xi_1=0}$, with the largest imaginary part is:

$$-\frac{1}{\Delta t} \pm \frac{2\gamma\sqrt{2Dg}}{\Delta x}i.$$

Hence the damping in the pseudo time iteration is given by $-\frac{\xi}{\Delta t}$. In the pseudo-time Equations (7.38)-(7.40) for the second stage, the damping is given by the $-\frac{\xi}{\Delta t}$-component, as well.

The Damping in Pseudo-ODE for only h-Approximation

For the new method where only h^{n+1} is approximated as a pseudo-time variable and p^{n+1} and q^{n+1} are approximated using a combined forward-backward approach, we also try to find the damping in the corresponding pseudo-equations. In Section 7.1.3, these methods can be found.

Damping in the Backward Euler Case For the pseudo-time ODE the eigenvalues of the linearized system are found in this subsection, by approximating and simplifying the pseudo-time dependent ODE.

For the τ-dependent problem in (7.46), we again linearize $G(\xi)$ around a zero-point. To obtain this linearized system we first try to simplify and find an approximation of G.

Firstly, we present the function G as:

$$\begin{aligned} G(\xi) = -\frac{\xi - h^n}{\Delta t} + \\ (f_h(p^n + \Delta t(f_p(\xi) + g_p(q^{n+1}) + p_p(\xi, p^{n+1}, q^{n+1}) + f_{tide}(\xi, t^{n+1}))) + \\ g_h(q^n + \Delta t(f_q(\xi) + g_q(p^{n+1}) + v_q(\xi, p^{n+1}, q^{n+1})))). \end{aligned}$$

Furthermore we only take the pressure terms and mass conservation-terms into account. The simplified version of G is $\tilde{G}$:

$$\tilde{G}(\xi) = -\frac{\xi - h^n}{\Delta t} + (f_h(p^n + \Delta t(f_p(\xi))) + g_h(q^n + \Delta t f_q(\xi))),$$

where $f_h(\cdot)$ and $g_h(\cdot)$ represent the spatially discretized pressure terms in our test-case SWEs. Both $f_h(\cdot)$ and $g_h(\cdot)$ are the derivatives of h with respect to x- and y-coordinates, respectively, using central difference discretization. Additionally, these central differences are scaled by H and g. For simplicity in our analysis, we approximate $D \approx H$.

Therefore the function G is approximated as:

$$\begin{aligned} \tilde{G}(\xi) \approx -\frac{\xi - h^n}{\Delta t} + D_x(p^n) + \Delta t D g D_x(D_x(\xi)) + D_y(q^n) + \Delta t D g D_y(D_y(\xi))), \\ \frac{\tilde{G}(\xi)}{\partial \xi} \approx -\frac{1}{\Delta t} + \Delta t D g(D_x^2 + D_y^2), \end{aligned}$$

where D_x and D_y, correspond to the central difference operators for spatial discretization in the x-and y-directions, respectively. The D_x^2 and D_y^2, correspond to discrete Laplacian operators in the x- and

y-direction. The eigenvalues of those operators are computed in subsection 3.3.2 of chapter 3. The eigenvalues of the matrix $(D_x^2 + D_y^2)$ are :

$$-\frac{8}{\Delta x^2} < \lambda_L < 0, \tag{7.51}$$

where λ_L is the eigenvalue of the Laplacian. The eigenvalues of $\tilde{G}$ are in the following range:

$$-\frac{8Dg}{\Delta x^2}\Delta t - \frac{1}{\Delta t} < \lambda < -\frac{1}{\Delta t}$$

For the pseudo-time methods, with only approximations for h^{n+1} and a Forward-Backward approach for computing p^{n+1}, $qn+1$, we can observe that the negative real parts are larger than $-\frac{1}{\Delta t}$. Hence, the damping for in this Forward-Backward approach, is larger compared to the approach from Section (7.1.2) where all the unknowns are approximated as pseudo-time variables. Hence for the same pseudo-time steps τ, we would like to study wether this forward-backward approach is an improvement compared to the pseudo-time schemes in Section (7.1.2).

In the next few paragraphs we derive in a similar way the approximations of eigenvalues pseudo-time differential equations for solving the Crank-Nicolson, BDF2- and SDIRK2-time iterations.

Damping in the Crank-Nicolson Case For the pseudo-time ODE the eigenvalues of the linearized system are found in this subsection, by approximating and simplifying the pseudo-time dependent ODE.

For the τ-dependent problem solving an implicit Crank-Nicolson iteration with pseudo-times is given by :

$$\begin{aligned}\frac{d\xi}{d\tau} = G(\xi) = -\frac{\xi - h^n}{\Delta t} + \frac{1}{2}F_h\left(p^n + \frac{\Delta t}{2}\left(F_p(\xi, p^{n+1}, q^{n+1}, t^{n+1}) + F_p(h^n, p^n, q^n, t^n)\right)\right) + \\ \frac{1}{2}G_h\left(q^n + \frac{\Delta t}{2}\left(F_q(\xi, p^{n+1}, q^{n+1}) + F_q(h^n, p^n, q^n)\right)\right) + \frac{1}{2}\left(F_h(p^n) + G_h(q^n)\right).\end{aligned}$$

We approximate the right-hand side of this ODE by only taking the mass conservation and pressure properties into account. The simplified problem is given by $\tilde{G}$.

$$\begin{aligned}\tilde{G}(\xi) = -\frac{\xi - h^n}{\Delta t} + \frac{1}{2}D_x\left(p^n + \frac{\Delta t}{2}(Dg)D_x\left(\xi + h^n\right)\right)\right) + \\ \frac{1}{2}D_y\left(q^n + \frac{\Delta t}{2}\left((Dg)D_y(\xi + h^n)\right)\right) + \frac{1}{2}\left(D_x(p^n) + D_y(q^n)\right).\end{aligned}$$

The Jacobian of $\tilde{G}$ is :

$$\frac{\tilde{G}(\xi)}{\partial \xi} \approx -\frac{1}{\Delta t} + \frac{1}{4}\Delta t Dg(D_x^2 + D_y^2).$$

The eigenvalues of $\frac{\tilde{G}(\xi)}{\partial \xi}$ are:

$$-\frac{2Dg}{\Delta x^2}\Delta t - \frac{1}{\Delta t} < \lambda < -\frac{1}{\Delta t}$$

From this we can observe that the pseudo-ODE approach given by Section 7.1.3, has a damping that is both caused by the pressure-terms and the $-\frac{\xi}{\Delta t}$-component. The negative real component from the pressure terms, that is at most $-\frac{2Dg}{\Delta x^2}\Delta t$, is in this case smaller compared to the Backward Euler case. Therefore, one could perhaps expect slower convergence.

Damping in BDF2-schemes For the pseudo-time ODE the eigenvalues of the linearized system are found in this subsection, by approximating and simplifying the pseudo-time dependent ODE.

For the τ-dependent problem solving an implicit BDF2-iteration with pseudo-times is given by:

$$\frac{d\xi}{d\tau} = G(\xi) = -\frac{\xi - \frac{4}{3}h^n + \frac{1}{3}h^{n-1}}{\Delta t} +$$
$$\frac{2}{3}\Delta t F_h\left(\frac{4}{3}p^n - \frac{1}{3}p^{n-1} + \frac{2}{3}\Delta t F_p(\xi, p^{n+1}, q^{n+1}, t^{n+1})\right) +$$
$$\frac{2}{3}\Delta t G_h\left(\frac{4}{3}q^n - \frac{1}{3}q^{n-1} + \frac{2}{3}\Delta t F_q(\xi, p^{n+1}, q^{n+1})\right).$$

We approximate the right-hand side of this pseudo-ODE by only taking the mass conservation and pressure properties into account. The simplified problem is given by $\tilde{G}$.

$$\tilde{G}(\xi) = -\frac{\xi - \frac{4}{3}h^n + \frac{1}{3}h^{n-1}}{\Delta t} + \frac{2}{3}\Delta t D_x\left(\frac{4}{3}p^n - \frac{1}{3}p^{n-1} + \frac{2}{3}\Delta t(Dg)D_x\xi\right) +$$
$$\frac{2}{3}\Delta t D_y\left(\frac{4}{3}q^n - \frac{1}{3}q^{n-1} + \frac{2}{3}\Delta t(Dg)D_y\xi\right).$$

The Jacobian of $\tilde{G}$ is :

$$\frac{\tilde{G}(\xi)}{\partial\xi} \approx -\frac{1}{\Delta t} + \frac{4}{9}\Delta t Dg(D_x^2 + D_y^2).$$

The eigenvalues of $\frac{\tilde{G}(\xi)}{\partial\xi}$ are:

$$-\frac{8Dg}{\Delta x^2}\frac{4}{9}\Delta t - \frac{1}{\Delta t} < \lambda < -\frac{1}{\Delta t}.$$

Similar to the Backward Euler and Crank-Nicolson methods, the eigenvalues for the BDF2-technique, where only h is approximated as a pseudo-time ODE, exhibit a negative real part smaller than $-\frac{1}{\Delta t}$, primarily due to the inclusion of pressure terms. The added term $-\frac{8Dg}{\Delta x^2}\frac{4}{9}\Delta t$, is scaled with $\frac{4}{9}$, which is larger compared to the Crank-Nicolson case. Hence, for this BDF2-version one could expect more damping and perhaps even faster convergence.

Damping in SDIRK2-pseudo-time ODE The τ-dependet problem from Section (7.1.3) for solving the first stage of the implicit SDIRK2-method is given by:

$$\frac{\partial\xi}{\partial\tau} = G_1(\xi_1) = \frac{1}{\Delta t}(F_h(p^n + \gamma\Delta t(F_p(h^n + \gamma\Delta t\xi_1, p^n + \gamma\Delta t k_{1,p}, q^n + \gamma\Delta t k_{1,q}, t^n + \gamma\Delta t)))+$$
$$G_h(q^n + \gamma\Delta t F_q(h^n + \gamma\Delta t\xi_1, p^n + \gamma\Delta t k_{1,p}, q^n + \gamma\Delta t k_{1,q})) - \frac{1}{\Delta t}\xi_1.$$

We approximate the right-hand side of the τ-dependent-ODE by only taking the mass conservation and pressure properties of the SWEs into account. The simplified problem is given by $\tilde{G}_1$.

$$\tilde{G}_1(\xi_1) = \frac{1}{\Delta t}\left(D_x(p^n + \gamma\Delta t(D_x(h^n + \gamma\Delta t\xi_1))) + D_y(q^n + \gamma\Delta t(Dg)D_y(h^n + \gamma\Delta t\xi_1)) - \xi_1\right).$$

The Jacobian of $\tilde{G}$ is :

$$\frac{\tilde{G}_1(\xi)}{\partial\xi} \approx -\frac{1}{\Delta t} + Dg\gamma^2(D_x^2 + D_y^2)\Delta t.$$

The eigenvalues of $\frac{G_1}{\partial \xi}$ are:

$$-\frac{8Dg\gamma^2}{\Delta x^2}\Delta t - \frac{1}{\Delta t} < \lambda < -\frac{1}{\Delta t}.$$

Similarly to the previous methods, the eigenvalues of this system are negative and smaller than $\frac{1}{\Delta t}$. However the extra negative real part caused by the pressure terms has values between $-\frac{8Dg\gamma^2}{\Delta x^2}\Delta t$ and 0 and is scaled by γ^2. The value of $\gamma \approx 0.29$, since this scaling is lower compared to the BDF2, CN and BE cases, one could perhaps expect slower convergence.

7.2. Stability properties of Pseudo-Time Iterations

In this section, the maximum allowable size for the pseudo-time step $\Delta\tau$, is derived. For both the explicit numerical solver of the pseudo-time ODE and the Forward-Backward approach from Section (7.1.3), we will analytically derive the stability region. Moreover, to determine the stability of the IMEX-style inner-integration, the eigenvalues of the Jacobian of two inner time-iterations is computed.

7.2.1. Stability Analysis for Fully Explicit Pseudo-Time Integration

For both the pseudo-time ODEs for solving iterations using Backward Euler, Crank-Nicolson, BDF2 and SDIRK schemes, the stability regions are computed. The stability function is equal to:

$$Q(\Delta\tau) = \frac{|1 + \Delta\tau\lambda_{EX}|}{|1 - \Delta\tau\lambda_{IM}|},$$

where λ_{EX} is the eigenvalue of the explicit component in the scheme and λ_{IM} is the implicit component of the pseudo-time integration scheme. Since for all the methods, we choose to integrate $\frac{-\xi}{\Delta t}$ implicitly, the value of $\lambda_I M$ will always be equal to $\frac{-1}{\Delta t}$.

For approximating λ_{EX}, we take the values λ, computed in Section 7.1.4 (about damping). Let $\lambda_{EX} = \lambda_m - \lambda_{IM}$, where the value of λ_m is the value of λ with the largest modulus. In Table 7.1 the eigenvalues corresponding to the parts of the equation that is integrated explicitly are shown.

Table 7.1: Eigenvalues of Jacobian of the explicit-component with largest modulus, from explicit parts.

method	Backward Euler	Crank-Nicolson	BDF2	SDIRK2
λ_{EX}	$\frac{2\sqrt{2Dg}}{\Delta x}i$	$\frac{\sqrt{2Dg}}{\Delta x}i$	$\frac{2\frac{2}{3}\sqrt{2Dg}}{\Delta x}i$	$\frac{2\gamma\sqrt{2Dg}}{\Delta x^2}$
λ_{EX}^2	$\frac{-8Dg}{\Delta x^2}$	$\frac{-2Dg}{\Delta x^2}$	$\frac{-8\frac{4}{9}Dg}{\Delta x^2}$	$\frac{-8\gamma^2 Dg}{\Delta x^2}$

To determine the stability region, we determine $\Delta\tau$ such that :

$$Q(\Delta\tau) = \frac{|1 + \Delta\tau\lambda_{EX}|}{|1 + \frac{\Delta\tau}{\Delta t}|} < 1$$

holds. The following stability condition holds:

$$\Delta\tau < \frac{2}{\lambda_{EX}^2\Delta t - \frac{1}{\Delta t}}. \tag{7.52}$$

Finally, the stability upper-bound is computed by substituting the explicit eigenvalues from Table 7.1 in Equation (7.52). For pseudo-time ODES corresponding to different implicit methods, the different upper bounds for $\Delta\tau$, are shown in Table 7.2.

Table 7.2: Maximum allowable pseudo-time steps.

Implicit Method	$\Delta\tau_{max}$
Backward Euler	$\frac{2}{\Delta t \frac{8Dg}{\Delta x^2} - \frac{1}{\Delta t}}$
Crank-Nicolson	$\frac{2}{\Delta t \frac{2Dg}{\Delta x^2} - \frac{1}{\Delta t}}$
BDF2	$\frac{2}{\Delta t \frac{8\frac{4}{9}Dg}{\Delta x^2} - \frac{1}{\Delta t}}$
SDIRK2	$\frac{2}{\gamma^2 \Delta t 8Dg\Delta x^{-2} - \frac{1}{\Delta t}}$

7.2.2. Stability Analysis for IMEX-style Inner Integration

For examining the stability of the IMEX-solvers of the pseudo-time ODE, we choose a different approach. We define a function with the input of the pseudo-time variables at iteration k and the output at iteration $k+2$. Furthermore, The absolute value of the Jacobian of this function, is an approximation of the stability function for the semi-explicit solvers. The Jacobian of this function is computed with the ForwardDiff.jl package. We evaluate this Jacobian in the zero vector and compute its eigenvalues. For different guesses, we try to select a value of $\Delta\tau$, such that all the eigenvalues of the Jacobian have absolute values lower than 1.

In Figure 7.1 various eigenvalues corresponding to stability functions of different implicit methods with inner IMEX-iterations are shown.

Note that, we selected an outer time step of 100 seconds and a 270×270 grid. Jacobian and the eigenvalues were computed on subsection off the grid, by letting $\Delta x = \Delta y = \frac{L}{270}$ and setting the number of grid points $n_x = n_y = 20$.

In the next sections, we use this approach for selecting the maximum allowable time-step size for various methods.

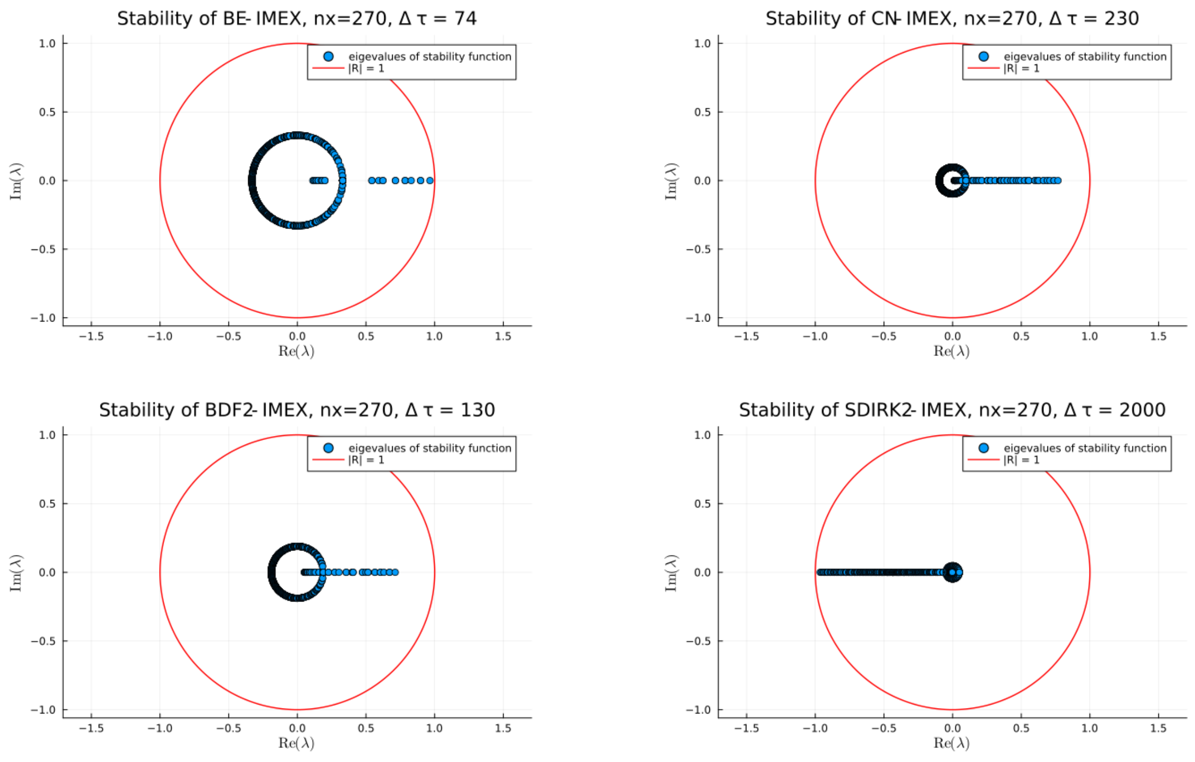


Figure 7.1: Eigenvalues of Linearized Stability Functions, with $\Delta t = 100$.

7.2.3. Stablitity analysis for applying pseudo-time integration to h^{n+1}

For the pseudo-time methods, where only h^{n+1} is approximated as a pseudo-time variable, given by Section 7.1.3, derive the stability region. From now on, these methods are referred to as Forward-Backward-methods.

For both the pseudo-time ODEs for solving iterations using Backward Euler, Crank-Nicolson, BDF2 and SDIRK schemes, the stability regions are computed. The stability function is equal to:

$$Q(\Delta\tau) = \frac{|1 + \Delta\tau\lambda_{EX}|}{|1 - \Delta\tau\lambda_{IM}|},$$

where λ_{EX} is the eigenvalue of the explicit component in the scheme and λ_{IM} is the implicit component of the pseudo-time integration scheme. Since for all the Forward-Backward-methods, we choose to integrate $\frac{-\xi}{\Delta t}$ implicitly, the value of λ_{IM} will always be equal to $\frac{-1}{\Delta t}$.

For approximating λ_{EX}, we take the values λ, approximated in Section 7.1.4 about damping. Let $\lambda_{EX} = \lambda_m - \lambda_{IM}$, where The value of λ_m is the value of λ with the negative real part. In Table 7.3 the eigenvalues of the Jacobian corresponding t0 the parts of the equation that are integrated explicitly are shown.

Table 7.3: Eigenvalues of Jacobian of the explicit-component with largest modulus, from explicit parts.

method	Backward Euler	Crank-Nicolson	BDF2	SDIRK2
λ_{EX}	$\frac{-8Dg}{\Delta x^2}\Delta t$	$\frac{-2Dg}{\Delta x^2}\Delta t$	$\frac{-8\frac{4}{9}Dg}{\Delta x^2}\Delta t$	$\frac{-8\gamma^2 Dg}{\Delta x^2}\Delta t$

To determine the stability region, we determine $\Delta\tau$ such that :

$$Q(\Delta\tau) = \frac{|1 + \Delta\tau\lambda_{EX}|}{|1 + \frac{\Delta\tau}{\Delta t}|} < 1$$

holds. The following stability condition holds:

$$\Delta\tau < \frac{2}{(-\lambda_{EX}) - \frac{1}{\Delta t}}. \tag{7.53}$$

Finally, we find that the stability conditions for Forward-Backward methods applied to different implicit schemes are similar to the stability condition shown in Table 7.2. The stability conditions for $\Delta\tau$ in the Forward-Backward methods are the same as the for the fully explicit solver of implicit time-schemes.

7.3. Experiments for Pseudo-time methods

In this Section experiments with different inner time-integration schemes to different implicit outer time-integration schemes are presented. The numerical errors and simulation times are computed and compared for different methods.

In Subsection 7.3.1, we experiment with different pseudo-time schemes by varying the time step. After that, in Subsection 7.3.2 experiments are conducted on different mesh sizes.

7.3.1. Experiments for different time steps

Setting up the Experiments

To assess the accuracy and computational times of the different pseudo-time methods presented in Section 7.1, the following experiments are designed:

- The SWEs are solved on a 20×20 computational grid, with a time span of 10 days.
- For the every inner time iterations, we evaluate the right-hand side of the pseudo-time dependent ODE corresponding to the right hand side of the h-component, during the inner iterations, using the updated values of h, p, and q. If the infinity norm of this value is smaller than 1×10^{-8} after at least n_t iterations, then we set n_t as the number of inner iterations per outer time step. We

choose to let n_t to be the same for solving every outer time step. We let for our experiments n_t be uniform for every outer time step. We determine n_t such that the stopping criterion for the last 20 outer iteration are less than 1×10^{-8} during a 10 day simulation after n_t inner iterations. After having determined n_t, we run experiments with n_t inner iterations for solving every outer time-iteration and measure time.

- To approximate the numerical errors, a fourth-order Runge-Kutta scheme with $\Delta t = 10$ is used as the benchmark solution, computed on a 20×20 grid. The numerical errors are determined by calculating the infinity norm between the benchmark solution and the experimental solution obtained from the pseudo-time solver. For each mesh node, the absolute value of the error is computed, and the maximum error is selected.
- An H100-PCIe, 80GB GPU is selected for running the experiments. For computations on a CPU, an Intel(R) Xeon(R) Platinum 8462Y+ node is utilized. The experiments were run on the Deltares Cluster.

Experimental Results

In Table 7.4, the number of function evaluations per inner time iteration, denoted as n_t, is shown. The value N_f represents the number of function evaluations per outer time step.

Table 7.4: Experiments with pseudo-transient, different stepsizes.

method	Δt	$\Delta\tau$	n_t	N_f
BE_IM	250	3000	4	4
	500	1100	8	8
	1000	300	33	33
BE_IMEX	250	3000	4	4
	500	2600	6	6
	1000	1100	8	8
BE_FW	250	3000	4	4
	500	1100	8	8
	1000	300	34	34
CN_IM	250	6000	3	4
	500	6000	4	5
	1000	2000	10	11
CN_IMEX	250	6000	4	5
	500	6000	4	5
	1000	4500	6	7
CN_FW	250	6000	4	5
	500	6000	4	5
	1000	2000	10	11
BDF2_IM	250	10000	3	3
	500	10000	11	11
	1000	850	17	17
BDF2_IMEX	250	10000	4	4
	500	10000	4	4
	1000	2400	8	8
BDF2_FW	250	10000	4	4
	500	10000	4	4
	1000	850	16	16
SDIRK_IM	250	20000	2	4
	500	40000	3	6
	1000	80000	7	14
SDIRK2_ IMEX	250	20000	2	4
	500	40000	2	4
	1000	80000	2	4
SDIRK_FW	250	20000	2	4
	500	40000	2	4
	1000	80000	2	4

With the selected values of $\Delta\tau$ and n_t from Table 7.4, the experiments are carried out on either a CPU or a GPU, with double or single precision. In Figures 7.2-**??**, the results are plotted.

In these figures, the methods labeled with ”FW” correspond to those presented in Section 7.1.3, where only the next time step of h is approximated as a pseudo-time variable. The methods labeled with ”IM” refer to those that use fully explicit time integration within the inner time iterations. The methods labeled with ”IMEX” denote those that employ a semi-explicit scheme within the inner iterations.

Comparing Numerical Errors In the left graph of Figures 7.2 and 7.3 the numerical errors are plotted against the time step size for experiments with double precision on CPU and GPU, respectively. The plotted lines in the left graph of Figure 7.3, are very similar to the lines in Figure 7.2. Therefore, the errors for the methods on the GPU and CPU are similar.

In the left graph of Figures 7.4 and 7.5 the numerical errors are plotted against the time step size for experiments with single precision on a GPU and GPU. Furthermore, we can observe that the plotted lines in the left graph of Figures 7.4 and 7.5, are identical to the plotted lines in the left graph of Figures 7.2 and 7.3, respectively. Hence, in this experiment the errors are identical for single and double precision.

It can be observed that for different implicit time-integration schemes in the outer loop, the errors vary. The lines corresponding to the first-order Backward Euler method are less steep compared to those for the second-order implicit methods. As a result, reducing the time step leads to a smaller error reduction in BE compared to the other second-order implicit methods. This is explained by the fact that the BE-scheme has a first-order error.

Additionally, the lines corresponding to the second-order SDIRK method are closest to the x-axis, indicating that the SDIRK methods exhibit the highest accuracy in our experiments. Furthermore, the CN methods show lower errors compared to the BDF2 methods.

The lines corresponding to different inner explicit time-integration techniques are nearly identical for the same outer time-integration scheme. Therefore, the choice of inner time-integration method does not significantly affect the errors in experiments with different time steps.

Comparing Simulation Times In the right-hand graphs of Figures 7.2 and 7.3, the simulation time is plotted against the time step size for experiments conducted in double precision on the CPU and GPU, respectively. Comparing the computational times, we observe that the experiments on the GPU require more simulation time than those on the CPU. This is because our experiments are designed for coarse grids, which do not fully utilize the parallelism of the GPU, leaving many GPU cores idle.

In the left-hand graphs of Figures 7.4 and 7.5, the simulation time is plotted against the time step size for experiments conducted in single precision on the CPU and GPU, respectively. The lines plotted in the right-hand graphs of Figures 7.4 and 7.5 are very similar to those in the right-hand graphs of Figures 7.3 and 7.3, respectively. Therefore, switching to single precision does not reduce the computational times in our experiments.

Furthermore, for a time step of 250 seconds, the fastest numerical method is the BDF2 scheme with fully explicit inner time integration. Conversely, the SDIRK2 method with semi-explicit inner time integration is the slowest numerical method at this time step. For the largest time step of 1000 seconds, the SDIRK2 method with the Backward-Forward approach is the fastest, while the BE schemes with fully explicit inner time iterations and the Forward-Backward approach are the slowest.

In the experiments, we observe that increasing the outer time step leads to longer run times for some methods, while it reduces run times for others. These changes, however, do not vary linearly with the time step.

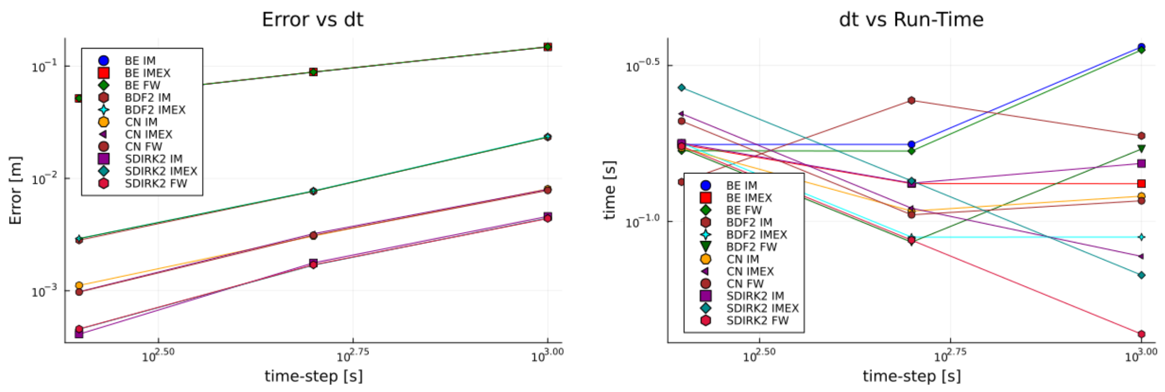


Figure 7.2: Tests on a CPU, double precision.

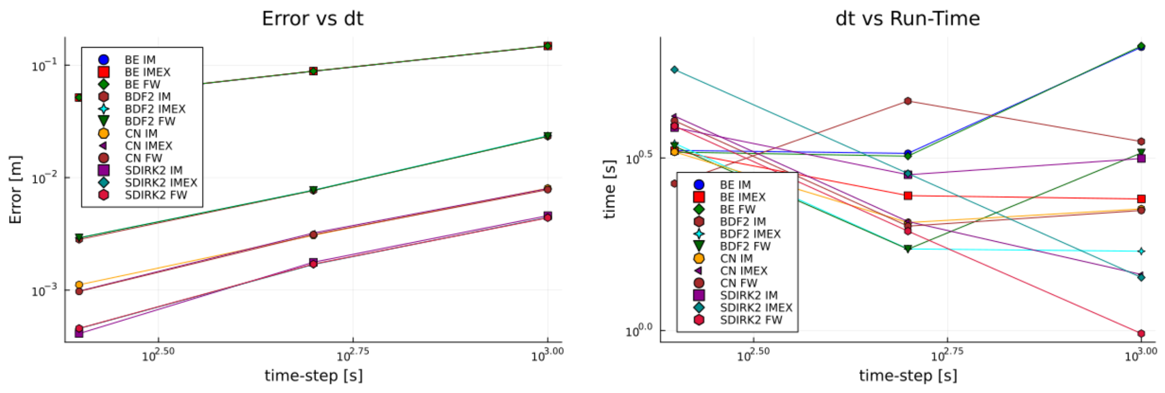


Figure 7.3: Tests-results on a GPU, double precision.

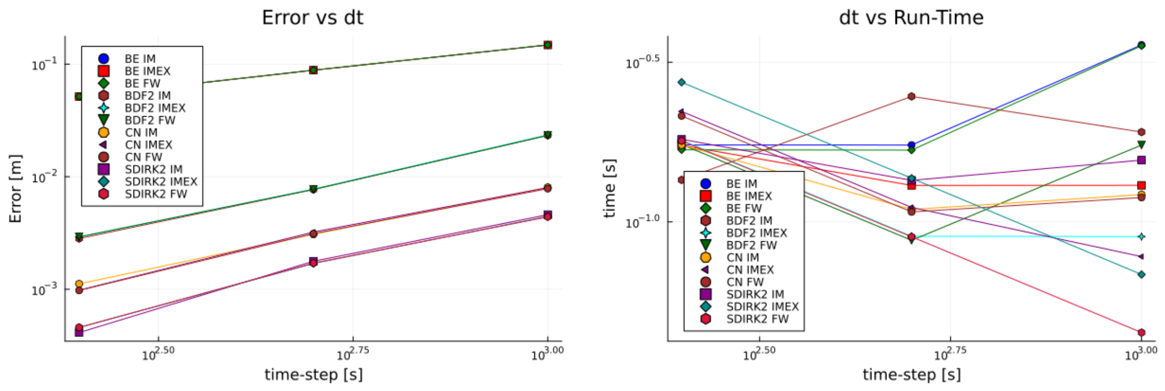


Figure 7.4: Tests-results on a CPU, single precision.

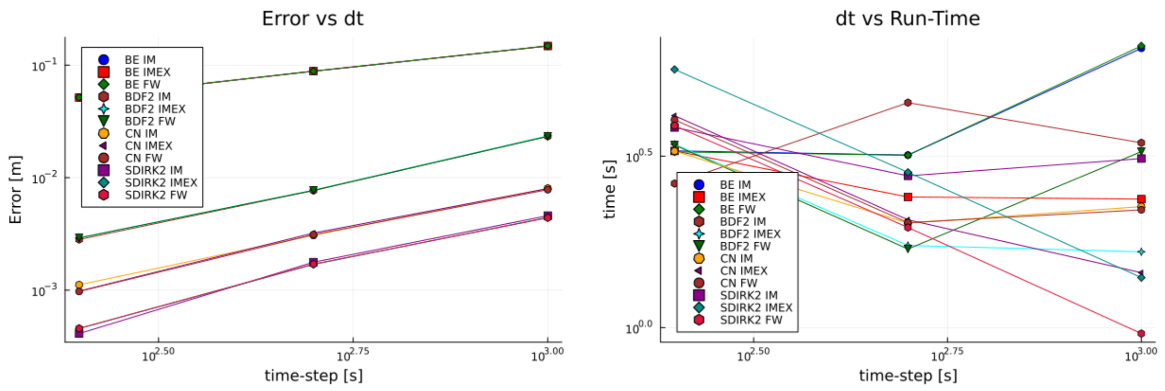


Figure 7.5: Tests-results on a GPU, single precision.

7.3.2. Experiments for different grid-sizes

7.3.3. Experimental Setup

To determine the computational cost of a pseudo-time-stepping technique in the context of using an implicit solver, the following experiments are designed:

- The SWEs are solved with a time step of 100 seconds over a time span of 10 days. This time step is chosen to ensure that the errors of the SDIRK, BDF2, and CN methods remain below 1×10^{-3} on the finest 810×810 grid.
- The experiments are conducted on grids of sizes 30×30, 90×90, 270×270, and 810×810.

- For the every inner time iterations, we evaluate the right-hand side of the pseudo-time dependent ODE corresponding to the right hand side of the h-component, during the inner iterations, using the updated values of h, p, and q. If the infinity norm of this value is smaller than 1×10^{-8} after at least n_t iterations, then we set n_t as the number of inner iterations per outer time step. We choose to let n_t to be the same for solving every outer time step. We let for our experiments n_t be uniform for every outer time step. We determine n_t such that the stopping criterion for the last 20 outer iteration is less than 1×10^{-8} during a 10 day simulation after n_t inner iterations. After having determined n_t, we run experiments with n_t inner iterations for solving every outer time-iteration and measure time.
- To approximating the errors, a fourth-order Runge-Kutta scheme with $\Delta t = 8$ on a 2430×2430 grid is used as the benchmark solution. The errors are computed by taking the infinity norm (Inf-norm) between the benchmark solution and the result of the pseudo-time solve.
- An H100 GPU with 96GB (snellius) is used for the experiments on a GPU. The CPU experiments were done on a AMD EPYC 9334 32-Core Processor (snellius).

In the following table, the number of pseudo-iterations n_t per real-time iteration are shown. Furthermore, the value N_f, represents the number of function evaluations per inner-time iteration.

Table 7.5: Experiments with pseudo-transient, on different grids.

method	n_x	$\Delta\tau$	n_t	N_f
CN_IM	30	10000	3	3
	90	10000	3	3
	270	75	100	101
CN_IMEX	30	20000	2	3
	90	20000	2	3
	270	200	8	9
	810	40	55	56
CN_FW	30	20000	2	3
	90	20000	2	3
	270	75	18	19
	810	6	160	161
BDF2_IM	30	10000	3	3
	90	10000	3	3
	270	35	35	35
BDF2_IMEX	30	20000	2	3
	90	20000	2	3
	270	130	12	12
	810	30	18	18
BDF2_FW	30	20000	2	2
	90	20000	2	2
	270	35	32	32
	810	3	300	300
SDIRK2_IM	30	10000	3	6
	90	10000	3	3
	270	800	78	156
SDIRK2_IMEX	30	10000	2	4
	90	10000	2	4
	270	2000	2	4
	810	80	10	20
BDF2_FW	30	10000	2	4
	90	10000	2	4
	270	800	4	8
	810	20	30	60

With the values of $\Delta\tau$ and n_t from Table 7.5, we do experiments on both GPU and CPU. During those experiments, we measure the error and the computational time.

Error-analysis In Figures 7.6 and 7.7, the results for double precision experiments are shown for the CPU- and GPU-versions, respectively. For double precision experiments, one can observe that for the BDF2-methods the errors are slightly larger on the finest grid, compared to Crank-Nicolson and SDIRK2. Hence, the solver is not robust for the BDF2-method.

In Figures 7.8 and 7.9, the results for single precision experiments are shown for the CPU- and GPU-versions, respectively. If we switch to single precision, the error for implicit Crank-Nicolson with fully explicit inner time-integration explodes on the 270x270 grid. Hence, the fully explicit pseudo-time solver for the implicit Crank-Nicolson scheme is not robust. For the SDIRK-method always an improvement in accuracy is obtained, as the grid becomes finer. The SDIRK-method is the most robust outer time-integration scheme.

Run-Times For experiments on a GPU, the run times do not increase as the grid size increases from 30 to 90. Furthermore, the computational time for the SDIRK2 method with semi-explicit inner time-integration, remains the same we refine the mesh from $n_x = 90$ to $n_x = 270$. For serial experiments on a CPU, different results are obtained. The simulation times on a CPU, do not remain the same for coarse grids, but instead increase linearly. For mesh-sizes with $n_x = 270$ or larger, all the experiments on a CPU, yield a super-linear increase in simulation time. Explanations for this phenomenon can be found in Table 7.5. For every method the number of inner iterations does not increase, if the grid is refined from $n_x = 30$ to $n_x = 90$. For the SDIRK2 with IMEX, the number of inner iterations does not become larger, if we refine the mesh from $n_x = 90$ to $n_x = 270$. If the mesh becomes finer, without an increase in the number of inner iterations, only the number of unknowns increases. For the GPU-version, the larger quantity of unknowns can be solved in parallel, and we do not observe an increase in simulation time. For the serial CPU-version, however, the runtime increases linearly with the number of unknowns in our system.

As the grids become finer than $n_x = 270$, the computational time increases super-linearly with the grid size for the CPU version. For the GPU version, the simulation times start to rise as the mesh is refined from $n_x = 90$ to 270 (except for SDIRK2_IMEX) and from 270 to 810.

For both the Float32 and Float64 versions, as well as the CPU and GPU implementations, we observe that the SDIRK2 (IMEX) method with semi-explicit inner time integration results in the shortest computational times for both $n_x = 270$ and $n_x = 810$.

Given that the IMEX solver of SDIRK2 demonstrates the highest robustness and achieves the shortest runtime on the finest grid, we propose it as the preferred method. In the following chapter, a multilevel technique will be applied to the SDIRK2 scheme to further improve its performance.

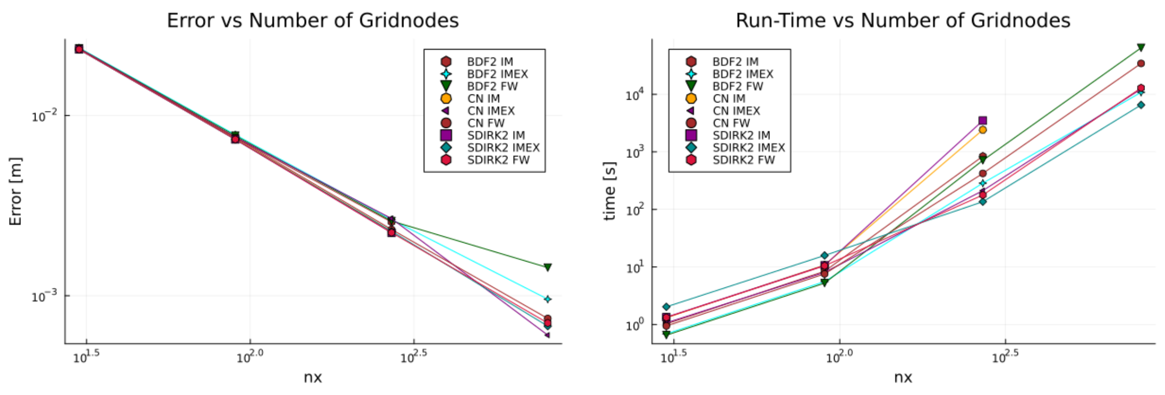


Figure 7.6: Tests-results on a CPU, double precision.

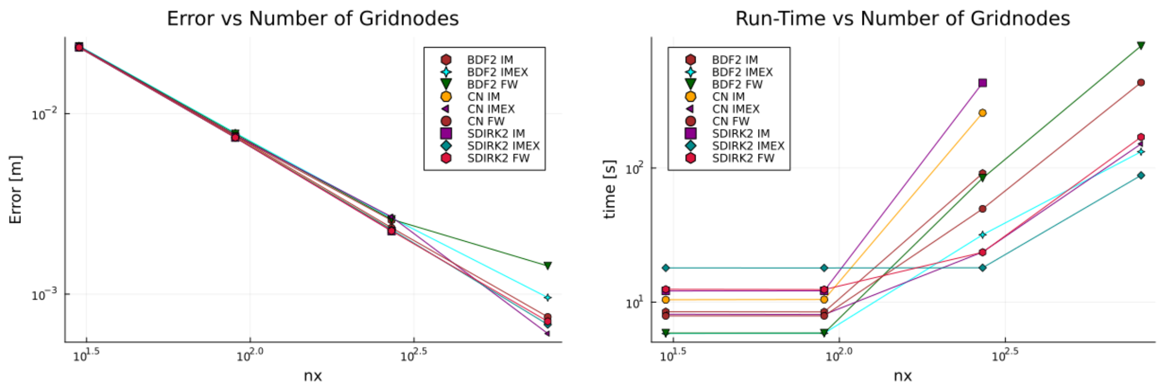


Figure 7.7: Tests-results on a GPU, double precision.

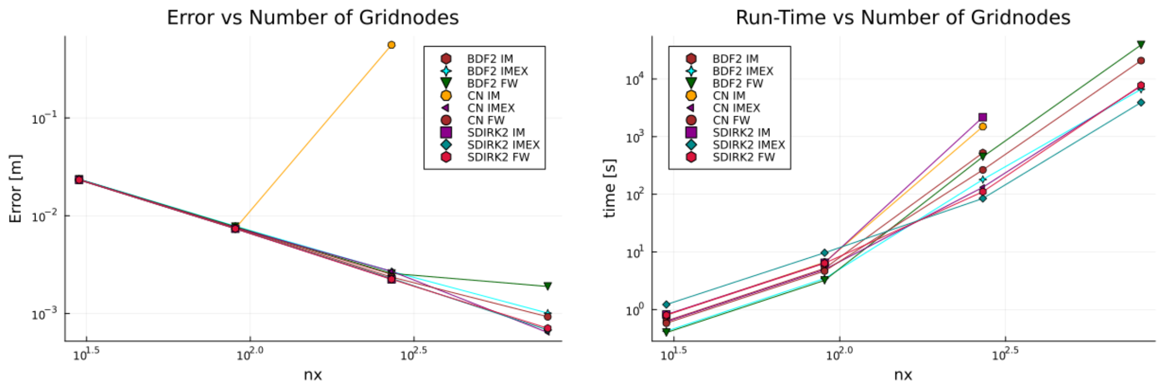


Figure 7.8: Tests-results on a CPU, single precision.

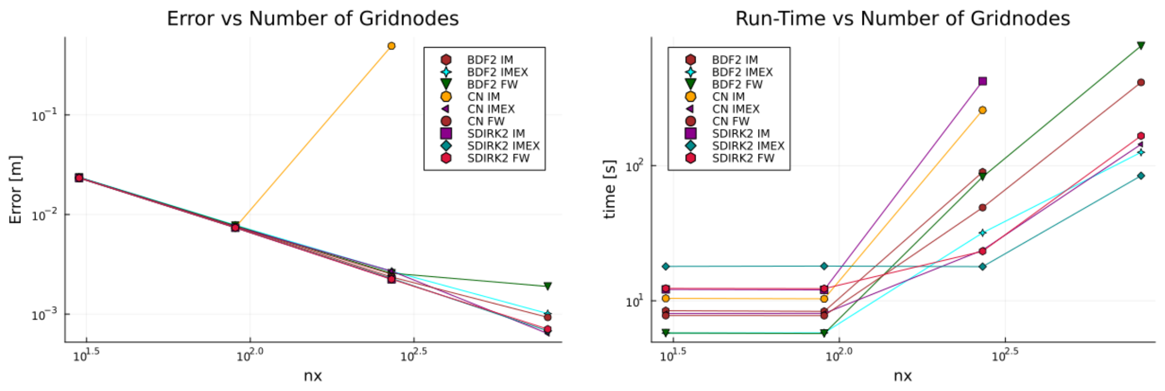


Figure 7.9: Tests-results on a GPU, single precision.

8 Incorporating a Multilevel Technique into the SDIRK2_IMEX Pseudo-Time Solver

In the previous chapter, we introduced pseudo-time-stepping techniques to solve for the unknowns in the outer time iteration of an implicit scheme. Various implicit schemes, incorporating different methods for pseudo-time integration, were evaluated and compared. The second-order SDIRK method, combined with an IMEX-style scheme for the inner time integration, emerged as the most efficient and accurate approach for both the 270-by-270 and 810-by-810 mesh configurations. Nevertheless, if one increases grid resolution from 270-by-270 to 810-by-810, a larger number of inner iterations are required for convergence. Consequently, grid refinement leads to an increase in computational work, not only due to the higher number of unknowns but also due to the increasing number of inner iterations. This increase in number of pseudo-time-iterations is a disadvantage on a GPU, since only the work within every time iteration can be done in parallel. For grid sizes where not all cores are fully utilized, the number of iterations has a bigger impact on computational times than the grid size, since increasing grid refinement will only lead to the use of more cores. In contrast, for mesh configurations where all GPU cores are already fully occupied, increasing the grid size also leads to longer computational times due to the need to solve a larger system.

To solve a system on a fine grid with a large number of unknowns, an approximation of the solution can be computed on a coarser grid. This approximation can then be interpolated to the fine grid and used as an initial guess for the fine-grid solution. The process of solving a system using solutions across multiple scales is known as a multilevel method. By applying the multilevel-technique to the SDIRK2-solver, a coarse-grid solution can be derived with fewer inner iterations and used as an approximation for the fine-grid solution. Conceivably, fewer inner iterations for acquiring a fine-grid-solutions are needed. Furthermore, if the amount of work that can be parallelized on the fine-grid problem exceeds the amount of work that can be carried out by all the available GPU-cores in parallel, obtaining a coarse-grid approximation with perfect parallelism could further decrease simulation time. Hence, the objective of this chapter is to present a multilevel strategy for reducing time complexity within a SDIRK2-solver.

Firstly, in Section 8.1, the concept of a multilevel technique is introduced. Secondly, in Section 8.2, we describe the process of interpolating coarse-grid estimates to finer grids and restricting fine-grid solutions to coarser grids. Finally, in Section 8.3, various tests for evaluating time complexity are presented.

8.1. Solving Implicit Time Iterations with a Multilevel Strategy

In this Section, a strategy for solving unknowns in implicit time-integration schemes for solving (SWEs) with high grid resolution, utilizing approximations on coarse grids, is presented. On a coarse mesh, a solution is obtained and used as an approximation for solving the system on a fine grid.

Firstly, Subsection 8.1.1 provides an explanation of the multilevel concept. Secondly, Subsection 8.1.2 presents the nonlinear problem arising from an implicit time iteration, which is to be solved on different grid sizes. Finally, Subsection 8.1.3 applies the multilevel method to the SDIRK2 solver.

8.1.1. Integrating Multilevel Techniques with Solvers for Nonlinear Systems

After discretizing SWEs on a uniform spatial grid with Δx-by-Δx cells and applying an implicit time-integration scheme, we find the next time iteration by finding the root of a nonlinear system. In Chapter 7 a pseudo-time solver was selected for solving the nonlinear system. In this Subsection, we assume that an iterative solver for the nonlinear problem is available at all resolutions, to simplify the discussion of the multilevel approach.

The nonlinear systems solved in chapter 7 have the following structure:

$$\mathbf{F}^{\Delta x}(\mathbf{y}^{\Delta x}) = \mathbf{0}, \tag{8.1}$$

where $\mathbf{F}^{\Delta x}$ and $\mathbf{y}^{\Delta x}$ represent the nonlinear system and the root of the nonlinear system for the fine grid problem, respectively. The fine grid is a n-by-n mesh with cell-size Δx in both x-and y- directions. The superscript in (8.1) Δx denotes that this nonlinear problem is defined on mesh with Δx-by-Δx cells.

The System (8.1) is solved with both a solver and a multilevel method. Using the solver, the Equation (8.1) is redefined and solved on coarser grids. The approximations on the coarse grid are transferred to fine meshes and used as initial approximations for solving (8.1) on finer grids. In our work, a mesh configuration that is one level finer contains a factor of two more cells in both the x- and y-directions, resulting in a total of four times more cells. Figure 8.1 depicts the steps involved in a three-level multilevel approach. The details of the multilevel procedure are outlined in the following paragraphs.

Firstly, on the finest grid, the initial approximation of the root of Problem (8.1) is denoted by $\bar{\mathbf{y}}^{\Delta x}$. This approximation is transferred to a coarser grid twice.

The mid-level of the multilevel-technique is defined on a $\frac{n}{2}$-by-$\frac{n}{2}$ mesh, with cell sizes equal to $2\Delta x$-by-$2\Delta x$. The initial guess $\bar{\mathbf{y}}^{\Delta x}$ on the finest grid is transferred to this coarser grid: as follows:

$$\bar{\mathbf{y}}^{2\Delta x} = I_{\Delta x}^{2\Delta x} \bar{\mathbf{y}}^{\Delta x}, \tag{8.2}$$

where $I_{\Delta x}^{2\Delta x}$ is the restriction operator that transfers the initial guess from the fine grid with spatial step Δx to the coarse grid with with cell size $2\Delta x$

After restriction to the mid-level grid, the approximation $\bar{\mathbf{y}}^{2\Delta}$ is further transferred to a coarser grid. The third level, representing the coarsest grid, of the of a three-level multigrid method is defined on a $\frac{n}{4}$-by-$\frac{n}{4}$ mesh, with cells of size $4\Delta x$-by-$4\Delta x$. The initial guess from the mid-level grid is transferred to the coarsest grid as follows:

$$\hat{\mathbf{y}}^{4\Delta x} = I_{2\Delta x}^{4\Delta x} \bar{\mathbf{y}}^{2\Delta x}, \tag{8.3}$$

where $I_{2\Delta x}^{4\Delta x}$ is the transfer operator from the $\frac{n}{2}$-by-$\frac{n}{2}$ with cell size $2\Delta x$ to $\frac{n}{4}$-by-$\frac{n}{4}$ grid with cell size $2\Delta x$. The variable $\hat{\mathbf{y}}^{4\Delta x}$ denotes the initial estimate to solve the coarse grid problem.

On the coarsest level, we define $\mathbf{F}^{4\Delta x}$ as the rediscretized problem. As follows, the next equation is solved using a solver:

$$\mathbf{F}^{4\Delta x}(\mathbf{y}^{4\Delta x}) = \mathbf{0}, \tag{8.4}$$

where $\mathbf{y}^{4\Delta x}$ is the solution on the coarsest grid. The initial estimate for solving (8.4) is given by: $\hat{\mathbf{y}}^{4\Delta x}$.

After obtaining a solution on the coarsest grid, it is transferred back to the mid-level grid and used as an initial approximation for the mid-level problem: As follows:

$$\hat{\mathbf{y}}^{2\Delta x} = I_{4\Delta x}^{2\Delta x} \mathbf{y}^{4\Delta x}, \tag{8.5}$$

where $I_{4\Delta x}^{2\Delta x}$ is the interpolation operator from the $\frac{n}{4}$-by-$\frac{n}{4}$-grid to $\frac{n}{2}$-by-$\frac{n}{2}$-grid. The variable $\hat{\mathbf{y}}^{2\Delta x}$ denotes the initial estimate to solve the problem on the mid-level grid.

Furthermore, the nonlinear problem is rediscretized on a $\frac{n}{2}$-by-$\frac{n}{2}$-grid and is denoted by $\mathbf{F}^{2\Delta x}$. The initial approximation of the mid-level problem is $\hat{\mathbf{y}}^{2\Delta x}$. For the mid-level problem we find the root of the following equation using a solver:

$$\mathbf{F}^{2\Delta x}(\mathbf{y}^{2\Delta x}) = \mathbf{0}, \tag{8.6}$$

where $\mathbf{y}^{2\Delta x}$ is the solution on the mid-level grid. The initial estimate for solving (8.6) is given by: $\hat{\mathbf{y}}^{2\Delta x}$.

Thereafter, the mid-level solution is interpolated to the finest grid:

$$\hat{\mathbf{y}}^{\Delta x} = I_{2\Delta x}^{\Delta x}\mathbf{y}^{2\Delta x}, \tag{8.7}$$

where $I_{2\Delta x}^{\Delta x}$ is the transfer operator from the $\frac{n}{2}$-by-$\frac{n}{2}$-grid to n-by-n-grid. The approximation $\hat{\mathbf{y}}^{\Delta x}$ is used as the initial estimate to solve the fine grid problem in (8.1).

Finally, the fine-grid problem given by Equation (8.1) is solved using the interpolated mid-level solution $\hat{\mathbf{y}}^{\Delta x}$ as an initial approximation.

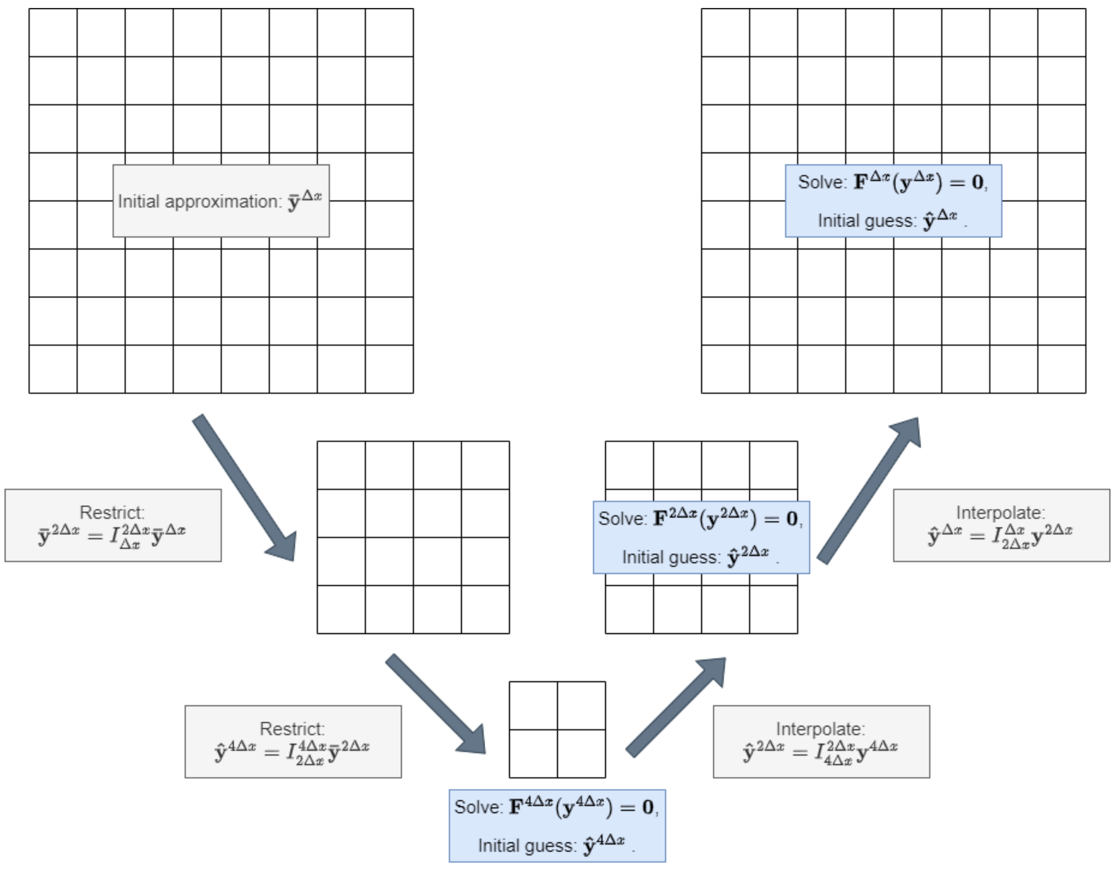

Initial approximation: $\bar{\mathbf{y}}^{\Delta x}$
Solve: $\mathbf{F}^{\Delta x}(\mathbf{y}^{\Delta x}) = \mathbf{0}$, Initial guess: $\hat{\mathbf{y}}^{\Delta x}$.
Restrict: $\bar{\mathbf{y}}^{2\Delta x} = I_{\Delta x}^{2\Delta x}\bar{\mathbf{y}}^{\Delta x}$
Solve: $\mathbf{F}^{2\Delta x}(\mathbf{y}^{2\Delta x}) = \mathbf{0}$, Initial guess: $\hat{\mathbf{y}}^{2\Delta x}$.
Interpolate: $\hat{\mathbf{y}}^{\Delta x} = I_{2\Delta x}^{\Delta x}\mathbf{y}^{2\Delta x}$
Restrict: $\hat{\mathbf{y}}^{4\Delta x} = I_{2\Delta x}^{4\Delta x}\bar{\mathbf{y}}^{2\Delta x}$
Interpolate: $\hat{\mathbf{y}}^{2\Delta x} = I_{4\Delta x}^{2\Delta x}\mathbf{y}^{4\Delta x}$
Solve: $\mathbf{F}^{4\Delta x}(\mathbf{y}^{4\Delta x}) = \mathbf{0}$, Initial guess: $\hat{\mathbf{y}}^{4\Delta x}$.

Figure 8.1: Solving a nonlinear system on three levels

8.1.2. Nonlinear Problems Arising from SDIRK2 Iterations

A solver can be combined with a multilevel technique to determine the root of a nonlinear system. In the previous chapter, a pseudo-time-stepping method was introduced for solving a nonlinear system arising from an implicit time-stepping scheme. For the implicit second-order SDIRK-scheme, two nonlinear systems have to be solved for two stage variables. These two implicit schemes can be solved using an IMEX-style pseudo-time-stepping method. The initial approximations for stage variables within the solver are selected to be the values of the stage variables found in the previous (outer) time iteration.

We combine the pseudo-time solver for solving the two nonlinear systems for the SDIRK-scheme with a multilevel approach, similar as shown in Figure (8.1). The two stage variables from the previous time iteration are transferred to coarser grids and used as initial approximations for solving a coarse grid problem. After that, the stage variables are obtained on a coarse grid and interpolated to fine grids. Finally, on fine grids the interpolated stage variables are used as initial guesses for the pseudo-time solver on the fine grid.

In this Subsection, we present the nonlinear systems to be solved using a pseudo-transient scheme combined with a multilevel method. Solving these nonlinear systems ultimately yields the solution for the next (outer) time step of the implicit second-order SDIRK2 method.

Firstly, Equations (8.8)-(8.10) are solved on the fine grid for the stage variables $\mathbf{k}_{1,h}^{\Delta x}$,$\mathbf{k}_{1,p}^{\Delta x}$ and $\mathbf{k}_{1,q}^{\Delta x}$, respectively.

$$\mathbf{k}_{1,h}^{\Delta x} = \mathbf{F}_h^{\Delta x}\left(\mathbf{p}^{n,\Delta x} + \gamma\Delta t\mathbf{k}_{1,p}^{\Delta x}\right) + G_h^{\Delta x}\left(\mathbf{q}^{n,\Delta x} + \gamma\Delta t\mathbf{k}_{1,q}^{\Delta x}\right), \tag{8.8}$$

$$\mathbf{k}_{1,p}^{\Delta x} = \mathbf{F}_p^{\Delta x}\left(\mathbf{p}^{n,\Delta x} + \gamma\Delta t\mathbf{k}_{1,h}^{\Delta x}, \mathbf{p}^{n,\Delta x} + \gamma\Delta t\mathbf{k}_{1,p}^{\Delta x}, \mathbf{q}^{n,\Delta x} + \gamma\Delta t\mathbf{k}_{1,q}^{\Delta x}, t^n + \gamma\Delta t\right), \tag{8.9}$$

$$\mathbf{k}_{1,q}^{\Delta x} = \mathbf{F}_q^{\Delta x}\left(\mathbf{h}^{n,\Delta x} + \gamma\Delta t\mathbf{k}_{1,h}^{\Delta x}, \mathbf{p}^{n,\Delta x} + \gamma\Delta t\mathbf{k}_{1,p}^{\Delta x}, \mathbf{q}^{n,\Delta x} + \gamma\Delta t\mathbf{k}_{1,q}^{\Delta x}\right), \tag{8.10}$$

where constant γ is chosen to be equal to $1 - \frac{1}{\sqrt{2}}$. Equations (8.8)-(8.10) are similar to Problem (7.25)-(7.27), defined on a spatial grid with Δx-by-Δx grid-cells. The stage variables $\mathbf{k}_{1,h}^{\Delta x}$, $\mathbf{k}_{1,p}^{\Delta x}$ and $\mathbf{k}_{1,q}^{\Delta x}$ are similar to the variables $k_{1,h}$, $k_{1,p}$ and $k_{1,q}$, respectively, discretized on Δx-by-Δx grid-cells. The solutions in the previous time step t^n, on the Δx-by-Δx meshes, are given by $\mathbf{h}^{n,\Delta x}$, $\mathbf{p}^{n,\Delta x}$ and $\mathbf{q}^{n,\Delta x}$.

Secondly, to obtain the solutions in the next time step using the SDIRK2-method, one has to solve the nonlinear system given by Equations (8.11)-(8.13) for the second stages.

$$\mathbf{k}_{2,h}^{\Delta x} = \mathbf{F}_h^{\Delta x}\Big(\mathbf{p}^{n,\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,p}^{\Delta x} + \gamma\mathbf{k}_{2,p}^{\Delta x})\Big) + \mathbf{G}_h^{\Delta x}\Big(\mathbf{q}^{n,\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,q}^{\Delta x} + \gamma\mathbf{k}_{2,q}^{\Delta x})\Big), \tag{8.11}$$

$$\mathbf{k}_{2,p}^{\Delta x} = \mathbf{F}_p^{\Delta x}\Big(\mathbf{h}^{n,\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,h}^{\Delta x} + \gamma\mathbf{k}_{2,h}\Delta x))), \tag{8.12}$$

$$\mathbf{p}^{n,\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,p}^{\Delta x} + \gamma\mathbf{k}_{2,p}^{\Delta x})), \mathbf{q}^{n,\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,q}^{\Delta x} + \gamma\mathbf{k}_{2,q}^{\Delta x}), t^n + \Delta t\Big),$$

$$\mathbf{k}_{2,q}^{\Delta x} = \mathbf{F}_q^{\Delta x}\Big(\mathbf{h}^{n,\Delta x} + \Delta t(((1-\gamma)\mathbf{k}_{1,h}^{\Delta x} + \gamma\mathbf{k}_{2,h}^{\Delta x}))), \tag{8.13}$$

$$\mathbf{p}^{n,\Delta x} + \gamma\Delta t((1-\gamma)\mathbf{k}_{1,p}^{\Delta x} + \gamma\mathbf{k}_{2,p}^{\Delta x})), \mathbf{q}^{n,\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,q}^{\Delta x} + \gamma\mathbf{k}_{2,q}^{\Delta x})\Big),$$

The stage variables $\mathbf{k}_{2,h}^{\Delta x}$, $\mathbf{k}_{2,p}^{\Delta x}$ and $\mathbf{k}_{2,q}^{\Delta x}$ are similar to the variables $k_{2,h}$, $k_{2,p}$ and $k_{2,q}$, respectively, discretized on Δx-by-Δx grid-cells. Equations (8.11)-(8.13) are similar to Problem (7.28)-(7.31), defined on a spatial grid with Δx-by-Δx mesh-cells.

Finally, in Equations (8.14) a weighted average is computed between the two stages to obtain the numerical solution at the next time step.

$$\mathbf{h}^{n+1,\Delta x} = \mathbf{h}^{n,\Delta x} + \Delta t\left((1-\gamma)\mathbf{k}_{1,h}^{\Delta x} + \gamma\mathbf{k}_{2,h}^{\Delta x}\right), \tag{8.14}$$

$$\mathbf{p}^{n+1,\Delta x} = \mathbf{p}^{n,\Delta x} + \Delta t\left((1-\gamma)\mathbf{k}_{1,p}^{\Delta x} + \gamma\mathbf{k}_{2,p}^{\Delta x}\right), \tag{8.15}$$

$$\mathbf{q}^{n+1,\Delta x} = \mathbf{q}^{n,\Delta x} + \Delta t\left((1-\gamma)\mathbf{k}_{1,q}^{\Delta x} + \gamma\mathbf{k}_{2,q}^{\Delta x}\right), \tag{8.16}$$

where $\mathbf{h}^{n+1,\Delta x}$, $\mathbf{p}^{n+1,\Delta x}$ and $\mathbf{q}^{n+1,\Delta x}$ represent the solutions for variables h, p and q, respectively, in the time step $n+1$, on the mesh with cell size Δx.

8.1.3. Solving SDIRK2-iterations with a multilevel strategy

In Subsection 8.1.1, the multilevel-strategy is outlined. This multilevel technique will be used to solve Equation (8.8)-(8.13). Since in the Equations (8.11)-(8.13), the unknowns are given by the stage variables, the stage variables in the multilevel method are the variables that will be interpolated and restricted to different grids.

First of all, on the fine grid level we have the following approximations for the stage variables which are similar to those in Subsection 7.1.2. These initial approximations (on the fine grid) are denoted by:$\bar{\mathbf{k}}_{1,h}^{\Delta x}$, $\bar{\mathbf{k}}_{1,p}^{\Delta x}$, $\bar{\mathbf{k}}_{1,q}^{\Delta x}$, $\bar{\mathbf{k}}_{2,h}^{\Delta x}$, $\bar{\mathbf{k}}_{2,p}^{\Delta x}$ and $\bar{\mathbf{k}}_{2,q}^{\Delta x}$.

We restrict the initial approximations for the stages to a coarser grid:

$$\bar{\mathbf{k}}_{1,h}^{2\Delta x} = I_{h,\Delta x}^{2\Delta x}\bar{\mathbf{k}}_{1,h}^{\Delta x}, \quad \bar{\mathbf{k}}_{1,p}^{2\Delta x} = I_{p,\Delta x}^{2\Delta x}\bar{\mathbf{k}}_{1,p}^{\Delta x}, \quad \bar{\mathbf{k}}_{1,q}^{2\Delta x} = I_{q,\Delta x}^{2\Delta x}\bar{\mathbf{k}}_{1,q}^{\Delta x}, \tag{8.17}$$

$$\bar{\mathbf{k}}_{2,h}^{2\Delta x} = I_{h,\Delta x}^{2\Delta x}\bar{\mathbf{k}}_{2,h}^{\Delta x}, \quad \bar{\mathbf{k}}_{2,p}^{2\Delta x} = I_{p,\Delta x}^{2\Delta x}\bar{\mathbf{k}}_{2,p}^{\Delta x}, \quad \bar{\mathbf{k}}_{2,q}^{2\Delta x} = I_{q,\Delta x}^{2\Delta x}\bar{\mathbf{k}}_{2,q}^{\Delta x}, \tag{8.18}$$

where $I_{h,\Delta x}^{2\Delta x}$, $I_{p,\Delta x}^{2\Delta x}$ and $I_{q,\Delta x}^{2\Delta x}$ are the restriction operators corresponding to the fine grid-nodes for h, p and q on the staggered grid.

Furthermore, we also restrict the previous time-step solution to a coarser grid:

$$\mathbf{h}^{n,2\Delta x} = I_{h,\Delta x}^{2\Delta x}\mathbf{h}^{n,\Delta x}, \quad \mathbf{p}^{n,2\Delta x} = I_{p,\Delta x}^{2\Delta x}\mathbf{p}^{n,\Delta x}, \quad \mathbf{q}^{n,2\Delta x} = I_{q,\Delta x}^{2\Delta x}\mathbf{q}^{n,\Delta x}. \tag{8.19}$$

Secondly, we again transfer both the mid-level approximations for the stages and the restricted previous time-step solutions to a coarser grid.

$$\hat{\mathbf{k}}_{1,h}^{4\Delta x} = I_{h,2\Delta x}^{4\Delta x}\bar{\mathbf{k}}_{1,h}^{2\Delta x}, \quad \bar{\mathbf{k}}_{1,p}^{4\Delta x} = I_{p,2\Delta x}^{4\Delta x}\bar{\mathbf{k}}_{1,p}^{2\Delta x}, \quad \hat{\mathbf{k}}_{1,q}^{4\Delta x} = I_{q,4\Delta x}^{2\Delta x}\bar{\mathbf{k}}_{1,q}^{2\Delta x}, \tag{8.20}$$

$$\hat{\mathbf{k}}_{2,h}^{4\Delta x} = I_{h,2\Delta x}^{4\Delta x}\bar{\mathbf{k}}_{2,h}^{2\Delta x}, \quad \bar{\mathbf{k}}_{2,p}^{4\Delta x} = I_{p,2\Delta x}^{4\Delta x}\bar{\mathbf{k}}_{2,p}^{2\Delta x}, \quad \hat{\mathbf{k}}_{2,q}^{4\Delta x} = I_{q,2\Delta x}^{4\Delta x}\bar{\mathbf{k}}_{2,q}^{2\Delta x}, \tag{8.21}$$

$$\mathbf{h}^{n,4\Delta x} = I_{h,2\Delta x}^{4\Delta x}\mathbf{h}^{n,2\Delta x}, \quad \mathbf{p}^{n,4\Delta x} = I_{p,2\Delta x}^{4\Delta x}\mathbf{p}^{n,2\Delta x}, \quad \mathbf{q}^{n,4\Delta x} = I_{q,2\Delta x}^{4\Delta x}\mathbf{q}^{n,2\Delta x}. \tag{8.22}$$

the Problem (8.8)-(8.10) is redefined on the coarsest grid and solved using pseudo-time integration (using IMEX, as defined in Subsection 7.1.2). Equations (8.23)-(8.25) are solved for the stage variables $\mathbf{k}_{1,h}^{4\Delta x}$,$\mathbf{k}_{1,p}^{4\Delta x}$ and $\mathbf{k}_{1,q}^{4\Delta x}$.

$$\mathbf{k}_{1,h}^{4\Delta x} = \mathbf{F}_h^{4\Delta x}\left(\mathbf{p}^{n,4\Delta x} + \gamma\Delta t\mathbf{k}_{1,p}^{4\Delta x}\right) + \mathbf{G}_h^{4\Delta x}\left(\mathbf{q}^{n,4\Delta x} + \gamma\Delta t\mathbf{k}_{1,q}^{4\Delta x}\right), \tag{8.23}$$

$$\mathbf{k}_{1,p}^{4\Delta x} = \mathbf{F}_p^{4\Delta x}\left(\mathbf{p}^{n,4\Delta x} + \gamma\Delta t\mathbf{k}_{1,h}^{4\Delta x}, \mathbf{p}^{n,4\Delta x} + \gamma\Delta t\mathbf{k}_{1,p}^{4\Delta x}, \mathbf{q}^{n,4\Delta x} + \gamma\Delta t\mathbf{k}_{1,q}^{4\Delta x}, t^n + \gamma\Delta t\right), \tag{8.24}$$

$$\mathbf{k}_{1,q}^{4\Delta x} = \mathbf{F}_q^{4\Delta x}\left(\mathbf{h}^{n,4\Delta x} + \gamma\Delta t\mathbf{k}_{1,h}^{4\Delta x}, \mathbf{p}^{n,\Delta x} + \gamma\Delta t\mathbf{k}_{1,p}^{4\Delta x}, \mathbf{q}^{n,4\Delta x} + \gamma\Delta t\mathbf{k}_{1,q}^{4\Delta x}\right), \tag{8.25}$$

Furthermore, we also solve for $\mathbf{k}_{2,h}^{4\Delta x}$,$\mathbf{k}_{2,p}^{4\Delta x}$ and $\mathbf{k}_{2,q}^{4\Delta x}$ on a redefined coarse grid problem for the second stages:

$$\mathbf{k}_{2,h}^{4\Delta x} = \mathbf{F}_h^{4\Delta x}\Big(\mathbf{p}^{n,4\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,p}^{4\Delta x} + \gamma\mathbf{k}_{2,p}^{4\Delta x})\Big) + \mathbf{G}_h^{4\Delta x}\Big(\mathbf{q}^{n,4\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,q}^{4\Delta x} + \gamma\mathbf{k}_{2,q}^{4\Delta x}))\Big), \tag{8.26}$$

$$\mathbf{k}_{2,p}^{4\Delta x} = \mathbf{F}_p^{4\Delta x}\Big(\mathbf{h}^{n,4\Delta x} + \Delta t(((1-\gamma)\mathbf{k}_{1,h}^{4\Delta x} + \gamma\mathbf{k}_{2,h}4\Delta x))), \tag{8.27}$$

$$\mathbf{p}^{n,4\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,p}^{4\Delta x} + \gamma\mathbf{k}_{2,p}^{4\Delta x})), \mathbf{q}^{n,4\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,q}^{4\Delta x} + \gamma\mathbf{k}_{2,q}^{4\Delta x}), t^n + \Delta t\Big),$$

$$\mathbf{k}_{2,q}^{4\Delta x} = \mathbf{F}_q^{4\Delta x}\Big(\mathbf{h}^{n,4\Delta x} + \Delta t(((1-\gamma)\mathbf{k}_{1,h}^{4\Delta x} + \gamma\mathbf{k}_{2,h}^{4\Delta x}))), \tag{8.28}$$

$$\mathbf{p}^{n,4\Delta x} + \gamma\Delta t((1-\gamma)\mathbf{k}_{1,p}^{\Delta x} + \gamma\mathbf{k}_{2,p}^{4\Delta x})), \mathbf{q}^{n,4\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,q}^{4\Delta x} + \gamma\mathbf{k}_{2,q}^{4\Delta x})\Big).$$

After obtaining the coarse-grid solutions $\mathbf{k}_{1,h}^{4\Delta x}$, $\mathbf{k}_{1,p}^{4\Delta x}$, $\mathbf{k}_{1,q}^{4\Delta x}$, $\mathbf{k}_{2,h}^{4\Delta x}$, $\mathbf{k}_{2,p}^{4\Delta x}$ and $\mathbf{k}_{2,q}^{4\Delta x}$. These stages are interpolated to a coarser grid and used as initial approximations for the mid-level problem.

$$\bar{\mathbf{k}}_{1,h}^{2\Delta x} = I_{h,4\Delta x}^{2\Delta x}\mathbf{k}_{1,h}^{4\Delta x}, \quad \bar{\mathbf{k}}_{1,p}^{2\Delta x} = I_{p,4\Delta x}^{2\Delta x}\mathbf{k}_{1,p}^{4\Delta x}, \quad \bar{\mathbf{k}}_{1,q}^{2\Delta x} = I_{q,4\Delta x}^{2\Delta x}\mathbf{k}_{1,q}^{4\Delta x}, \tag{8.29}$$

$$\bar{\mathbf{k}}_{2,h}^{2\Delta x} = I_{h,4\Delta x}^{2\Delta x}\mathbf{k}_{2,h}^{4\Delta x}, \quad \bar{\mathbf{k}}_{2,p}^{2\Delta x} = I_{p,4\Delta x}^{2\Delta x}\mathbf{k}_{2,p}^{4\Delta x}, \quad \bar{\mathbf{k}}_{2,q}^{2\Delta x} = I_{q,4\Delta x}^{2\Delta x}\mathbf{k}_{2,q}^{4\Delta x}, \tag{8.30}$$

Following the previous restriction steps, Problem (8.8)-(8.10) is redefined on the coarsest gid and solved using pseudo-time integration (using IMEX, as defined in Subsection 7.1.2). Equations (8.31)-(8.33) are solved for the stage variables $\mathbf{k}_{1,h}^{2\Delta x}$,$\mathbf{k}_{1,p}^{2\Delta x}$ and $\mathbf{k}_{1,q}^{2\Delta x}$.

$$\mathbf{k}_{1,h}^{2\Delta x} = \mathbf{F}_h^{2\Delta x}\left(\mathbf{p}^{n,2\Delta x} + \gamma\Delta t\mathbf{k}_{1,p}^{2\Delta x}\right) + \mathbf{G}_h^{4\Delta x}\left(\mathbf{q}^{n,2\Delta x} + \gamma\Delta t\mathbf{k}_{1,q}^{2\Delta x}\right), \tag{8.31}$$

$$\mathbf{k}_{1,p}^{2\Delta x} = \mathbf{F}_p^{2\Delta x}\left(\mathbf{p}^{n,2\Delta x} + \gamma\Delta t\mathbf{k}_{1,h}^{2\Delta x}, \mathbf{p}^{n,2\Delta x} + \gamma\Delta t\mathbf{k}_{1,p}^{2\Delta x}, \mathbf{q}^{n,2\Delta x} + \gamma\Delta t\mathbf{k}_{1,q}^{2\Delta x}, t^n + \gamma\Delta t\right), \tag{8.32}$$

$$\mathbf{k}_{1,q}^{2\Delta x} = \mathbf{F}_q^{2\Delta x}\left(\mathbf{h}^{n,2\Delta x} + \gamma\Delta t\mathbf{k}_{1,h}^{2\Delta x}, \mathbf{p}^{n,\Delta x} + \gamma\Delta t\mathbf{k}_{1,p}^{2\Delta x}, \mathbf{q}^{n,2\Delta x} + \gamma\Delta t\mathbf{k}_{1,q}^{2\Delta x}\right), \tag{8.33}$$

Furthermore, we also solve for $\mathbf{k}_{2,h}^{2\Delta x}$,$\mathbf{k}_{2,p}^{2\Delta x}$ and $\mathbf{k}_{2,q}^{2\Delta x}$ on a redefined coarse grid problem for the second stages:

$$\mathbf{k}_{2,h}^{2\Delta x} = \mathbf{F}_h^{2\Delta x}\Big(\mathbf{p}^{n,2\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,p}^{2\Delta x} + \gamma\mathbf{k}_{2,p}^{2\Delta x})\Big) + \mathbf{G}_h^{2\Delta x}\Big(\mathbf{q}^{n,2\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,q}^{2\Delta x} + \gamma\mathbf{k}_{2,q}^{2\Delta x}))\Big), \tag{8.34}$$

$$\mathbf{k}_{2,p}^{2\Delta x} = \mathbf{F}_p^{2\Delta x}\Big(\mathbf{h}^{n,2\Delta x} + \Delta t(((1-\gamma)\mathbf{k}_{1,h}^{2\Delta x} + \gamma\mathbf{k}_{2,h}2\Delta x))), \tag{8.35}$$

$$\mathbf{p}^{n,2\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,p}^{2\Delta x} + \gamma\mathbf{k}_{2,p}^{2\Delta x})), \mathbf{q}^{n,2\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,q}^{2\Delta x} + \gamma\mathbf{k}_{2,q}^{2\Delta x}), t^n + \Delta t\Big),$$

$$\mathbf{k}_{2,q}^{2\Delta x} = \mathbf{F}_q^{2\Delta x}\Big(\mathbf{h}^{n,2\Delta x} + \Delta t(((1-\gamma)\mathbf{k}_{1,h}^{2\Delta x} + \gamma\mathbf{k}_{2,h}^{2\Delta x}))), \tag{8.36}$$

$$\mathbf{p}^{n,2\Delta x} + \gamma\Delta t((1-\gamma)\mathbf{k}_{1,p}^{\Delta x} + \gamma\mathbf{k}_{2,p}^{2\Delta x})), \mathbf{q}^{n,2\Delta x} + \Delta t((1-\gamma)\mathbf{k}_{1,q}^{2\Delta x} + \gamma\mathbf{k}_{2,q}^{2\Delta x})\Big).$$

Likewise, we interpolate the mid-level stages $\mathbf{k}_{1,h}^{2\Delta x}$, $\mathbf{k}_{1,p}^{2\Delta x}$, $\mathbf{k}_{1,q}^{2\Delta x}$, $\mathbf{k}_{2,h}^{2\Delta x}$, $\mathbf{k}_{2,p}^{2\Delta x}$ and $\mathbf{k}_{2,q}^{2\Delta x}$ to a fine grid. These stages are interpolated to a fine grid and used as initial approximations for the fine problem.

$$\bar{\mathbf{k}}_{1,h}^{\Delta x} = I_{h,2\Delta x}^{\Delta x}\mathbf{k}_{1,h}^{2\Delta x}, \quad \bar{\mathbf{k}}_{1,p}^{\Delta x} = I_{p,2\Delta x}^{\Delta x}\mathbf{k}_{1,p}^{2\Delta x}, \quad \bar{\mathbf{k}}_{1,q}^{\Delta x} = I_{q,2\Delta x}^{\Delta x}\mathbf{k}_{1,q}^{2\Delta x}, \tag{8.37}$$

$$\bar{\mathbf{k}}_{2,h}^{\Delta x} = I_{h,2\Delta x}^{\Delta x}\mathbf{k}_{2,h}^{2\Delta x}, \quad \bar{\mathbf{k}}_{2,p}^{\Delta x} = I_{p,2\Delta x}^{\Delta x}\mathbf{k}_{2,p}^{2\Delta x}, \quad \bar{\mathbf{k}}_{2,q}^{\Delta x} = I_{q,2\Delta x}^{\Delta x}\mathbf{k}_{2,q}^{2\Delta x}. \tag{8.38}$$

Finally, the problem on the finest grid, as defined by Equations (8.8)-(8.13), can be solved using the interpolated mid-level solution as the initial approximation.

8.2. Restriction and Interpolation on a Staggered Grid

In the previous Section, numerical solutions were transferred between different grids. In this Subsection, the inter-grid transfer operators for restricting approximations to coarser grids and interpolating solutions to finer grids for the multilevel-method are presented. In Subsection 8.1.1, the procedure for restricting vectors to coarser meshes is outlined. After that, in Subsection 8.1.2. the procedure for transferring solutions to finer grids is presented.

8.2.1. Restricting to Coarser Meshes

The h-variable in the SWEs is discretized in the center of the cell. For cell-centered discretizations, the restriction operator $I_{h,\Delta x}^{2\Delta x}$ is employed, which based on the averages of four neighbouring grid nodes. In Figure 8.2 the locations of h on the coarse and fine grid are shown.

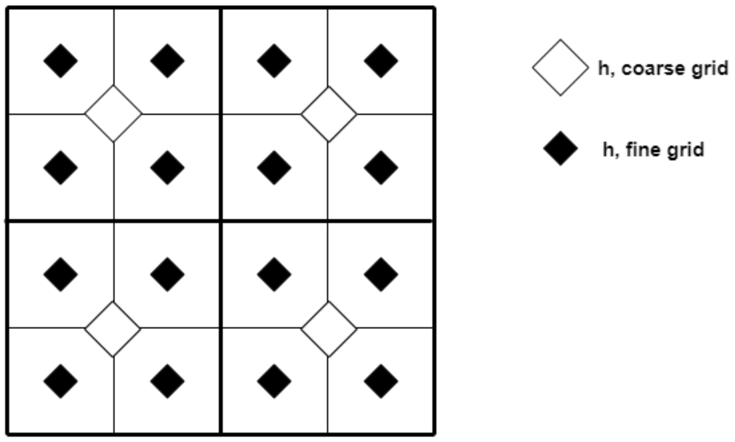


Figure 8.2: The spatial alignment for variable h for both the coarse and fine grids, with spatial steps $2\Delta x$ and Δx, respectively.

For approximating $\mathbf{h}^{2\Delta x}$ on the coarse grid (denoted by white diamonds in Figure 8.2), we use mean values of neighboring points (Trottenberg et al., 2001, p.318):

$$I_{h,\Delta x}^{2\Delta x} h^{\Delta x}(x,y,t) = \frac{1}{4}\begin{bmatrix}1 & & 1\\ & \cdot & \\ 1 & & 1\end{bmatrix} h^{\Delta x} =$$
$$\frac{1}{4}\left(h^{\Delta x}\left(x-\frac{\Delta x}{2}, y-\frac{\Delta y}{2}, t\right) + h^{\Delta x}\left(x-\frac{\Delta x}{2}, y+\frac{\Delta y}{2}, t\right) + \right.$$
$$\left. h^{\Delta x}\left(x+\frac{\Delta x}{2}, y-\frac{\Delta y}{2}, t\right) + h^{\Delta x}\left(x+\frac{\Delta x}{2}, y+\frac{\Delta y}{2}, t\right)\right).$$

The variable p is discretized in the cell-vertex and the locations of the grid nodes corresponding to p on the coarse and fine grids are shown in Figure 8.3.

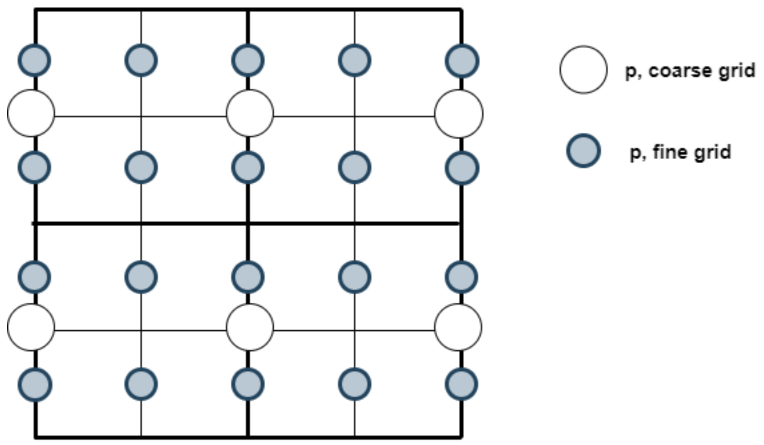


Figure 8.3: The spatial alignment for variable p for both the coarse and fine grids, with spatial steps $2\Delta x$ and Δx, respectively.

Furthermore, for transferring $\mathbf{p}^{\Delta x}$ to coarse grid, we again use mean values of neighboring points (Trottenberg et al., 2001, p.318):

$$I_{p,\Delta x}^{2\Delta x} p^{\Delta x}(x,y,t) = \frac{1}{2}\begin{bmatrix}1\\ \cdot \\ 1\end{bmatrix} p^{\Delta x} =$$
$$\frac{1}{2}\left(p^{\Delta x}\left(x, y-\frac{\Delta y}{2}, t\right) + p^{\Delta x}\left(x, y+\frac{\Delta y}{2}, t\right)\right).$$

The variable q is discretized at the cell vertices and the locations of the grid nodes corresponding to p on a coarse and fine grids are shown in Figure 8.4.

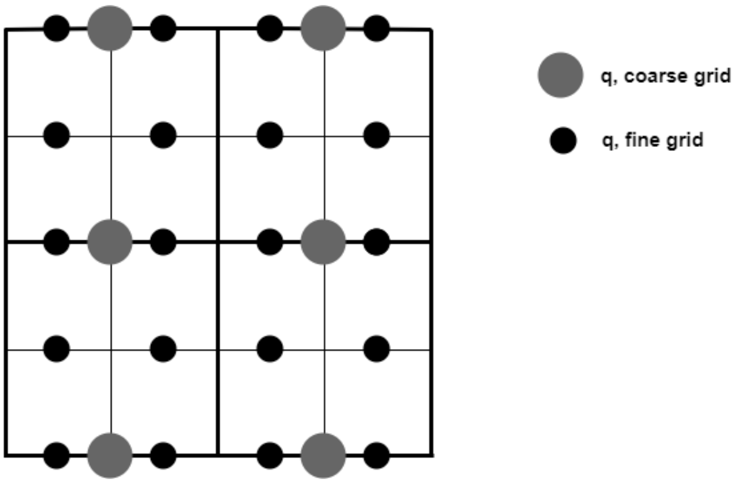


Figure 8.4: The spatial alignment for variable q for both the coarse and fine grids, with spatial steps $2\Delta x$ and Δx, respectively.

Likewise, fine grid solution $\mathbf{q}^{\Delta x}$ is restricted to coarse grid with the following operator (Trottenberg et al., 2001, p.318):

$$I_{q,\Delta x}^{2\Delta x} q^{\Delta x}(x,y,t) = \frac{1}{2}\begin{bmatrix}1 & \cdot & 1\end{bmatrix} q^{\Delta x} =$$
$$\frac{1}{2}\left(q^{\Delta x}\left(x - \frac{\Delta x}{2}, y, t\right) + q^{\Delta x}\left(x + \frac{\Delta x}{2}, y, t\right)\right).$$

8.2.2. Interpolation to Finer Grids

In this Subsection the interpolation schemes from coarse to fine meshes for both the cell-centered and vertex-centered variables are presented.

In Figure 8.5 the cell-centered nodes corresponding to the discretized h-variable are shown with symbols for both fine and coarse mesh points. Moreover, the various fine grid nodes computed using different bilinear interpolation formulas, as given in Equation (8.39), are displayed.

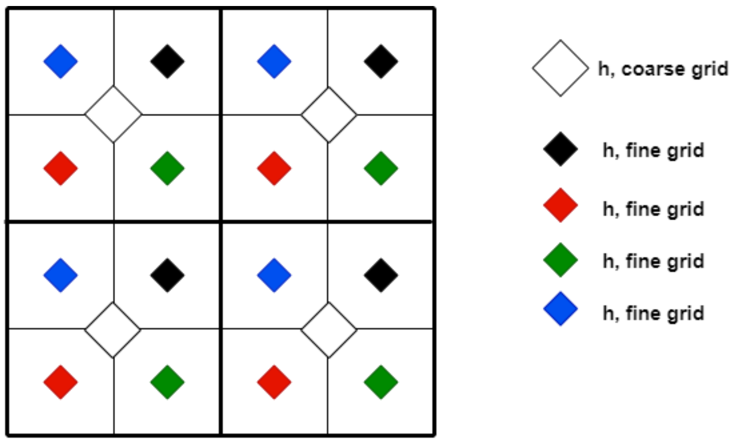


Figure 8.5: A fine grid with colored nodes for demonstrating the bilinear interpolation (8.39) to a coarse mesh with white dicretization-points.

The bilinear interpolation procedure for transferring the cell centered $\mathbf{h}^{2\Delta x}$-vector is given by (Trottenberg et al., 2001, p.70):

$$\mathbf{h}^{\Delta x}(x,y,t) = I^{\Delta x}_{h,2\Delta x}\mathbf{h}^{2\Delta x}(x,y,t) = \begin{cases} \frac{1}{16}\Bigg(9h^{2\Delta x}\left(x-\frac{\Delta x}{2}, y-\frac{\Delta y}{2}, t\right) + 3h^{2\Delta x}\left(x-\frac{\Delta x}{2}, y+\frac{3\Delta y}{2}, t\right) + \\ 3h^{2\Delta x}\left(x+\frac{3\Delta x}{2}, y-\frac{\Delta y}{2}, t\right) + h^{2\Delta x}\left(x+\frac{3\Delta x}{2}, y+\frac{3\Delta y}{2}, t\right)\Bigg) & \text{for black nodes} \\ \frac{1}{16}\Bigg(9h^{2\Delta x}\left(x+\frac{\Delta x}{2}, y-\frac{\Delta y}{2}, t\right) + 3h^{2\Delta x}\left(x+\frac{\Delta x}{2}, y+\frac{3\Delta y}{2}, t\right) + \\ 3h^{2\Delta x}\left(x-\frac{\Delta x}{2}, y-\frac{3\Delta y}{2}, t\right) + h^{2\Delta x}\left(x-\frac{3\Delta x}{2}, y+\frac{3\Delta y}{2}, t\right)\Bigg) & \text{for blue nodes} \\ \frac{1}{16}\Bigg(9h^{2\Delta x}\left(x-\frac{\Delta x}{2}, y+\frac{\Delta y}{2}, t\right) + 3h^{2\Delta x}\left(x-\frac{\Delta x}{2}, y-\frac{3\Delta y}{2}, t\right) + \\ 3h^{2\Delta x}\left(x+\frac{\Delta x}{2}, y+\frac{3\Delta y}{2}, t\right) + h^{2\Delta x}\left(x+\frac{3\Delta x}{2}, y-\frac{3\Delta y}{2}, t\right)\Bigg) & \text{for green nodes} \\ \frac{1}{16}\Bigg(9h^{2\Delta x}\left(x+\frac{\Delta x}{2}, y+\frac{\Delta y}{2}, t\right) + 3h^{2\Delta x}\left(x+\frac{\Delta x}{2}, y-\frac{3\Delta y}{2}, t\right) + \\ 3h^{2\Delta x}\left(x-\frac{\Delta x}{2}, y+\frac{3\Delta y}{2}, t\right) + h^{2\Delta x}\left(x-\frac{3\Delta x}{2}, y-\frac{3\Delta y}{2}, t\right)\Bigg) & \text{for red nodes.} \end{cases} \tag{8.39}$$

In Figure 8.6, the cell-centered nodes corresponding to the discretized variable p are shown with symbols for both fine and coarse mesh points. In addition, the colored fine grid nodes, corresponding to the bilinear interpolation described in Equation (8.40), are displayed.

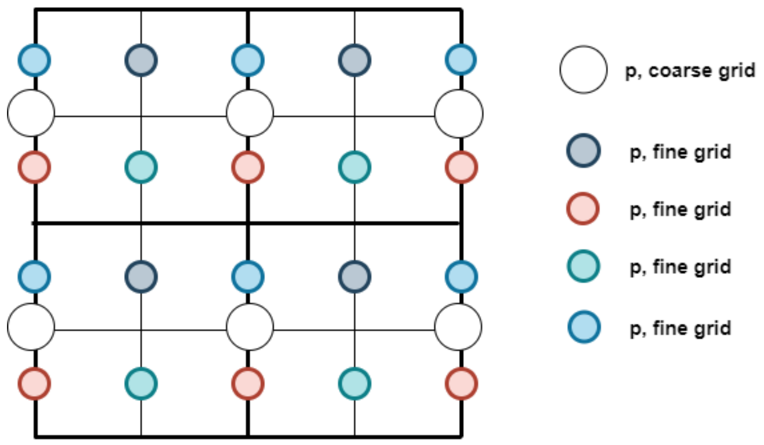


Figure 8.6: Bilinear Interpolation for the vertex-centered discretized p-variable

The bilinear interpolation procedure for transferring the vertex-centered $\mathbf{p}^{2\Delta x}$-vector to a coarse grid is given by (Trottenberg et al., 2001, p.319):

$$\mathbf{p}^{\Delta x}(x,y,t) = I_{p,2\Delta x}^{\Delta x}\mathbf{p}^{2\Delta x}(x,y,t) = \begin{cases} \frac{1}{8}\Big(p^{2\Delta x}\left(x-\Delta x, y-\frac{3\Delta y}{2}, t\right) + p^{2\Delta x}\left(x+\Delta x, y-\frac{3\Delta y}{2}, t\right) + \\ 3p^{2\Delta x}\left(x-\Delta x, y+\frac{\Delta y}{2}, t\right) + 3p^{2\Delta x}\left(x+\Delta x, y-\Delta\frac{y}{2}, t\right)\Big) & \text{for black nodes} \\ \frac{1}{4}\left(p^{2\Delta x}\left(x, y+\frac{3\Delta y}{2}, t\right) + 3p^{2\Delta x}\left(x, y-\frac{\Delta y}{2}, t\right)\right) & \text{for blue nodes} \\ \frac{1}{8}\Big(p^{2\Delta x}\left(x-\Delta x, y+\frac{3\Delta y}{2}, t\right) + p^{2\Delta x}\left(x+\Delta x, y+\frac{3\Delta y}{2}, t\right) \\ 3p^{2\Delta x}\left(x-\Delta x, y-\frac{\Delta y}{2}, t\right) + 3p^{2\Delta x}\left(x+\Delta x, y-\frac{\Delta y}{2}, t\right)\Big) & \text{for green nodes} \\ \frac{1}{4}\left(p^{2\Delta x}\left(x, y-\frac{3\Delta y}{2}, t\right) + 3p^{2\Delta x}\left(x, y+\frac{\Delta y}{2}, t\right)\right) & \text{for red nodes.} \end{cases} \tag{8.40}$$

In Figure 8.7, the cell-centered nodes corresponding to the discretized q-variable are shown with symbols for both fine and coarse mesh points. Moreover, the colored fine grid nodes, corresponding to the bilinear interpolation described in Equation (8.41), are displayed.

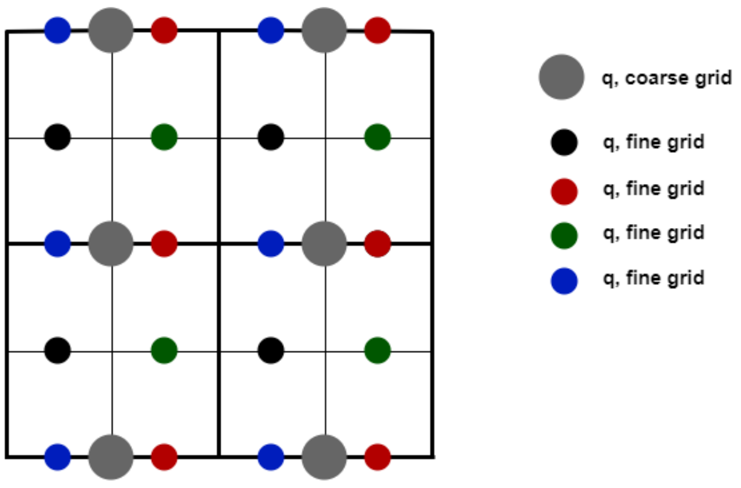


Figure 8.7: Bilinear Interpolation for the vertex-centered discretized q-variable.

The bilinear interpolation formula for interpolating $\mathbf{q}^{2\Delta x}$-vector to a coarse mesh is given by (Trottenberg et al., 2001, p.319):

$$\mathbf{q}^{\Delta x}(x,y,t) = I^{\Delta x}_{q,2\Delta x}\mathbf{q}^{2\Delta x}(x,y,t) = \begin{cases} \frac{1}{8}\Bigg(q^{2\Delta x}\left(x-\frac{3\Delta x}{2}, y+\Delta y, t\right) + q^{2\Delta x}\left(x-\frac{3\Delta x}{2}, y-\Delta y, t\right) + \\ 3q^{2\Delta x}\left(x+\frac{\Delta x}{2}, y+\Delta y, t\right) + 3q^{2\Delta x}\left(x+\frac{\Delta x}{2}, y-\Delta y, t\right)\Bigg) & \text{for black nodes} \\ \frac{1}{4}\Bigg(q^{2\Delta x}\left(x-\frac{3\Delta x}{2}, y, t\right) + 3q^{2\Delta x}\left(x+\frac{\Delta x}{2}, y, t\right)\Bigg) & \text{for blue nodes} \\ \frac{1}{8}\Bigg(q^{2\Delta x}\left(x+\frac{3\Delta x}{2}, y-\Delta y, t\right) + q^{2\Delta x}\left(x+\frac{3\Delta x}{2}, y+\Delta y, t\right) + \\ 3q^{2\Delta x}\left(x-\frac{\Delta x}{2}, y-\Delta y, t\right) + 3q^{2\Delta x}\left(x-\frac{\Delta x}{2}, y+\Delta y, t\right)\Bigg) & \text{for green nodes} \\ \frac{1}{4}\Bigg(q^{2\Delta x}\left(x+\frac{3\Delta x}{2}, y, t\right) + 3q^{2\Delta x}\left(x-\frac{\Delta x}{2}, y, t\right)\Bigg) + & \text{for red nodes.} \end{cases} \tag{8.41}$$

8.3. Experiments with a Multilevel Strategy

In this subsection, various multilevel approaches are explored. These approaches solve the SWEs using different configurations for the levels involved. Additionally, experiments are conducted to examine the impact of varying the coarseness between the finest grid and the coarse grid used for approximations.

Firstly, in Subsection 8.3.1 Multilevel-based solvers for SWEs on a 1080-by-1080 grid are tested with regards to accuracy and time-efficiency. Secondly, in Subsection 8.3.2 the simulation times for multilevel methods are displayed and compared on a finer 4320-by-4320 mesh.

Remark: The value of n_t represents the number of inner iterations. Note, that we still use uniform values of n_t on ever level, i.e. the values of n_t corresponding to different levels are the same for solving the system in every outer implicit iteration. We select n_t on different levels such that on the finest grid, the error/stopping criterion has after performing different numbers of inner iterations on different grid levels has value of most $1e-8$ during the last 20 outer time-steps. After having experimentally determined the uniform values of n_t for different levels, we rerun the experiments without measuring the error and use those experiments to measure computational time.

8.3.1. Error Approximations for a Multilevel Approach on 1080-by-1080 Grid

The experiment setup is as follows:

- The multilevel SDIRK2-scheme will be used for solving the SWEs for a simplified North-Sea model given in Subsection 6.1.1. This test-case is used for testing both the explicit-schemes and the implicit-schemes in Chapters 6 and 7, respectively.
- The SWEs are solved on a 1080-by-1080 grid.
- The time span of the simulation is 10 days.
- The time step of the outer time iterations is $\Delta t = 100$ seconds.
- For computing the errors the benchmark solution is found by solving SWEs on a 3240-by-3240-grid, using a fourth-order RK-method with $\Delta t = 6$ seconds. No interpolation is needed to compute the solutions, we refined with a factor 3 and the grid points of the coarse and fine grid overlap.
- For computing the errors the inf-norm between the benchmark and the multilevel solution is computed.
- A H100 GPU with 96GB (snellius) is used for the experiments.

In Table 8.1 the results of the experiments are found for a tree-level approach. The number of n_t, represents the number of inner iterations per level.

Note that, a two-level version of the interpolation process is defined, where approximations are computed on a 270×270 grid and interpolated to a 1080×1080 grid. The interpolation operators are constructed by combining the two interpolation operators used in the three-level method for transitioning between the 270×270,540×540 and 1080×1080 grids. Specifically, for grid points corresponding to h, the interpolation operator is given by:

$$I^{\Delta x}_{h,4\Delta x} = I^{\Delta x}_{h,2\Delta x} I^{2\Delta x}_{h,4\Delta x},$$

where $\Delta x = \frac{L}{1080}$. A similar approach is applied to interpolate the mesh nodes corresponding to the variables u and v.

Furthermore, by looking at Table 8.1, we can observe that including a multilevel strategy does not lead to an increase in errors. The multilevel approaches still have an error below $1e-3$ and hence are accurate enough.

If we compare the computational times of different methods, one can observe that the two multilevel methods are faster compared to the scheme without multilevel. A two-level approach on a 1080-by-1080 grid, by finding approximations on a 270-by-270 grid is the fastest method reduces computational times by a factor 0.70 compared to the one-level method.

If we introduce a three level scheme, where we also include approximations on 540-by-540-grid, the error is not drastically reduced. However, in the case where a mid-level approximation is included, the simulation times increases. In our specific test-case, only a large scale continuous tidal-model with a Coriolis-component is included and there are no extreme local phenomena or very high local waves. Hence, an approximation on a very coarse grid is very similar to an approximation on a fine grid and a mid-level grid. If we apply this multilevel method to a model with irregular waves on smaller spatial scales, the irregular waves might not be included in the very coarse grid solution. These local waves can be present on the mid-level grid. Therefore, we highly recommend investigating the benefits of including a mid-level grid for discontinuous phenomena.

The experiments in our study are done in double precision. If we repeat the experiment and switch only to single precision, the multilevel methods result in errors of the type NaN. Hence, in single precision our multilevel technique is not stable.

If we select the explicit RK4-method for solving the same problem on a 1080-by-1080-grid with $\Delta t = 18$ seconds (GPU, with double precision), the simulation time is 95 seconds and the error is $0.0004234m$. The percentage of GPU used is at most 96% and the amount of memory used is 900 MiB (1%). The errors for the multilevel scheme in Table (8.1), are similar to the errors for the RK4-scheme. The time complexity of the fastest multilevel scheme in (8.1) and the RK4-scheme are similar. Since in the multilevel scheme solutions for different levels are stored, more memory is needed compared to the single level implicit method and RK4-scheme. Hence, for as solving the test-case on small 1080-by-1080-grid, selecting a multilevel method instead of an RK4-scheme does not improve performance and accuracy on a GPU.

In addition, we also run an SDIRK2_IMEX scheme on a single level with a much smaller time step of $\Delta t = 25$ seconds and. We select an inner time step of 540 seconds and only two inner iterations are needed. The simulation time is 101 seconds. The approximated error is 0.0005824m. The percentage of GPU used is at most 98% and the amount of memory used is 900 MiB (1%). The time complexity of the fastest multilevel scheme in (8.1) and the single-level scheme with a smaller time steps are similar. Hence, for the test-case with a 1080-by-1080 resolution, using a multilevel scheme instead of a method with a smaller time-step, does yield a faster method.

Table 8.1: A three-level approach with on a 1080-by-1080 grid.

	n_x	number of cells	$\Delta\tau$	n_t method 1	n_t method 2	n_t
level 1	1080	1,166,400	58	6	6	12
level 2	540	291,600	160	2	0	0
level 3	270	72,900	2000	2	2	0
Error [m]				0.000440995	0.000444143	0.000622823
time [s]				121	98	141
GPU %				96	99	99
GPU memory [MiB]				1030 (1%)	966 (1%)	900 (1%)

8.3.2. Multilevel Method on a 4320-by-4320 Grid

The experiment setup is as follows:

- The SWEs are solved on a 4320-by-4320-grid.
- The time span of the simulation is 10 days.
- The time step of the outer time iterations is $\Delta t = 100$ or $\Delta t = 100$ seconds.
- A H100 GPU with 96GB (snellius) is used for the experiments.
- Only the simulation time of different Multilevel versions is computed.
- Double precision is selected for the experiments.

Table 8.2: A five-level approach with on a 4320×4320-grid, $\Delta t = 100$s.

	n_x	grid-cells	$\Delta\tau$	n_t I	n_t II	n_t III	n_tIV	n_t V	n_t VI
level 1	4320	18,662,400	12	12	14	16	16	12	44
level 2	2160	4,665,600	25	0	0	0	0	6	0
level 3	1080	1,166,400	58	4	8	0	0	6	0
level 4	540	291,600	160	0	0	4	0	0	0
level 5	270	72,900	2000	2	0	0	2	0	0
time [s]				2017	2361	2600	2599	2311	6996
Memory [MiB]				4964 (5%)	4964 (5%)	4804 (5%)	4772 (5%)	6116 (6%)	5156 (5%)

Table 8.3: A five-level approach with on a 4320×4320 grid, $\Delta t = 25.0$s.

	n_x	grid-cells	$\Delta\tau$	n_t I	n_t II
level 1	4320	18,662,400	13	4	14
level 2	2160	4,665,600	42	0	0
level 3	1080	1,166,400	540	2	0
time [s]				2849	8951
Memory [MiB]				4964 (5%)	5156 (5%)

The time measurements for different multilevel versions are shown in Table 8.2. If the SWEs are solved on only one level, the simulation time is equal to 116 minutes. In Table 8.2 one can notice that the simulation times for the five different multilevel versions are at least three times faster. The improvements in terms of speedup as a result of using multilevel, are bigger for the method in this Subsection compared to the test-case in Subsection 8.3.1. The only difference between the test-case in this Subsection and in the previous Subsection is the grid sizes. In Subsection 8.3.1 a smaller 1080-by-1080 grid was presented. Hence, for fine grids compared to coarse grids, using a multilevel technique is more beneficial.

In Table 8.2, one can observe that a multilevel version with three levels is the fastest. For this method only 12 inner iterations are needed on the finest level, which is lower compared to 3 other variations.

The coarse-grid-approximations are first obtained on a very coarse 270-by-270-grid. This coarse-grid-solution is interpolated to a 1080-by-1080 grid and used as an approximation for the 1080-by-1080-grid-approximation. Furthermore, the 1080-by-1080-grid-solution is computed and used as an approximation for the fine grid solution.

The computational time for the fastest multilevel version (i.e., version I) is 2017 seconds, which is approximately 34 minutes. The fastest explicit time-integration schemes presented in Chapter 6 are RK4 and ORK256. The maximum allowable time step for both RK4 and ORK256 on a 20×20 grid is approximately $\Delta t = 1000$. For a 4320×4320 grid, we approximate the stable time step using the stability condition as follows:

$$\Delta t = \left\lfloor 1000 \times \frac{20}{4320} \right\rfloor = \lfloor 4.62962962962963 \rfloor = 4.$$

Using the explicit RK4 and ORK256 methods, we solve the SWEs over a 10-day period on a 4320×4320 grid in double precision. The simulation times are 5292 seconds and 7320 seconds for the RK4 and ORK256 schemes, respectively. Therefore, in double precision, the SDIRK2 scheme combined with pseudo-time-stepping and the fastest multilevel strategy is more time-efficient than the RK4 and ORK256 methods.

8.3.3. Key Discussion Points

The multilevel experiment, selecting the fastest schemes from Tables 8.1 and 8.2, for $n_x = 4320$ is around 20 times slower compared to the simulation with $n_x = 1080$. However, increasing the grid resolution four times in the x- and y-direction results in a $4^3 = 64$ times increase in computational work for a serial explicit method, based on the stability-condition. For both the 1080- and 4320-resolution the GPU is fully occupied, and hence an explicit method would scale n_x^3-times. However, the multilevel method in our case does not scale with the stability condition.

Additionally, increasing the grid resolution from $n_x = 1080$ to $n_x = 4320$ results in an increase in computational time for the single-level time solver, from 141 seconds to 6996 seconds. This fourfold increase in the number of grid points leads to a runtime that is approximately 50 times longer. Notably, $50 \approx 3.7^3$ and $3.7 \approx 4$, indicating that the computational cost is linked to the stability condition.

For the RK4 method, a similar increase in grid resolution from $n_x = 1080$ to $n_x = 4320$ causes the runtime to increase from 95 seconds to 5292 seconds. Here, the fourfold increase in grid size results in a runtime that is approximately 56 times longer. Similarly, $56 \approx 3.8^3$ and $3.8 \approx 4$, showing again that computational cost is restricted by the stability-condition.

The benefits of applying a mutlilevel technique to the single-level pseudo-time-step method increase as we solve problems for finer grid configurations. For a problem defined on a 4320×4320 grid, computational times are reduced by approximately a factor 3.46, as shown in Table 8.2. On a smaller 1080×1080 grid, the reduction in computational time is equal to a factor 1.44 . Therefore, extending the mutlilevel approach to pseudo-time-stepping for an 8640×8640 grid is expected to yield even greater reductions in computational time.

In Tables 8.1 and 8.2, the fastest multilevel schemes are those that skip one intermediate level. For these multilevel methods, the solution obtained on a coarse grid is interpolated directly onto a grid with spatial step sizes that are four times smaller. In our study, we have selected a tidal problem that contains no local extremities, such as a very high local wave. Hence, numerical solutions on a fine mesh are already somewhat similar to solutions on a coarse mesh. Hence, the solver does not require iterations on some mid-levels. The simplicity of the tidal model furthermore reduces the number of inner iterations required after interpolating a coarse-grid solution onto a finer grid.

In Tables 8.1 and 8.2, we can observe that the fastest multilevel schemes include an initial approximation on a 270-by-270-grid and skip both a level. In Figure 7.7, we observed that for $n_x = 270$, the SDIRK_IMEX is just as fast as for $n_x = 90$ and $n_x = 30$. Hence, there would be no advantage for including coarser grids in the multilevel scheme. Additionally, only two inner iterations are required for an $n_x = 270$ grid problem (with $\Delta t = 100$ and $\Delta \tau = 2000$). In contrast, both the CN and BDF2 schemes require more inner iterations to solve problems on the 270×270 grid.

Extending the Multilevel Approach to Other Implicit Methods In this chapter, a multilevel technique is applied to the SDIRK2_IMEX scheme. In the previous chapter, this scheme was identified as the most effective method without the use of multilevel techniques. In Figure 7.7, it can be seen that for a grid resolution of 810-by-810, the IMEX schemes outperform the other versions. Additionally, the linear portion of the curve corresponding to SDIRK2_IMEX is less steep than the other curves. The IMEX version of the BDF2 method also exhibits a less steep curve. A major downside of this version, however, is illustrated in the left figure of Figure 7.7. As the grid size increases, the error associated with BDF2_IMEX does not decrease as quickly and linearly as with the SDIRK2 schemes. Therefore, SDIRK2_IMEX is more robust than BDF2_IMEX.

In the left figure of Figure 7.7, the line corresponding to CN_IMEX is as steep as the line corresponding to SDIRK2_IMEX. On the 810-by-810 grid, SDIRK2_IMEX is faster compared to CN_IMEX. Both curves exhibit similar steepness, and applying a multilevel approach to SDIRK2_IMEX yields promising results in terms of speed. This suggests that it may be worthwhile to explore whether the CN_IMEX method could be improved in terms of speed with a multilevel approach. However, the CN_IMEX method is not L-stable. As the spatial step sizes decrease, the imaginary eigenvalues near the imaginary axis of the problem become larger, which could lead to instability.

9 Conclusion & Discussion

9.1. Conclusion

The primary objective of this thesis was to compare the performance of various time-integration solvers implemented on a GPU for a simplified North Sea tidal model. The study focused on two categories of time-integration schemes: fully explicit and fully implicit methods. For the GPU-implementations of the fully explicit methods, the GPU implementation was directly based on parallelizing BLAS operations. Conversely, employing an implicit time-integration solver for a nonlinear model requires the implementation of an efficient solver. In our study, a pseudo-time-stepping-based solver was developed and integrated with a multi-level technique to solve nonlinear systems within implicit time-steps.

The fastest method in our study was an implicit SDIRK2-scheme combined with an efficient solver. This solver incorporated a pseudo-transient approach with semi-explicit inner iterations, combined with a multilevel technique. we refer to this fast method as the SIM method. Our research determined that Runge-Kutta 4 was the fastest explicit time-integration scheme, while SDIRK2 was the fastest implicit method. For a problem with 1,166,400 grid cells, the RK4 method and the SIM method demonstrated similar performance. Reducing the spatial step size by a factor of four results in a grid with 18,662,400 cells On the larger grid, the SIM method outperforms the RK4 method.

Increasing the spatial step size by a factor of four led to a 56-fold increase in simulation time for the RK4 method. In contrast, the simulation time for the SIM scheme increases by only a factor of 20. This difference can be attributed to the stability condition for explicit methods. When the spatial step size for RK4 is increased by a factor of 4, the time step must be reduced by the same factor. Consequently, the computational workload increases by approximately factor of $4^3 = 64$ for explicit methods. Note that for the RK4 scheme, $3.8^3 \approx 56$ highlights how the increase in computational time for large grid problems is linked to the stability condition.

Implicit methods, in contrast, do not need to satisfy the CFL condition and have larger stability regions. However, within the iterative nonlinear solver for implicit methods, the number of inner iterations often increases as the grid is refined. This limitiation can be mitigated by the multilevel approach, which reduces the total number of inner iterations by solving problems on coarser grids. As a result, for the SIM method, the computational time increases by only a factor of $20 \approx 2.7^3$. Therefore, for high resolution grids, the SDIRK2 method is more efficient than RK4.

The multilevel approach was applied to a large-scale continuous tidal model. In regions without significant local extrema, the coarse-grid approximation closely matches the fine-grid results. However, in models with high local waves, these phenomena may not be captured on a coarse grid, potentially increasing the required number of iterations on the fine grid. The multilevel method may be less efficient for models with localized phenomena, highlighting the need for further research to investigate these limitations.

Moreover, switching to single precision can lead to instability in some methods. While all experiments were initially conducted using double precision, the SIM method lost its stability when switching to single precision. In contrast, the RK4 method remained stable under single precision. Nevertheless, the SIM method in double precision was still faster than the RK4 scheme in single precision.

The spatially discretized tidal model in the test case exhibited minimal damping and significant oscillations. Spectral analysis revealed that the model frequently had complex eigenvalues with small negative real parts. Similar stability profiles are common in CFD applications with high advection and low diffusion. We recommend conducting further research into the applicability of the SIM solver for such cases.

9.2. Suggestions for Future Work

The following ideas for future research are proposed:

Image Filter Implementation In our study, the pressure terms were spatially discretized using central differences, which were implemented through sparse matrix multiplications. Since the central differences in our study are based on the difference between values at adjacent grid points, an alternative approach could involve using convolution operations instead of sparse matrices.

A convolution operation works as follows (Goodfellow et al., 2016, p.330):

$$D(s,t) = (K * I)(s,t) = \sum_m \sum_n K(m,n) \cdot I(s+m, t+n),$$

where $I(s,t)$ is an image, $K(s,t)$ is an filter kernel and $D(s,t)$ is the output image. The kernel K is typically represented as a matrix.

Substituting a sparse matrix implementation with a convolution could potentially increase computational efficiency on a GPUs. Storing a filter is generally more memory-efficient than storing an equivalent sparse matrix, as the filter requires only the kernel values rather than the indices and values of nonzero elements. The kernel filter would be a 2-by-2-matrix.

Second-Order Implicit Method In our study, we selected a second-order SDIRK (Singly Diagonally Implicit Runge-Kutta) method, which exhibits the highly desirable L-stability property. Performing time integration with the SDIRK2 method involves solving two nonlinear systems, each corresponding to one of its two stage variables.

Alternatively, Sanderse (2013) introduced another implicit Runge-Kutta (RK) method with L-stability. This method combines the Radau IIA and Radau IIB schemes and has a third order accurady. In his study, Sanderse (2013) investigated several implicit RK methods for solving the Navier-Stokes equations. He concluded that the Radau IIA and IIB pairing is an accurate and reliable approach, allowing for large time-steps.

The butcher tableau of the Radau IIA-B-scheme is given by (Sanderse, 2013):

$$\begin{array}{c|cc} \frac{1}{3} & \frac{3}{8} & -\frac{1}{24} \\ 1 & \frac{7}{8} & \frac{1}{8} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}$$

Solving this scheme requires solving a single nonlinear system for two stage variables. However, since this nonlinear system is twice as large as each of the two nonlinear systems in the SDIRK2 method, the computational effort required during the inner pseudo-time iteration remains comparable.

Stability in Single Precision The SIM solver applied to the tidal model defined on a 4320×4320 grid converges to a solution. However, when rerun in single precision, the iterative solver fails to converge. A significant issue is that the SIM method seems highly sensitive to precision.

To ensure stability in double precision, we determined the maximum allowable pseudo-time step for different multilevels by computing the eigenvalues of the Jacobian matrices of to two inner times-iterations, defined on different grids and linearized around zero. With this approach, stability for different inner time-step sizes corresponding to different grids was found by computing the Jacobian of the time-iterations and selecting an inner time step such that the eigenvalue of the Jacobian was smaller than 1. Further investigation into the SIM method is required to understand why it fails to converge in Float32 precision.

To achieve convergence in Float32, within the pseudo-time solver of the SDIRK2 method, we need to delve further into the stability analysis of the SDIRK2 method combined with pseudo-stepping. Furthermore, to continue experimenting with SDIRK2, one could consider replacing pseudo-iterations with Newton's method, combined with a robust solver for linear systems. An example of a robust iterative solver for linear systems is GMRES.

Implementing a Multigrid-method for nonlinear systems In Chapter 8, a multilevel strategy is introduced, where the fine-grid solution is approximated by solving a related problem on a coarser grid. A specific type of multilevel method, known as the multigrid method, is presented in Chapter 3.

In our research, the nonlinear system arising from an implicit time method was solved directly on a coarse grid and then interpolated to a finer grid. However, we could also consider using a nonlinear multigrid method, where similar systems are solved on coarse grids. This approach would provide similar benefits in terms of stability within pseudo-time methods, particularly on the coarse grids.

A Full Approximation scheme (FAS) can be used as a nonlinear multigrid method. Next, we explain the two-grid version of this technique. Firstly, the nonlinear system arising from an implicit time-iteration is given by :

$$N_h(u_h) = f_h = 0,$$

where a grid with h-by-h-grid cells is used (Trottenberg et al., 2001,p.155). The idea behind solving the nonlinear problem with two-grid FAS is somewhat similar to the two-grid linear multigrid-scheme. Likewise, we start with smoothing the errors for the initial solution u_h^m. The u_h^m is restricted to a coarse grid. Furthermore, we compute a defect, $\bar{d}_h^m$ on the fine grid and transfer this defect to a coarse grid. Hence, for the nonlinear multigrid both the defect and the solutions have to be transferred to coarse grids.

Furthermore for mutli-grid methods a defect equation is defined for the coarse grid (Trottenberg et al., 2001, p.155) :

$$N_h(\bar{u}_h^m + v_h^m) - N_h \bar{u}_h^m = \bar{d}_h^m,$$

where $\bar{d}_h^m$ is the defect, corresponding to the h-by-h-grid.

For the coarse grid problem the defect equation is approximated as follows (Trottenberg et al., 2001, p.155):

$$N_H(\bar{u}_H^m + v_H^m) - N_H \bar{u}_H^m = \bar{d}_H^m,$$

where a grid with H-by-H-grid cells is used and N_H is the discrete problem redefined on the coarse grid.

After computing the coarse grid defect $\bar{d}_H^m$, the defect is transferred to a fine grid, and a postsmoother is applied.

Similar to the the linear two-grid cycle presented in Figure 4.2, Figure 9.1 illustrates the corresponding nonlinear two-grid cycle (Trottenberg et al., 2001, p.155).

One reason for experimenting with FAS methods is to determine whether including pre-smoothing reduces the number of inner iterations required by the solver on the coarse grid. Note that the error corrections are interpolated to fine grids. This is relevant because relaxation processes smooth errors, which can then be nicely approximated on coarser grids (Trottenberg et al., 2001,p.156).

$$\begin{tikzcd}
u_h^m \arrow[r, "\text{SMOOTH}^{\nu_1}"'] & \bar{u}_h^m \arrow[r] \arrow[d, "\hat{I}_h^H"] & \bar{d}_h^m = f_h - N_h \bar{u}_h^m \arrow[d, "I_h^H"] & \hat{v}_h^m \arrow[r] & \bar{u}_h^m + \hat{v}_h^m \arrow[r, "\text{SMOOTH}^{\nu_2}"'] & u_h^{m+1} \\
& \bar{u}_H^m & \bar{d}_H^m \arrow[r] & \boxed{N_H(\bar{u}_H^m + \hat{v}_H^m) - N_H \bar{u}_H^m = \bar{d}_H^m} \arrow[u, "I_H^h"'] & &
\end{tikzcd}$$

Figure 9.1: FAS, two-grid version (Trottenberg et al., 2001, p.155)

Determining n_t In our research we determined chose a uniform value of n_t inner iterations for solving every system in the outer iteration, i.e. we selected for every outer-time step the same number of inner iterations. Experimentally we determined n_t such that for every outer time-step the error of the iteration was at most $1e-8$ after performing n_t inner iterations..

We would like to build an implementation, for determining n_t that has different (nonuniform) values for every outer time-step. This includes finding an efficient way to compute a norm on a CuArray on a GPU, for the stopping criterion.

References

Aackermann, P. E., Pedersen, P. J. D., Engsig-Karup, A. P., Clausen, T., & Grooss, J. (2013). Development of a GPU-Accelerated Mike 21 Solver for Water Wave Dynamics. In R. Keller, D. Kramer, & J.-P. Weiss (Eds.), *Facing the Multicore-Challenge III: Aspects of New Paradigms and Technologies in Parallel Computing* (pp. 129–130). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-35893-7_15

Aissa, M., Verstraete, T., & Vuik, C. (2017). Toward a GPU-aware comparison of explicit and implicit CFD simulations on structured meshes [5th European Seminar on Computing ESCO 2016]. *Computers and Mathematics with Applications*, *74*(1), 201–217. https://doi.org/https://doi.org/10.1016/j.camwa.2017.03.003

Andersch, M., Palmer, G., Krashinsky, R., Stam, N., Mehta, V., Brito, G., & Ramaswamy, S. (2022). NVIDIA Hopper Architecture In-Depth [[Accessed 23-05-2024]].

Aureli, F., Prost, F., Vacondio, R., Dazzi, S., & Ferrari, A. (2020). A GPU-Accelerated Shallow-Water Scheme for Surface Runoff Simulations. *Water*, *12*(3). https://doi.org/10.3390/w12030637

Backhaus, J. O. (1983). A semi-implicit scheme for the shallow water equations for application to shelf sea modelling. *Continental Shelf Research*, *2*(4), 243–254. https://doi.org/https://doi.org/10.1016/0278-4343(82)90020-6

Bishnu, S., Strauss, R. R., & Petersen, M. R. (2023). Comparing the Performance of Julia on CPUs versus GPUs and julia-MPI versus Fortran-MPI: A case study with MPAS-ocean (version 7.1). *Geoscientific Model Development*, *16*(19), 5539–5559. https://doi.org/10.5194/gmd-16-5539-2023

Briggs, W. L., Henson, V. E., & McCormick, S. F. (2000). *A Multigrid Tutorial* (2nd ed.). Society for Industrial; Applied Mathematics.

Brodtkorb, A. R., & Holm, H. H. (2021). Coastal ocean forecasting on the GPU using a two-dimensional finite-volume scheme. *Tellus. Series A, Dynamic meteorology and oceanography*, *73*(1), 1–12. https://doi.org/10.1080/16000870.2021.1876341

Burden, R. L., & Faires, J. (2011). *Numerical Analysis* (9th ed.). Brooks/Cole, Cengage Learning.

Buwalda, F. J. L., De Goede, E., Knepflé, M., & Vuik, C. (2023). Comparison of an Explicit and Implicit Time Integration Method on GPUs for Shallow Water Flows on Structured Grids. *Water (Switzerland)*, *15*(6). https://doi.org/10.3390/w15061165

CFD-Wiki. (2012). Courant–friedrichs–lewy condition [Accessed: 2025-01-27]. https://www.cfd-online.com/Wiki/Courant%E2%80%93Friedrichs%E2%80%93Lewy_condition?utm_source=chatgpt.com

Dazzi, S., Vacondio, R., Dal Palù, A., & Mignosa, P. (2018). A local time stepping algorithm for GPU-accelerated 2D shallow water models. *Advances in Water Resources*, *111*, 274–288. https://doi.org/https://doi.org/10.1016/j.advwatres.2017.11.023

De Goede, E. D. (2020). Historical overview of 2D and 3D hydrodynamic modelling of shallow water flows in the netherlands. *Ocean Dynamics*, *70*(4), 521–539. https://doi.org/10.1007/s10236-019-01336-5

Font, B. (2015, August). *High-Order Shock-Capturing Schemes for Micro Shock Tubes* [Doctoral dissertation]. https://doi.org/10.13140/RG.2.1.5170.7601

García-Feal, O., González-Cao, J., Gómez-Gesteira, M., Cea, L., Domínguez, J. M., & Formella, A. (2018). An Accelerated Tool for Flood Modelling Based on Iber. *Water*, *10*(10). https://doi.org/10.3390/w10101459

Garratt, J. R. (1977). Review of Drag Coefficients over Oceans and Continents. *Monthly Weather Review*, *105*(7), 915–929. https://doi.org/10.1175/1520-0493(1977)105<0915:RODCOO>2.0.CO;2

Gerritsen, H., de Goede, E., Platzek, F., van Kester, J., Genseberger, M., & Uittenbogaard, R. (2008). Validation document: Delft3d-flow - a software system for 3d flow simulations [Technical report].

https://www.researchgate.net/publication/301363924_Validation_Document_Delft3D-FLOW_a_software_system_for_3D_flow_simulations

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning* [http://www.deeplearningbook.org]. MIT Press.

Grandclément, P., & Novak, J. (2009). Spectral Methods for Numerical Relativity. *Living Reviews in Relativity*, *12*. https://doi.org/10.12942/lrr-2009-1

Griffiths, G. W., & Schiesser, W. E. (2012). 2 - linear advection equation. In G. W. Griffiths & W. E. Schiesser (Eds.), *Traveling Wave Analysis of Partial Differential Equations* (pp. 7–45). Academic Press. https://doi.org/https://doi.org/10.1016/B978-0-12-384652-5.00002-9

Ha, S., Park, J., & You, D. (2018). A GPU-accelerated semi-implicit fractional-step method for numerical solutions of incompressible Navier–Stokes equations. *Journal of Computational Physics*, *352*, 246–264. https://doi.org/https://doi.org/10.1016/j.jcp.2017.09.055

Hansen, W. (1956). Theorie zur Errechnung des Wasserstandes und der Strömungen in Randmeeren nebst Anwendungen. *Tellus A: Dynamic Meteorology and Oceanography*, *8*(2), 287–300.

Harding, S. (2022). *What is TDP? a Basic Definition*. https://www.tomshardware.com/reviews/tdp-thermal-design-power-definition,5764.html

Harten, A. (1983). High resolution schemes for hyperbolic conservation laws. *Journal of Computational Physics*, *49*(3), 357–393. https://doi.org/https://doi.org/10.1016/0021-9991(83)90136-5

Intel. (2024a). Intel® core™ i9 processor 14900ks [[Accessed 14-05-2024]].

Intel. (2024b, January). APP Metrics for Intel® Microprocessors [Accessed: 2025-01-27]. https://www.intel.com/content/www/us/en/content-details/840270/app-metrics-for-intel-microprocessors-intel-xeon-processor.html

Iserles, A. (2008). A first course in the numerical analysis of differential equations. Cambridge University Press.

Kennedy, C. A., & Carpenter, M. H. (2016). *Diagonally implicit runge-kutta methods for ordinary differential equations: A review* (tech. rep. No. NASA/TM–2016–219173) (Technical Memorandum). NASA Langley Research Center. https://ntrs.nasa.gov/citations/20160005923

Kirk, D. B., & Hwu, W.-m. W. (2017). Chapter 1 - introduction. In D. B. Kirk & W.-m. W. Hwu (Eds.), *Programming Massively Parallel Processors* (Third Edition, pp. 1–18). Morgan Kaufmann. https://doi.org/https://doi.org/10.1016/B978-0-12-811986-0.00001-7

Klingbeil, K., Lemarié, F., Debreu, L., & Burchard, H. (2018). The numerics of hydrostatic structured-grid coastal ocean models: State of the art and future perspectives. *Ocean Modelling*, *125*, 80–105. https://doi.org/https://doi.org/10.1016/j.ocemod.2018.01.007

Knabner, P., & Angermann, L. (2021). The finite volume method. In *Numerical Methods for Elliptic and Parabolic Partial Differential Equations: With contributions by Andreas Rupp* (pp. 487–555). Springer International Publishing. https://doi.org/10.1007/978-3-030-79385-2_8

Krashinsky, R., Giroux, O., Jones, S., Stam, N., & Ramaswamy, S. (2020). NVIDIA Ampere Architecture In-Depth [[Accessed 23-05-2024]].

LeVeque, R. J. (2002). *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press.

Lin, Y., & Grover, V. (2018). Using CUDA Warp-Level Primitives [[accessed 20-05-2024]].

Luo, P., Rodrigo, C., Gaspar, F., & Oosterlee, C. (2018). Monolithic multigrid method for the coupled Stokes flow and deformable porous medium system. *Journal of Computational Physics*, *353*, 148–168. https://doi.org/https://doi.org/10.1016/j.jcp.2017.09.062

Maitre, O. (2013). Understanding NVIDIA GPGPU Hardware. In S. Tsutsui & P. Collet (Eds.), *Massively Parallel Evolutionary Computation on GPGPUs* (pp. 15–34). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-37959-8_2

Miller, R. N. (2007). *Numerical modeling of ocean circulation*. Cambridge University Press.

Nikolopoulos, S., Kalogeris, I., Stavroulakis, G., & Papadopoulos, V. (2024). AI-enhanced iterative solvers for accelerating the solution of large-scale parametrized systems. *International Journal for Numerical Methods in Engineering*, *125*(2), e7372. https://doi.org/https://doi.org/10.1002/nme.7372

Nvidia. (2024). CUDA C++ Programming Guide [[accessed 16-05-2024]].

Onodera, N., Idomura, Y., Hasegawa, Y., Yamashita, S., Shimokawabe, T., & Aoki, T. (2021). GPU Acceleration of Multigrid Preconditioned Conjugate Gradient Solver on Block-Structured Cartesian Grid. *The International Conference on High Performance Computing in Asia-Pacific Region*, 120–128. https://doi.org/10.1145/3432261.3432273

Pugh, D., & Woodworth, P. (2014). *Sea-Level Science: Understanding Tides, Surges, Tsunamis and Mean Sea-Level Changes*. Cambridge University Press.

Rackauchas, C. (2020). *Ordinary Differential Equations, Applications and Discretizations*. Retrieved April 24, 2024, from https://book.sciml.ai/notes/07-Ordinary_Differential_Equations-Applications_and_Discretizations/

Rackauckas, C. (2020a, 14 October). The different flavors of parallelism - mit parallel computing and scientific machine learning (sciml) [[Accessed 14-05-2024]].

Rackauckas, C. (2020b, 14 October). Solving stiff ordinary differential equations. Solving Stiff Ordinary Differential Equations - MIT Parallel Computing and Scientific Machine Learning (SciML) [[Accessed 14-05-2024]].

Rackauckas, C., & Nie, Q. (2017). Differentialequations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia [Exported from https://app.dimensions.ai on 2019/05/05]. *The Journal of Open Research Software*, *5*(1). https://doi.org/10.5334/jors.151

Rak, A., Mewis, P., & Guthe, S. (2024). Accelerating flash flood simulations: An efficient GPU implementation for a slim shallow water solver. *Environmental Modelling and Software*, *177*, 106030. https://doi.org/https://doi.org/10.1016/j.envsoft.2024.106030

Räss, L., Werder, M., Omlin, S., & Utkin, I. (2023). Lecture 5 — pde-on-gpu.vaw.ethz.ch [[Accessed 25-04-2024]].

Reid, R. (1990). Tides and Storm Surges. In *Handbook of Coastal and Ocean Engineering* (pp. 533–590). Gulf Publishing Company.

Safarzadeh Maleki, F., & Khan, A. A. (2015). Effect of channel shape on selection of time marching scheme in the discontinuous Galerkin method for 1-D open channel flow. *Journal of Hydrodynamics*, *27*(3), 413–426. https://doi.org/10.1016/S1001-6058(15)60499-1

Sanderse, B. (2013). Energy-conserving Runge–Kutta methods for the incompressible Navier–Stokes equations. *Journal of Computational Physics*, *233*, 100–131. https://doi.org/https://doi.org/10.1016/j.jcp.2012.07.039

Smith, L. S., & Liang, Q. (2013). Towards a generalised GPU/CPU shallow-flow modelling tool. *Computers and Fluids*, *88*, 334–343. https://doi.org/https://doi.org/10.1016/j.compfluid.2013.09.018

Stoer, J., & Bulirsch, R. (1993). *Introduction to Numerical Analysis* (nd ed., Ser. Texts in applied mathematics, 12). Springer-Verlag.

Süli, E., & Mayers, D. F. (2003). *An Introduction to Numerical Analysis*. Cambridge University Press.

Technical-City. (n.d.). Xeon Gold 6248R vs Platinum 8268 [[accessed 25-01-2025]]. https://technical.city/en/cpu/Xeon-Platinum-8268-vs-Xeon-Gold-6248R

Techpowerup. (2018). NVIDIA Tesla V100 PCIe 32 GB [[accessed 21-05-2024]].

Techpowerup. (2021). NVIDIA A100 PCIe 80 GB [[accessed 18-06-2024]].

Techpowerup. (2023). NVIDIA H100 PCIe 80 GB [[accessed 18-06-2024]].

Trottenberg, U., Oosterlee, C. W., & Schüller, A. (2001). *Multigrid*. Acedemic Press.

van Kan, J., Segal, A., Vermolen, F., & H.Kraaijevanger. (2019). *Numerical Methods for Partial Differential Equations*. VSSD.

Vreugdenhil, C. B. (1994). *Numerical Methods for Shallow-Water Flow*. Springer Netherlands. https://doi.org/10.1007/978-94-015-8354-1_1

Vuik, C., & Lahaye, D. (2019). *Scientific Computing*. DIAM.

Vuik, C., & Lemmens, C. (2018). Programming on the GPU with CUDA theoretical background [[Accessed 14-05-2024]].

Vuik, C., Vermolen, F. J., van Gijzen, M. B., & Vuik, M. J. (2018). *Numerical Methods for Ordinary Differential Equations* (2nd ed.). VSSD.

Wang, J., Li, S., Hou, J., Liu, Y., Hu, W., Shi, X., & Yao, J. (2024). Study on Multi-Time Scale Hydrodynamic Model Based on Local Time Stepping Scheme and GPUs and its Application in Urban Inundation. *Water Resources Management*, *38*, 1615–1637. https://doi.org/https://doi.org/10.1007/s11269-024-03742-x

Wang, L., & Yu, M. (2020). Comparison of row, esdirk, and bdf2 for unsteady flows with the high-order flux reconstruction formulation. *Journal of Scientific Computing*, *83*(39). https://doi.org/10.1007/s10915-020-01222-z

Westerink, J. J., Luettich, R. A., Feyen, J. C., Atkinson, J. H., Dawson, C., Roberts, H. J., Powell, M. D., Dunion, J. P., Kubatko, E. J., & Pourtaheri, H. (2008). A Basin- to Channel-Scale Unstructured

Grid Hurricane Storm Surge Model Applied to Southern Louisiana. *Monthly Weather Review*, *136*(3), 833–864. https://doi.org/10.1175/2007MWR1946.1

Wikipedia. (2024, 16 January). Parallel Thread Execution [[Accessed 16-05-2024]].

Zandbergen, A., van Noorden, T., & Heinlein, A. (2023, October). *Improving Pseudo-Time Stepping Convergence for CFD Simulations with Neural Networks* (tech. rep.) (Submitted).

Zhang, Y., & Jia, Y. (2013). Parallelized CCHE2D flow model with CUDA Fortran on Graphics Processing Units. *Computers and Fluids*, *84*, 359–368. https://doi.org/https://doi.org/10.1016/j.compfluid.2013.06.021

Zolfaghari, H., & Obrist, D. (2021). A high-throughput hybrid task and data parallel Poisson solver for large-scale simulations of incompressible turbulent flows on distributed GPUs. *Journal of Computational Physics*, *437*, 110329. https://doi.org/https://doi.org/10.1016/j.jcp.2021.110329

A

Code for Implementation

A.1. Code for Explicit Methods in Chapter 6

We use functions from DifferentialEquations.jl to solve ODEs, as following:

```
wave_ode=ODEProblem(fs,x0,time_span)
sol  = solve(wave_ode, RK4(), dt=d4, adaptive=false,
save_everystep=false,save_start=false,save_end=true);
```

Here, only the numerical solution in my simulation is saved.

A.1.1. Input data

First we define the input of our functions and push arrays to the GPU

```
#parameters
Lx=500000.0  # length
Ly=500000.0  # length

Δx=Lx/(nx)    # spatial step for x
Δy=Ly/(ny)    # spatial step for y

seconds_per_day=24*3600
ω=2*π/seconds_per_day
λ=π/3

# spatial derivatives on Arakawa C-grid
Dx_h_ =∂x_h_matrix(nx,Δx)
Dy_h_ =∂y_h_matrix(ny,Δy)
Dx_p_ =∂x_u_matrix(nx,Δx)
Dy_q_ =∂y_v_matrix(ny,Δy)

# initial condition

h0_=zeros(nx,ny)
p0_=zeros(nx+1,ny)
q0_=zeros(nx,ny+1)
r_ = zeros(nx+1,ny+1)

#Averages of u, v in gridpoints corresponding to v,u, respectively
Ax_p_= avg_xdir_u(nx)
Ay_p_= avg_ydir_u(ny)
Ay_q_=avg_ydir_v(nx)
```

```
Ax_q_= avg_xdir_v(ny)

#Mask array
M_arr_q_ = mask_v(nx,ny+1)
M_arr_p_ = mask_u(nx+1,ny)

#fC=Float32(2*ω*sin(λ)) # 0.00012

#fC= 2* 7.2921159 * sin(π/3) * 1e-5  #Coriolis accerleration

#Cf= 0.003f0 #bottom friction constant

if run_on_cuda
    if run_on_precision_single
        Tfloat = Float32
        Tdense = CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}
        Tsparse_x = CUDA.CUSPARSE.CuSparseMatrixCSC{Float32, Int32}
        Tsparse_y = CUDA.CUSPARSE.CuSparseMatrixCSR{Float32, Int32}

        D=60.0f0      # dept
        g=9.81f0      # gravity

        f_tide = 0.0f0
        m2 = Float32((2*pi/(12.4*3600.0)))
        α = Float32(0.1e-4)
        Cf= 0.003f0
        fC=Float32(2*ω*sin(λ))
        Lx = Float32(Lx)
        Ly = Float32(Ly)
        Δx = Float32(Δx)
        Δy = Float32(Δy)
    else
        Tfloat = Float64
        Tdense = CuArray{Float64, 2, CUDA.Mem.DeviceBuffer}
        Tsparse_x = CUDA.CUSPARSE.CuSparseMatrixCSC{Float64, Int32}
        Tsparse_y = CUDA.CUSPARSE.CuSparseMatrixCSR{Float64, Int32}

        D=60.0      # dept
        g=9.81      # gravity

        f_tide = 0.0
        m2 = Float64((2*pi/(12.4*3600.0)))
        α = Float64(0.1e-4)
        Cf= 0.003
        fC=Float64(2*ω*sin(λ))
        #time_span=(0.0, Float64(10.0*24*3600))
    end
    Dx_h=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Dx_h_)
    Dy_h=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Dy_h_)
    Dx_p=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Dx_p_)
    Dy_q=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Dy_q_)
    Ax_p=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Ax_p_)
    Ay_p=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Ay_p_)
    Ay_q=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Ay_q_)
    Ax_q=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Ax_q_)
    M_arr_p= CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(M_arr_p_)
    M_arr_q= CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(M_arr_q_)
    #Dx_h=CuArray{T}(Dx_h) # push as full matrix ... not needed
    h0=CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(h0_) # push to GPU
    p0=CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(p0_)
    q0=CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(q0_)
    r = CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(r_)
else
    if run_on_precision_single
        Tfloat = Float32
        f_tide= 0.0f0
        Tdense = Matrix{Float32}
        Tsparse_x = SparseMatrixCSC{Float32, Int32}
        Tsparse_y = SparseMatrixCSC{Float32, Int32}
        m2 = Float32((2*pi/(12.4*3600.0)))
        α = Float32(0.1e-4)

        Cf= 0.003f0
        #time_span=(0.0f0, Float32(10.0*24*3600))
        fC=Float32(2*ω*sin(λ))

        D=60.0f0      # dept
        g=9.81f0      # gravity

        Lx = Float32(Lx)
        Ly = Float32(Ly)
        Δx = Float32(Δx)
        Δy = Float32(Δy)

    else
        Tfloat = Float64
        f_tide= Float64(0.0)
        Tdense = Matrix{Float64}
        Tsparse_x = SparseMatrixCSC{Float64, Int32}
        Tsparse_y = SparseMatrixCSC{Float64, Int32}
        m2 = Float64((2*pi/(12.4*3600.0)))
        α = Float64(0.1e-4)
        Cf= 0.003
        #time_span=(0.0, Float64(10.0*24*3600))
        fC=Float64(2*ω*sin(λ))

        D=60.0      # dept
        g=9.81      # gravity

    end
    Dx_h=Tfloat.(Dx_h_)
    Dy_h=Tfloat.(Dy_h_)
    Dx_p=Tfloat.(Dx_p_)
    Dy_q=Tfloat.(Dy_q_)
    Ax_p=Tfloat.(Ax_p_)
    Ay_p=Tfloat.(Ay_p_)
    Ay_q=Tfloat.(Ay_q_)
    Ax_q=Tfloat.(Ax_q_)
    M_arr_p= Tfloat.(M_arr_p_)
    M_arr_q= Tfloat.(M_arr_q_)
    h0=Tfloat.(h0_)
    p0=Tfloat.(p0_)
    q0=Tfloat.(q0_)
    r=Tfloat.(r_)
end



x0=ComponentVector(h=h0,p=p0,q=q0) # initial condition

# allocate memory for temporal storage of the spatial derivatives
∂h∂x=similar(p0)
∂h∂y=similar(q0)
∂p∂x=similar(h0)
∂q∂y=similar(h0)

Ap1 = similar(r)
Aq1 = similar(h0)
Ap2 = similar(q0)
Aq2 = similar(p0)

Ah_p= similar(p0)
Ah_q= similar(q0)
H = similar(h0)

dh = similar(h0)
dp = similar(p0)
dq = similar(q0)
dx_dt = similar(x0)
fs=Wave2DSt{Tfloat, Tdense, Tsparse_x, Tsparse_y}(g,D,fC,Cf,f_tide,m2,α, Lx,Ly,Δx,Δy,
∂h∂x,∂h∂y,∂p∂x,∂q∂y,Dx_h,Dy_h,Dx_p,Dy_q,Ax_p,Ay_p,Ay_q,Ax_q,
Ap1,Aq1,Ap2,Aq2,H, Ah_p,Ah_q, M_arr_p, M_arr_q,dh,dp,dq)
```

A.1.2. Function fs

The fs a functor and is defined as follows:

```
"""
    Wave2DSt
    contains data, including cachearrays
"""
struct Wave2DSt{Tfloat,Tdense,Tsparse_x,Tsparse_y}
    g::Tfloat # gravity
    D::Tfloat # depth
    fC::Tfloat #Coriolisestimate
    Cf::Tfloat #bottom friction const
    f_tide::Tfloat #newtonian tide potential
    m2::Tfloat # tidal period
    α::Tfloat #tidal amplitude
    Lx::Tfloat # length
    Ly::Tfloat # length
    Δx::Tfloat # spatial step
    Δy::Tfloat # spatial step
    ∂h∂x::Tdense # cache arrays
    ∂h∂y::Tdense
    ∂p∂x::Tdense
    ∂q∂y::Tdense
    Dx_h::Tsparse_x  # x derivative matrix for h
    Dy_h::Tsparse_y
    Dx_p::Tsparse_x
    Dy_q::Tsparse_y
    Ax_p::Tsparse_x  # Averages for computing Coriolis
    Ay_p::Tsparse_y
    Ay_q::Tsparse_y
    Ax_q::Tsparse_x
    Ap1::Tdense
    Aq1::Tdense
    Ap2::Tdense
    Aq2::Tdense
    H::Tdense
    Ah_p::Tdense
    Ah_q::Tdense
    M_arr_p::Tdense
    M_arr_q::Tdense
    dh::Tdense
    dp::Tdense
    dq::Tdense
end

"""
    Wave2D
    Compute time derivative of the state
    function (f::Wave2D)(dx_dt,x,p,t)
    This is a functor, so it's based on the Wave2D struct, and can access its data
"""
function (f::Wave2DSt)(dx_dt,x,p,t)
    g=f.g
    D=f.D
    fC=f.fC
    Cf=f.Cf
    f_tide = f.f_tide
    m2 = f.m2
    α = f.α
    # temporary variables
    ∂h∂x = f.∂h∂x
    ∂h∂y = f.∂h∂y
    ∂p∂x = f.∂p∂x
    ∂q∂y = f.∂q∂y
    Dx_h = f.Dx_h
    Dy_h = f.Dy_h
    Dx_p = f.Dx_p
    Dy_q = f.Dy_q
    Ap1 = f.Ap1
    Aq1= f.Aq1
    Ap2 = f.Ap2
    Aq2= f.Aq2
    Ax_p = f.Ax_p
    Ay_p = f.Ay_p
    Ay_q = f.Ay_q
    Ax_q = f.Ax_q
    M_arr_p=f.M_arr_p
    M_arr_q=f.M_arr_q
    H=f.H
    Ah_p = f.Ah_p
    Ah_q = f.Ah_q

    dh = f.dh
    dp = f.dp
    dq = f.dq

    @. H = D + x.h  #H=D+h

    #Compute averages for bottom friction-term of water-depths
    mul!(Ah_p,Ax_q,H)
    mul!(Ah_q,H,Ay_p)


    #Compute averages in y-direction
    mul!(Ap1,x.p, Ay_p)
    mul!(Aq1,x.q, Ay_q)
    #compute averages in x-direction
    mul!(Ap2, Ax_p, Ap1)
    mul!(Aq2, Ax_q, Aq1)

    # compute spatial derivatives
    mul!(∂h∂x,Dx_h,x.h)
    mul!(∂h∂y,x.h,Dy_h)
    mul!(∂p∂x,Dx_p,x.p)
    mul!(∂q∂y,x.q,Dy_q)


    #compute averages
    # compute time derivatives
    ## Continuity
    @. dh = -∂p∂x - ∂q∂y
    ## Momentum x and y
    # M2 tide has one cycle every 12.4 hours
    f_tide=α*sin(m2*t)


    @. dp = M_arr_p * (  -g*Ah_p*∂h∂x + fC*Aq2
    - ((Cf/(Ah_p*Ah_p ))*x.p*sqrt(x.p*x.p+Aq2*Aq2) + Ah_p * f_tide  )  )
    @. dq = M_arr_q * (  -g*Ah_q*∂h∂y
    - fC*Ap2 - ((Cf/(Ah_q*Ah_q))*x.q*sqrt(Ap2*Ap2+x.q*x.q) ) )

    dx_dt.h = dh
    dx_dt.p = dp
    dx_dt.q = dq
    nothing
end
```

A.1.3. Functions for Finite Differences, Mask-arrays and Averages

```
function ∂x_u_matrix(nx,Δx)
    row_indices=Int64[]
    col_indices=Int64[]
    values=Float64[]
    @inbounds for i in 1:(nx)
        push!(row_indices,i)
        push!(col_indices,i)
        push!(values,-1.0/Δx)
        push!(row_indices,i)
        push!(col_indices,i+1)
        push!(values,1.0/Δx)
    end
    return sparse(row_indices,col_indices,values,nx,nx+1)
end

function ∂x_h_matrix(nx,Δx)
    row_indices=Int64[]
    col_indices=Int64[]
    values=Float64[]
    @inbounds for i in 1:(nx)
        push!(col_indices,i)
        push!(row_indices,i)
        push!(values,1.0/Δx)
        push!(col_indices,i)
        push!(row_indices,i+1)
        push!(values,-1.0/Δx)
    end
    return sparse(row_indices,col_indices,values,nx+1,nx)
end


function ∂y_v_matrix(ny,Δy)
    row_indices=Int64[]
    col_indices=Int64[]
    values=Float64[]
    @inbounds for i in 1:(ny)
        push!(col_indices,i)
        push!(row_indices,i)
        push!(values,-1.0/Δy)
        push!(col_indices,i)
        push!(row_indices,i+1)
        push!(values,1.0/Δy)
    end
    return sparse(row_indices,col_indices,values,ny+1,ny)
end

function ∂y_h_matrix(ny,Δy)
    row_indices=Int64[]
    col_indices=Int64[]
    values=Float64[]
    @inbounds for i in 1:(ny)
        push!(col_indices,i)
        push!(row_indices,i)
        push!(values,1.0/Δy)
        push!(col_indices,i+1)
        push!(row_indices,i)
        push!(values,-1.0/Δy)
    end
    return sparse(row_indices,col_indices,values,ny,ny+1)
end

function avg_xdir_u(nx)
```

```
    row_indices=Int64[]
    col_indices=Int64[]
    values=Float64[]
    @inbounds for i in 1:(nx)
        push!(row_indices,i)
        push!(col_indices,i)
        push!(values,1.0/2.0)
        push!(row_indices,i)
        push!(col_indices,i+1)
        push!(values,1.0/2.0)
    end
    return sparse(row_indices,col_indices,values,nx,nx+1)
end

function avg_ydir_u(ny)
    row_indices=Int64[]
    col_indices=Int64[]
    values=Float64[]
    @inbounds for i in 1:(ny)
        push!(col_indices,i)
        push!(row_indices,i)
        push!(values,1.0/2.0)
        push!(col_indices,i+1)
        push!(row_indices,i)
        push!(values,1.0/2.0)
    end
    return sparse(row_indices,col_indices,values,ny,ny+1)
end

function avg_ydir_v(ny)
    row_indices=Int64[]
    col_indices=Int64[]
    values=Float64[]
    @inbounds for i in 1:(ny)
        push!(col_indices,i)
        push!(row_indices,i)
        push!(values,1.0/2.0)
        push!(col_indices,i)
        push!(row_indices,i+1)
        push!(values,1.0/2.0)
    end
    return sparse(row_indices,col_indices,values,ny+1,ny)
end

function avg_xdir_v(nx)
    row_indices=Int64[]
    col_indices=Int64[]
    values=Float64[]
    @inbounds for i in 1:(nx)
        push!(col_indices,i)
        push!(row_indices,i)
        push!(values,1.0/2.0)
        push!(col_indices,i)
        push!(row_indices,i+1)
        push!(values,1.0/2.0)
    end
    return sparse(row_indices,col_indices,values,nx+1,nx)
end

function mask_v(nx,ny)
    #set elements on this array to zero corresponding to
    #upper and lower boundaries for v=0
    B=ones(nx,ny)
    @inbounds for i in 1:nx
        B[i]=0
        B[i+(ny-1)*nx]=0
    end
    return B
end

function mask_u(nx,ny)
    #set elements on this array to zero corresponding to
    # left and right boundaries for u=0
    B=ones(nx,ny)
    @inbounds for j in 1:ny
        B[(j-1)*nx+1]=0
        B[(j)*nx]=0
    end
    return B
end
```

A.2. Code for Pseudo-Time-Stepping Methods in Chapter 7

A.2.1. Defining the SWEs

First of all we define tree functors instead of one, these functors correspond to the right hand sides of $\frac{\partial h}{\partial t} = G_h$, $\frac{\partial p}{\partial t} = G_p$ and $\frac{\partial q}{\partial t} = G_q$.

```
struct Wave2Dh{Tfloat,Tdense,Tsparse_x,Tsparse_y}
    Lx::Tfloat # length
    Ly::Tfloat # length
    Δx::Tfloat # spatial step
    Δy::Tfloat # spatial step
    ∂p∂x::Tdense
    ∂q∂y::Tdense
    Dx_p::Tsparse_x
    Dy_q::Tsparse_y
end

struct Wave2Dp{Tfloat,Tdense,Tsparse_x,Tsparse_y}
    g::Tfloat # gravity
    D::Tfloat # depth
    fC::Tfloat #Coriolisestimate
    Cf::Tfloat #bottom friction const
    f_tide::Tfloat #newtonian tide potential
    m2::Tfloat # tidal period
    α::Tfloat #tidal amplitude
    Lx::Tfloat # length
    Ly::Tfloat # length
    Δx::Tfloat # spatial step
    Δy::Tfloat # spatial step
    ∂h∂x::Tdense # cache arrays
    Dx_h::Tsparse_x  # x derivative matrix for h
    Ay_q::Tsparse_y
    Ax_q::Tsparse_x
    Aq1::Tdense
    Aq2::Tdense
    H::Tdense
    Ah_p::Tdense
    M_arr_p::Tdense
end

struct Wave2Dq{Tfloat,Tdense,Tsparse_x,Tsparse_y}
    g::Tfloat # gravity
    D::Tfloat # depth
    fC::Tfloat #Coriolisestimate
    Cf::Tfloat #bottom friction const
    Lx::Tfloat # length
    Ly::Tfloat # length
    Δx::Tfloat # spatial step
    Δy::Tfloat # spatial step
    ∂h∂y::Tdense
    Dy_h::Tsparse_y
    Ay_p::Tsparse_y
    Ax_p::Tsparse_x
    Ap1::Tdense
    Ap2::Tdense
    H::Tdense
    Ah_q::Tdense
    M_arr_q::Tdense
end

function (Gh::Wave2Dh)(Evalh, h,p,q,t)
    # temporary variables
    ∂p∂x = Gh.∂p∂x
    ∂q∂y = Gh.∂q∂y
    Dx_p = Gh.Dx_p
    Dy_q = Gh.Dy_q


    mul!(∂p∂x,Dx_p,p) #Show three arguments before this line, to debug.
    mul!(∂q∂y,q,Dy_q)

    ## Continuity
    @. Evalh = -∂p∂x - ∂q∂y

    nothing
end

function (Gp::Wave2Dp)(Evalp, h,p,q,t)
    g=Gp.g
    D=Gp.D
    fC=Gp.fC
    Cf=Gp.Cf
    f_tide = Gp.f_tide
    m2 = Gp.m2
    α = Gp.α
    # temporary variables
    ∂h∂x = Gp.∂h∂x
    Dx_h = Gp.Dx_h
```

```julia
    Aq1= Gp.Aq1
    Aq2= Gp.Aq2
    Ay_q = Gp.Ay_q
    Ax_q = Gp.Ax_q
    M_arr_p=Gp.M_arr_p
    H=Gp.H
    Ah_p = Gp.Ah_p



    @. H = D + h  #H=D+h

    #Compute averages for bottom friction-term of water-depths
    mul!(Ah_p,Ax_q,H)

    #Compute averages in y-direction
    mul!(Aq1,q, Ay_q)
    #compute averages in x-direction
    mul!(Aq2, Ax_q, Aq1)

    # compute spatial derivatives
    mul!(∂h∂x,Dx_h,h)

    ## Momentum x and y
    # M2 tide has one cycle every 12.4 hours
    f_tide=α*sin(m2*t)


    @. Evalp = M_arr_p * (  -g*Ah_p*∂h∂x + fC*Aq2
    -((Cf/(Ah_p*Ah_p ))*p*sqrt(p*p+Aq2*Aq2) + Ah_p * f_tide  )  )

    nothing
end

function (Gq::Wave2Dq)(Evalq, h,p,q,t)
    g=Gq.g
    D=Gq.D
    fC=Gq.fC
    Cf=Gq.Cf
    # temporary variables
    ∂h∂y = Gq.∂h∂y
    Dy_h = Gq.Dy_h
    Ap1 = Gq.Ap1
    Ap2 = Gq.Ap2


    Ax_p = Gq.Ax_p
    Ay_p = Gq.Ay_p
    M_arr_q=Gq.M_arr_q
    H=Gq.H

    Ah_q = Gq.Ah_q

    @. H = D + h  #H=D+h

    #Compute averages for bottom friction-term of water-depths
    mul!(Ah_q,H,Ay_p)

    #Compute averages in y-direction
    mul!(Ap1,p, Ay_p)
    #compute averages in x-direction
    mul!(Ap2, Ax_p, Ap1)


    # compute spatial derivatives

    mul!(∂h∂y,h,Dy_h)

    @. Evalq = M_arr_q * (  -g*Ah_q*∂h∂y - fC*Ap2 -
    ((Cf/(Ah_q*Ah_q))*q*sqrt(Ap2*Ap2+q*q) ) )


    nothing
end


function ODEProblemII(nx,ny, run_on_cuda, run_on_precision_single)

    Lx=500000.0  # length
    Ly=500000.0  # length
    Δx=Lx/(nx)    # spatial step for x
    Δy=Ly/(ny)    # spatial step for y

    seconds_per_day= 24*3600
    ω=2*π/seconds_per_day
    λ=π/3


    #initialization
    #fC = 0
    #Cf = 0
    # spatial grid
    (x_h,y_h)=h_grid(nx,ny,Δx,Δy)
    (x_u,y_u)=p_grid(nx,ny,Δx,Δy)
    (x_v,y_v)=q_grid(nx,ny,Δx,Δy)
    #x_u_=x_ut*ones(1,size(x_ut)[1]-1)
    # spatial derivatives on Arakawa C-grid
    Dx_h_ =∂x_h_matrix(nx,Δx)
    Dy_h_ =∂y_h_matrix(ny,Δy)
    Dx_p_ =∂x_u_matrix(nx,Δx)
    Dy_q_ =∂y_v_matrix(ny,Δy)

    # initial condition
    #h0_= gaussian_bump(x_h,y_h,1.0,0.5*Lx,0.5*Ly,0.1*Lx,0.1*Ly)
    h0_=zeros(nx,ny)
    p0_=zeros(nx+1,ny)
    q0_=zeros(nx,ny+1)
    r_ = zeros(nx+1,ny+1)

    #Averages of u, v in gridpoints corresponding to v,u, respectively
    Ax_p_= avg_xdir_u(nx)
    Ay_p_= avg_ydir_u(ny)
```

```
Ay_q_=avg_ydir_v(nx)
Ax_q_= avg_xdir_v(ny)

#Mask array
M_arr_q_ = mask_v(nx,ny+1)
M_arr_p_ = mask_u(nx+1,ny)

if run_on_cuda
    if run_on_precision_single
        Tfloat = Float32
        Tdense = CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}
        Tsparse_x = CUDA.CUSPARSE.CuSparseMatrixCSC{Float32, Int32}
        Tsparse_y = CUDA.CUSPARSE.CuSparseMatrixCSR{Float32, Int32}
        f_tide = 0.0f0
        m2 = Float32((2*pi/(12.4*3600.0)))
        α = Float32(0.1e-4)

        D=60.0f0     # dept
        g=9.81f0     # gravity

        Cf= 0.003f0
        fC=Float32(2*ω*sin(λ))
        time_span=(0.0f0, Float32(3*24*3600))

        Lx = Float32(Lx)
        Ly = Float32(Ly)
        Δx = Float32(Δx)
        Δy = Float32(Δy)
    else
        Tfloat = Float64
        Tdense = CuArray{Float64, 2, CUDA.Mem.DeviceBuffer}
        Tsparse_x = CUDA.CUSPARSE.CuSparseMatrixCSC{Float64, Int32}
        Tsparse_y = CUDA.CUSPARSE.CuSparseMatrixCSR{Float64, Int32}
        f_tide = 0.0
        m2 = Float64((2*pi/(12.4*3600.0)))
        α = Float64(0.1e-4)

        D=60.0     # dept
        g=9.81     # gravity

        Cf= 0.003
        fC=Float64(2*ω*sin(λ))
        time_span=(0.0, Float64(3*24*3600))

    end
    Dx_h=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Dx_h_) # push to GPU as sparse mat
    Dy_h=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Dy_h_)
    Dx_p=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Dx_p_)
    Dy_q=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Dy_q_)
    Ax_p=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Ax_p_)
    Ay_p=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Ay_p_)
    Ay_q=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Ay_q_)
    Ax_q=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Ax_q_)
    M_arr_p= CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(M_arr_p_)
    M_arr_q= CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(M_arr_q_)
    #Dx_h=CuArray{T}(Dx_h) # push as full matrix ... not needed
```

```
        h0=CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(h0_) # push to GPU
        p0=CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(p0_)
        q0=CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(q0_)
        r = CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(r_)
    else
        if run_on_precision_single
            Tfloat = Float32
            f_tide= 0.0f0
            Tdense = Matrix{Float32}
            Tsparse_x = SparseMatrixCSC{Float32, Int32}
            Tsparse_y = SparseMatrixCSC{Float32, Int32}
            m2 = Float32((2*pi/(12.4*3600.0)))
            α = Float32(0.1e-4)

            Cf= 0.003f0
            time_span=(0.0f0, Float32(3*24*3600))
            fC=Float32(2*ω*sin(λ))
            D=60.0f0     # dept
            g=9.81f0     # gravity
            Lx = Float32(Lx)
            Ly = Float32(Ly)
            Δx = Float32(Δx)
            Δy = Float32(Δy)

        else
            Tfloat = Float64
            f_tide= Float64(0.0)
            Tdense = Matrix{Float64}
            Tsparse_x = SparseMatrixCSC{Float64, Int32}
            Tsparse_y = SparseMatrixCSC{Float64, Int32}
            m2 = Float64((2*pi/(12.4*3600.0)))
            α = Float64(0.1e-4)
            Cf= 0.003
            time_span=(0.0, Float64(3.0*24*3600))
            fC=Float64(2*ω*sin(λ))
            D=60.0     # dept
            g=9.81     # gravity

        end
        Dx_h=Tfloat.(Dx_h_)
        Dy_h=Tfloat.(Dy_h_)
        Dx_p=Tfloat.(Dx_p_)
        Dy_q=Tfloat.(Dy_q_)
        Ax_p=Tfloat.(Ax_p_)
        Ay_p=Tfloat.(Ay_p_)
        Ay_q=Tfloat.(Ay_q_)
        Ax_q=Tfloat.(Ax_q_)
        M_arr_p= Tfloat.(M_arr_p_)
        M_arr_q= Tfloat.(M_arr_q_)
        h0=Tfloat.(h0_)
        p0=Tfloat.(p0_)
        q0=Tfloat.(q0_)
        r=Tfloat.(r_)
    end
    ∂h∂x=zero(p0)
    ∂h∂y=zero(q0)
    ∂p∂x=zero(h0)
    ∂q∂y=zero(h0)

    Ap1 = zero(r)
    Aq1 = zero(h0)
    Ap2 = zero(q0)
    Aq2 = zero(p0)

    Ah_p= zero(p0)
    Ah_q= zero(q0)
    H = zero(h0)


    Gh=Wave2Dh{Tfloat, Tdense, Tsparse_x,Tsparse_y}(Lx,Ly,Δx,Δy,∂p∂x,∂q∂y,Dx_p,Dy_q)
    Gp=Wave2Dp{Tfloat, Tdense, Tsparse_x, Tsparse_y}(g,D,fC,Cf,f_tide,m2,α, Lx,Ly,
    Δx,Δy,∂h∂x,Dx_h,Ay_q,Ax_q,Aq1,Aq2,H, Ah_p, M_arr_p)
    Gq=Wave2Dq{Tfloat, Tdense, Tsparse_x, Tsparse_y}(g,D,fC,Cf, Lx,Ly,
    Δx,Δy,∂h∂y,Dy_h,Ay_p,Ax_p,Ap1,Ap2,H,Ah_q, M_arr_q)
    return Gh,Gp,Gq,h0,p0,q0
end
```

A.2.2. Code For Applying Pseudo-Time-Stepping

The following code applies a pseudo-time-stepping method for solving implicit methods. We can choose a label such as BE_IM() or SDIRK2_IMEX() as input and within the function the corresponding pseudo-time-method will be applied.

```
struct sol_seudo{Tfloat,Tdense}
    t::Tfloat
    h::Tdense
    p::Tdense
    q::Tdense
end


function solve_pseudo(method::PseudoMethod, Δt, Δτ,time_span,n_t, h0,p0,q0,
Gh,Gp,Gq;printerr =false, save_everystep=true, GPU=false)
    #number of iterations

    Tfloat= typeof(Δt)
    Tdense = typeof(h0)
    TInt= typeof(n_t)
    Nt = TInt(round(time_span[2]/Δt))
    t=time_span[1]
    @show Tdense

    if save_everystep
        sol = Array{sol_seudo,1}(undef,(Nt+1))
    else
        sol =  Array{sol_seudo,1}(undef,1)
    end
   time_t=0
    I = initialize_integrator(method, h0,p0,q0,Tdense)
    CH = initialize_cache(method,Δt,Δτ, n_t, h0,p0,q0, Tfloat,TInt,Tdense)

    for i in 1:Nt
        if save_everystep
           sol[i] = sol_seudo{Tfloat, Tdense}(t, copy(I.h),copy(I.p),copy(I.q))
        end

        if GPU
            time =  CUDA.@elapsed perform_step!(method, CH, I,Gh,Gp,Gq,t, printerr )
        else
            time =  @elapsed perform_step!(method, CH, I,Gh,Gp,Gq,t, printerr )
        end

        t+=Δt
        time_t += time
    end

    sol[end] = sol_seudo{Tfloat, Tdense}(t, copy(I.h),copy(I.p),copy(I.q))

    I=nothing
    CH=nothing

    return sol, time_t
end
```

For solving the SWEs with the solve_pseudo function, this is the function call:

```
        sol,time = solve_pseudo(SDIRK2_IMEX(), Δt, Δτ_SDIRK,time_span,nt_SDIRK, h0,p0,q
```

where in sol1 the solution is stored and time is the computational time.

A.2.3. Code For Performing One SDIRK2_IMEX Step

If we choose the SDIRK2_IMEX option, this is the function for performing an implicit step:

```
function perform_step!(method::SDIRK2_IMEX,  CH::CacheSDIRK2,
I::integratorI, Gh,Gp,Gq,t, printerr)

    n_t = CH.n_t
    Δτ = CH.Δτ
    Δt = CH.Δt
    dd= CH.dd #dd = 1 / (1+ (Δτ/Δt)), for implicit integration of xi


    h = I.h
    p = I.p
    q = I.q

    hprev  = I.hprev
    pprev  = I.pprev
    qprev  = I.qprev

    eval_gh = CH.eval_gh
    eval_gp = CH.eval_gp
    eval_gq = CH.eval_gq

    x1_h = CH.k1_h
    x1_p = CH.k1_p
    x1_q = CH.k1_q
    x2_h = CH.k2_h
    x2_p = CH.k2_p
    x2_q = CH.k2_q


    y1 = CH.y1
    y2 = CH.y2

    t+=y1*Δt
    n_it=div(n_t,2)

    @. p = pprev + Δt * y1 * x1_p
    @. q = qprev + Δt * y1 * x1_q
    @. h = hprev + Δt * y1 * x1_h

    for i in 1:n_it
        Gp(eval_gp, h, p, q, t)
        @. x1_p = (x1_p + (Δτ/Δt) * (eval_gp )) * dd
        @. p = pprev + Δt * y1 * x1_p

        Gq(eval_gq, h,p, q, t)
        @. x1_q = (x1_q + (Δτ/Δt) * (eval_gq )) * dd
        @. q = qprev + Δt * y1 * x1_q


        Gh(eval_gh,h,p, q, t)
        @. x1_h = (x1_h + (Δτ/Δt) * ( eval_gh)) * dd
        @. h = hprev + Δt * y1 * x1_h


        Gq(eval_gq, h,p, q, t)
        @. x1_q = (x1_q + (Δτ/Δt) * (eval_gq )) * dd
        @. q = qprev + Δt * y1 * x1_q


        Gp(eval_gp, h, p, q, t)
        @. x1_p = (x1_p + (Δτ/Δt) * (eval_gp )) * dd
        @. p = pprev + Δt * y1 * x1_p


        Gh(eval_gh,h,p, q, t)
        @. x1_h = (x1_h + (Δτ/Δt) * ( eval_gh)) * dd
        @. h = hprev + Δt * y1 * x1_h

    end

    t+=y2*Δt

    @. h = hprev + Δt * (y2 * x1_h + y1 * x2_h)
    @. p = pprev + Δt * (y2 * x1_p + y1 * x2_p)
    @. q = qprev + Δt * (y2 * x1_q + y1 * x2_q)

    for i in 1:n_t

        Gp(eval_gp, h, p, q, t)
        @. x2_p = (x2_p + (Δτ/Δt) * eval_gp ) * dd
        @. p = pprev + Δt * (γ2 * x1_p + γ1 * x2_p)

        Gq(eval_gq, h, p, q, t)
        @. x2_q = (x2_q + (Δτ/Δt) * eval_gq ) * dd
        @. q = qprev + Δt * (γ2 * x1_q + γ1 * x2_q)

        Gh(eval_gh, h, p, q, t)
        @. x2_h = (x2_h + (Δτ/Δt) * eval_gh ) * dd
        @. h = hprev + Δt * (γ2 * x1_h + γ1 * x2_h)

        Gq(eval_gq, h, p, q, t)
        @. x2_q = (x2_q + (Δτ/Δt) * eval_gq ) * dd
        @. q = qprev + Δt * (γ2 * x1_q + γ1 * x2_q)

        Gp(eval_gp, h, p, q, t)
        @. x2_p = (x2_p + (Δτ/Δt) * eval_gp ) * dd
        @. p = pprev + Δt * (γ2 * x1_p + γ1 * x2_p)

        Gh(eval_gh, h, p, q, t)
        @. x2_h = (x2_h + (Δτ/Δt) * eval_gh ) * dd
        @. h = hprev + Δt * (γ2 * x1_h + γ1 * x2_h)


    end
    if (printerr && (t<(3600*24*10)-Δt*21))  # we print error in last inner iteration t
        Gh(eval_gh, h, p, q, t)
        # for testing printerr=true, for timing printerr = false
        normI= norm(eval_gh - x2_h, Inf)
        @show normI
    end


    @. I.hprev = h
    @. I.pprev = p
    @. I.qprev = q
    nothing
end
```

A.3. Code for Multi-Level Method

A.3.1. Defining Problems with Varying Grid Sizes and Discretizations

For a tree level multi-grid, we use the functors Gh, Gp and Gq from the previous section. We explain how we can define a SWE problem for a tree-level grid from method 2, from Table 8.1. For the fine and the coarse grid, we rediscretize the ode.

```
Lx=500000.0  # length
Ly=500000.0  # length

nx2 = div(nx1,2)
ny2 = div(nx1,2)

nx3 = div(nx2,2)
ny3 = div(nx2,2)

Δx1=Lx/(nx1)    # spatial step for x finest grid
Δy1=Ly/(ny1)    # spatial step for y finest grid
```

```
Δx3=Lx/(nx3)    # spatial step for x mid level grid
Δy3=Ly/(ny3)    # spatial step for y  mid level grid


#initialization
Dx_h1_ =∂x_h_matrix(nx1,Δx1)
Dy_h1_ =∂y_h_matrix(ny1,Δy1)
Dx_p1_ =∂x_u_matrix(nx1,Δx1)
Dy_q1_ =∂y_v_matrix(ny1,Δy1)

Dx_h3_ =∂x_h_matrix(nx3,Δx3)
Dy_h3_ =∂y_h_matrix(ny3,Δy3)
Dx_p3_ =∂x_u_matrix(nx3,Δx3)
Dy_q3_ =∂y_v_matrix(ny3,Δy3)

# initial condition

h01_=zeros(nx1,ny1)
p01_=zeros(nx1+1,ny1)
q01_=zeros(nx1,ny1+1)
r1_ = zeros(nx1+1,ny1+1)

h03_=zeros(nx3,ny3)
p03_=zeros(nx3+1,ny3)
q03_=zeros(nx3,ny3+1)
r3_ = zeros(nx3+1,ny3+1)

#Averages of u, v in gridpoints corresponding to v,u, respectively
Ax_p1_= avg_xdir_u(nx1)
Ay_p1_= avg_ydir_u(ny1)
Ay_q1_= avg_ydir_v(nx1)
Ax_q1_= avg_xdir_v(ny1)

Ax_p3_= avg_xdir_u(nx3)
Ay_p3_= avg_ydir_u(ny3)
Ay_q3_= avg_ydir_v(nx3)
Ax_q3_= avg_xdir_v(ny3)

#Mask array
M_arr_q1_ = mask_v(nx1,ny1+1)
M_arr_p1_ = mask_u(nx1+1,ny1)


M_arr_q3_ = mask_v(nx3,ny3+1)
M_arr_p3_ = mask_u(nx3+1,ny3)

if run_on_cuda
    if run_on_precision_single
        Tfloat = Float32
        Tdense = CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}
        Tsparse_x = CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat, Int32}
        Tsparse_y = CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat, Int32}

    else
        Tfloat = Float64
        Tdense = CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}
        Tsparse_x = CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat, Int32}
        Tsparse_y = CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat, Int32}

    end
    Dx_h1=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Dx_h1_) # push to GPU as sparse matri
    Dy_h1=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Dy_h1_)
    Dx_p1=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Dx_p1_)
    Dy_q1=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Dy_q1_)
    Dx_h3=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Dx_h3_) # push to GPU as sparse matri
    Dy_h3=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Dy_h3_)
    Dx_p3=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Dx_p3_)
    Dy_q3=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Dy_q3_)


    Ax_p1=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Ax_p1_)
    Ay_p1=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Ay_p1_)
    Ay_q1=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Ay_q1_)
    Ax_q1=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Ax_q1_)
    Ax_p3=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Ax_p3_)
    Ay_p3=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Ay_p3_)
    Ay_q3=CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Ay_q3_)
    Ax_q3=CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Ax_q3_)


    M_arr_p1= CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(M_arr_p1_)
    M_arr_q1= CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(M_arr_q1_)


    M_arr_p3= CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(M_arr_p3_)
    M_arr_q3= CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(M_arr_q3_)

    #Dx_h=CuArray{T}(Dx_h) # push as full matrix ... not needed
    h01=CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(h01_) # push to GPU
    p01=CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(p01_)
    q01=CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(q01_)
    r1 = CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(r1_)


    h03=CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(h03_) # push to GPU
    p03=CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(p03_)
    q03=CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(q03_)
    r3 = CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}(r3_)
else
    if run_on_precision_single
        Tfloat = Float32
        Tdense = Matrix{Tfloat}
        Tsparse_x = SparseMatrixCSC{Tfloat, Int32}
        Tsparse_y = SparseMatrixCSC{Tfloat, Int32}

    else
        Tfloat = Float64
        Tdense = Matrix{Tfloat}
        Tsparse_x = SparseMatrixCSC{Tfloat, Int32}
        Tsparse_y = SparseMatrixCSC{Tfloat, Int32}
    end
    Dx_h1=Tfloat.(Dx_h1_)
    Dy_h1=Tfloat.(Dy_h1_)
    Dx_p1=Tfloat.(Dx_p1_)
    Dy_q1=Tfloat.(Dy_q1_)

    Dx_h3=Tfloat.(Dx_h3_)
    Dy_h3=Tfloat.(Dy_h3_)
    Dx_p3=Tfloat.(Dx_p3_)
    Dy_q3=Tfloat.(Dy_q3_)


    Ax_p1=Tfloat.(Ax_p1_)
    Ay_p1=Tfloat.(Ay_p1_)
    Ay_q1=Tfloat.(Ay_q1_)
    Ax_q1=Tfloat.(Ax_q1_)

    Ax_p3=Tfloat.(Ax_p3_)
    Ay_p3=Tfloat.(Ay_p3_)
    Ay_q3=Tfloat.(Ay_q3_)
    Ax_q3=Tfloat.(Ax_q3_)


    M_arr_p1= Tfloat.(M_arr_p1_)
    M_arr_q1= Tfloat.(M_arr_q1_)

    M_arr_p3= Tfloat.(M_arr_p3_)
    M_arr_q3= Tfloat.(M_arr_q3_)


    h01=Tfloat.(h01_)
    p01=Tfloat.(p01_)
    q01=Tfloat.(q01_)
    r1=Tfloat.(r1_)

    h03=Tfloat.(h03_)
    p03=Tfloat.(p03_)
    q03=Tfloat.(q03_)
    r3=Tfloat.(r3_)
end

seconds_per_day= 24*3600
ω=2*π/seconds_per_day
λ=π/3

f_tide = Tfloat(0.0)
m2 = Tfloat((2*pi/(12.4*3600.0)))
α = Tfloat(0.1e-4)

D=Tfloat(60.0)      # dept
g=Tfloat(9.81)      # gravity

Cf= Tfloat(0.003)
fC=Tfloat(2*ω*sin(λ))

Lx = Tfloat(Lx)
Ly = Tfloat(Ly)
```

```
Δx1 = Tfloat(Δx1)
Δy1 = Tfloat(Δy1)
Δx3 = Tfloat(Δx3)
Δy3 = Tfloat(Δy3)


∂h∂x1=zero(p01)
∂h∂y1=zero(q01)
∂p∂x1=zero(h01)
∂q∂y1=zero(h01)


∂h∂x3=zero(p03)
∂h∂y3=zero(q03)
∂p∂x3=zero(h03)
∂q∂y3=zero(h03)


Ap11 = zero(r1)
Aq11 = zero(h01)
Ap21 = zero(q01)
Aq21 = zero(p01)


Ap13 = zero(r3)
Aq13 = zero(h03)
Ap23 = zero(q03)
Aq23 = zero(p03)

Ah_p1 = zero(p01)
Ah_q1 = zero(q01)
H1 = zero(h01)


Ah_p3 = zero(p03)
Ah_q3 = zero(q03)
H3 = zero(h03)


Gh1=Wave2Dh{Tfloat, Tdense, Tsparse_x, Tsparse_y}
(Lx,Ly,Δx1,Δy1,∂p∂x1,∂q∂y1,Dx_p1,Dy_q1)
Gp1=Wave2Dp{Tfloat, Tdense, Tsparse_x, Tsparse_y}
(g,D,fC,Cf,f_tide,m2,α, Lx,Ly,Δx1,Δy1,∂h∂x1,Dx_h1,Ay_q1,Ax_q1,Aq11,Aq21,H1, Ah_p1, M_ar
Gq1=Wave2Dq{Tfloat, Tdense, Tsparse_x, Tsparse_y}

(g,D,fC,Cf, Lx,Ly,Δx1,Δy1,∂h∂y1,Dy_h1,Ay_p1,Ax_p1,Ap11,Ap21,H1,Ah_q1, M_arr_q1)


Gh3=Wave2Dh{Tfloat, Tdense, Tsparse_x, Tsparse_y}
(Lx,Ly,Δx3,Δy3,∂p∂x3,∂q∂y3,Dx_p3,Dy_q3)
Gp3=Wave2Dp{Tfloat, Tdense, Tsparse_x, Tsparse_y}
(g,D,fC,Cf,f_tide,m2,α,
Lx,Ly,Δx3,Δy3,∂h∂x3,Dx_h3,Ay_q3,Ax_q3,Aq13,Aq23,H3, Ah_p3, M_arr_p3)
Gq3=Wave2Dq{Tfloat, Tdense, Tsparse_x, Tsparse_y}
(g,D,fC,Cf,
Lx,Ly,Δx3,Δy3,∂h∂y3,Dy_h3,Ay_p3,Ax_p3,Ap13,Ap23,H3,Ah_q3, M_arr_q3)
```

A.3.2. The Time-Integration Function

If we select a tree-level approach, and choose to skip a level, with the option $S31()$ we can call the desired function:

```
sol, time = solve_pseudo_Multigrid(S31(), Δt, Δτ1, Δτ3,
time_span,n_t1,n_t3, nx1,ny1,Gh1,Gp1,Gq1,Gh3,Gp3,Gq3;
printerr= false, save_everystep=false, GPU=true);
```

where time is the computational time and sol is the solution.

As a result this function is called:

```
function solve_pseudo_Multigrid(method::S31, Δt, Δτ1, Δτ3,
time_span,n_t1,n_t3, nx,ny,Gh1,Gp1,Gq1, Gh3,Gp3,Gq3;
printerr= true, save_everystep=true, GPU=false)
    #number of iterations

    Tfloat= typeof(Δt)
    TInt= typeof(n_t1)
    Nt = TInt(round(time_span[2]/Δt))
    t=time_span[1]
    if GPU
        Tdense =  CuArray{Tfloat, 2, CUDA.Mem.DeviceBuffer}
    else
        Tdense =  Matrix{Tfloat}
    end


    if save_everystep
        sol = Array{sol_seudo,1}(undef,(Nt+1))
    else
        sol =  Array{sol_seudo,1}(undef,1)
    end
    time_t=0.0
    I = initialize_integrator(method,Δt, Δτ1, Δτ3, nx,ny,n_t1,n_t3,
    Tdense, Tfloat, TInt, GPU)

    for i in 1:Nt
        if save_everystep
           sol[i] = sol_seudo{Tfloat, Tdense}(t, copy(I.h1),copy(I.p1),copy(I.q1))
        end

        if GPU
            time =  CUDA.@elapsed perform_step_multi!(
            method,I,Gh1,Gp1,Gq1,Gh3,Gp3,Gq3,t,printerr )
        else
            time =  @elapsed perform_step_multi!(method,I,Gh1,Gp1,Gq1,
            Gh3,Gp3,Gq3,t, printerr )
        end

        t+=Δt
        time_t += time
    end

    sol[end] = sol_seudo{Tfloat, Tdense}(t, copy(I.h1),copy(I.p1),copy(I.q1))
    I=nothing

    return sol, time_t
end
```

A.3.3. Initialize Integrator

On the Integrator, the interpolation arrays are defined. Since we skip a mid-level the interpolation operator is computed by multiplying two interpolation operators. Within the function initialize integrator interpolation operators are computed and pushed to a GPU or CPU.

```
Ix_h1_ = Interpolate_hx(nx2)*Interpolate_hx(nx3)
Iy_h1_ = Interpolate_hy(ny3)*Interpolate_hy(ny2)


if GPU
    Tsparse_x = CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat, Int32}
    Tsparse_y = CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat, Int32}

    Ix_h1 = CUDA.CUSPARSE.CuSparseMatrixCSC{Tfloat}(Ix_h1_)
    Iy_h1 = CUDA.CUSPARSE.CuSparseMatrixCSR{Tfloat}(Iy_h1_)

else
    Tsparse_x = SparseMatrixCSC{Tfloat, Int32}
    Tsparse_y = SparseMatrixCSC{Tfloat, Int32}

    Ix_h1 = Tfloat.(Ix_h1_)
    Iy_h1 = Tfloat.(Iy_h1_)

end
```

The functions for interpolation are defined in the following subsection.

A.3.4. Interpolation Operators

For the grid-nodes corresponding to h on the staggered grid, we have defined the following operators for Interpolation.

```
function Interpolate_hy(ny)
    row_indices=Int64[]
    col_indices=Int64[]
    values=Float64[]
    push!(col_indices,1)
    push!(row_indices,1)
    push!(values,1)

    push!(col_indices,2*ny)
    push!(row_indices,ny)
    push!(values,1)

    @inbounds for i in 1:(ny-1)

        push!(col_indices,2*i)
        push!(row_indices,i)
        push!(values,3/4)

        push!(col_indices,2*i+1)
        push!(row_indices,i)
        push!(values,1/4)

        push!(col_indices,2*i)
        push!(row_indices,i+1)
        push!(values,1/4)

        push!(col_indices,2*i+1)
        push!(row_indices,i+1)
        push!(values,3/4)
    end
 return sparse(row_indices,col_indices,values, ny,ny*2)
end

function Interpolate_hx(nx)
    row_indices=Int64[]
    col_indices=Int64[]
    values=Float64[]
    push!(col_indices,1)
    push!(row_indices,1)
    push!(values,1)

    push!(row_indices,2*nx)
    push!(col_indices,nx)
    push!(values,1)

    @inbounds for i in 1:(nx-1)

        push!(row_indices,2*i)
        push!(col_indices,i)
        push!(values,3/4)

        push!(row_indices,2*i+1)
        push!(col_indices,i)
        push!(values,1/4)

        push!(row_indices,2*i)
        push!(col_indices,i+1)
        push!(values,1/4)

        push!(row_indices,2*i+1)
        push!(col_indices,i+1)
        push!(values,3/4)
    end
 return sparse(row_indices,col_indices,values, nx*2,nx)
end
```

A.3.5. Performing a Time-Step Using a Three-Level-Grid

. with S31() as input we select a tree-level method, that skips the mid-level stage. With the function function perform_step_multi!, an outer SDIRK2 time iteration is computed.

```
    function perform_step_multi!(method::S31,I::IntegratorS31,
    Gh1,Gp1,Gq1,Gh3,Gp3,Gq3,t,print_err)

    n_t1 = I.n_t1
    n_t3 = I.n_t3
    Δτ1 = I.Δτ1
    Δτ3 = I.Δτ3

    Δt = I.Δt
    dd1 = I.dd1
    dd3 = I.dd3


    h1 = I.h1
    p1 = I.p1
    q1 = I.q1
    h3 = I.h3
    p3 = I.p3
    q3 = I.q3

    hprev1 = I.hprev1
    pprev1 = I.pprev1
    qprev1 = I.qprev1
    hprev3 = I.hprev3
    pprev3 = I.pprev3
    qprev3 = I.qprev3


    eval_gh1 = I.eval_gh1
    eval_gp1 = I.eval_gp1
    eval_gq1 = I.eval_gq1
    eval_gh3 = I.eval_gh3
    eval_gp3 = I.eval_gp3
    eval_gq3 = I.eval_gq3

    x1_h1 = I.k1_h1
    x1_p1 = I.k1_p1
    x1_q1 = I.k1_q1
    x2_h1 = I.k2_h1
    x2_p1 = I.k2_p1
    x2_q1 = I.k2_q1


    x1_h3 = I.k1_h3
    x1_p3 = I.k1_p3
    x1_q3 = I.k1_q3
    x2_h3 = I.k2_h3
    x2_p3 = I.k2_p3
    x2_q3 = I.k2_q3

    Cx_h1 = I.Cx_h1
    Cy_h1 = I.Cy_h1
    Cx_p1 = I.Cx_p1
    Cy_p1 = I.Cy_p1
    Cx_q1 = I.Cx_q1
    Cy_q1 = I.Cy_q1

    Ix_h1 = I.Ix_h1
    Iy_h1 = I.Iy_h1
    Ix_p1 = I.Ix_p1
    Iy_p1 = I.Iy_p1
    Ix_q1 = I.Ix_q1
    Iy_q1 = I.Iy_q1


    h31 = I.h31
    p31 = I.p31
    q31 = I.q31

    y1 = I.y1
    y2 = I.y2


    t += y1*Δt

    n_it1=div(n_t1,2)
    n_it3=div(n_t3,2)

    @. p3 = pprev3 + (Δt * y1) * x1_p3
    @. q3 = qprev3 + (Δt * y1) * x1_q3
    @. h3 = hprev3 + (Δt * y1) * x1_h3


    for i in 1:n_it3
        Gp3(eval_gp3, h3, p3, q3, t)
        @. x1_p3 = (x1_p3 + (Δτ3/Δt) * (eval_gp3 )) * dd3
        @. p3 = pprev3 + (Δt * y1) * x1_p3

        Gq3(eval_gq3, h3,p3, q3, t)
        @. x1_q3 = (x1_q3 + (Δτ3/Δt) * (eval_gq3 )) * dd3
        @. q3 = qprev3 + (Δt * y1) * x1_q3


        Gh3(eval_gh3,h3,p3, q3, t)
        @. x1_h3 = (x1_h3 + (Δτ3/Δt) * ( eval_gh3)) * dd3
        @. h3 = hprev3 + (Δt * y1) * x1_h3


        Gq3(eval_gq3, h3,p3, q3, t)
        @. x1_q3 = (x1_q3+ (Δτ3/Δt) * (eval_gq3 )) * dd3
        @. q3 = qprev3 + (Δt * y1) * x1_q3


        Gp3(eval_gp3, h3, p3, q3, t)
        @. x1_p3 = (x1_p3 + (Δτ3/Δt) * (eval_gp3 )) * dd3
        @. p3 = pprev3 + (Δt * y1) * x1_p3


        Gh3(eval_gh3,h3,p3, q3, t)
        @. x1_h3 = (x1_h3 + (Δτ3/Δt) * ( eval_gh3)) * dd3
        @. h3 = hprev3 + (Δt * y1) * x1_h3


        #err_f=norm(Eval.h-( ξh-hprev)/Δt, Inf)
        #@show norm(eval_gh - x1_h)
    end
```

```
##Interpolate x1-stages
   mul!(h31,Ix_h1,x1_h3)
   mul!(x1_h1,h31,Iy_h1)

   mul!(p31,Ix_p1,x1_p3)
   mul!(x1_p1,p31,Iy_p1)

   mul!(q31,Ix_q1,x1_q3)
   mul!(x1_q1,q31,Iy_q1)


   @. p1 = pprev1 + Δt * y1 * x1_p1
   @. q1 = qprev1 + Δt * y1 * x1_q1
   @. h1 = hprev1 + Δt * y1 * x1_h1

   for i in 1:n_it1
       Gp1(eval_gp1, h1, p1, q1, t)
       @. x1_p1 = (x1_p1 + (Δτ1/Δt) * (eval_gp1 )) * dd1
       @. p1 = pprev1 + (Δt * y1) * x1_p1

       Gq1(eval_gq1, h1,p1, q1, t)
       @. x1_q1 = (x1_q1 + (Δτ1/Δt) * (eval_gq1 )) * dd1
       @. q1 = qprev1 + (Δt * y1) * x1_q1


       Gh1(eval_gh1,h1,p1, q1, t)
       @. x1_h1 = (x1_h1 + (Δτ1/Δt) * ( eval_gh1)) * dd1
       @. h1 = hprev1 + (Δt * y1) * x1_h1


       Gq1(eval_gq1, h1,p1, q1, t)
       @. x1_q1 = (x1_q1 + (Δτ1/Δt) * (eval_gq1 )) * dd1
       @. q1 = qprev1 + (Δt * y1) * x1_q1


       Gp1(eval_gp1, h1, p1, q1, t)
       @. x1_p1 = (x1_p1 + (Δτ1/Δt) * (eval_gp1 )) * dd1
       @. p1 = pprev1 + (Δt * y1) * x1_p1


       Gh1(eval_gh1,h1,p1, q1, t)
       @. x1_h1 = (x1_h1 + (Δτ1/Δt) * ( eval_gh1)) * dd1
       @. h1 = hprev1 + (Δt * y1) * x1_h1

   end

   ####Compute Second Stage
   t+=y2*Δt

   @. h3 = hprev3 + Δt * (y2 * x1_h3 + y1 * x2_h3)
   @. p3 = pprev3 + Δt * (y2 * x1_p3 + y1 * x2_p3)
   @. q3 = qprev3 + Δt * (y2 * x1_q3 + y1 * x2_q3)


   for i in 1:n_t3
```

```
        Gp3(eval_gp3, h3, p3, q3, t)
        @. x2_p3 = (x2_p3 + (Δτ3/Δt) * eval_gp3 ) * dd3
        @. p3 = pprev3 + Δt * (γ2 * x1_p3 + γ1 * x2_p3)

        Gq3(eval_gq3, h3, p3, q3, t)
        @. x2_q3 = (x2_q3 + (Δτ3/Δt) * eval_gq3 ) * dd3
        @. q3 = qprev3 + Δt * (γ2 * x1_q3 + γ1 * x2_q3)

        Gh3(eval_gh3, h3, p3, q3, t)
        @. x2_h3 = (x2_h3 + (Δτ3/Δt) * eval_gh3 ) * dd3
        @. h3 = hprev3 + Δt * (γ2 * x1_h3 + γ1 * x2_h3)

        Gq3(eval_gq3, h3, p3, q3, t)
        @. x2_q3 = (x2_q3 + (Δτ3/Δt) * eval_gq3 ) * dd3
        @. q3 = qprev3 + Δt * (γ2 * x1_q3 + γ1 * x2_q3)

        Gp3(eval_gp3, h3, p3, q3, t)
        @. x2_p3 = (x2_p3 + (Δτ3/Δt) * eval_gp3 ) * dd3
        @. p3 = pprev3 + Δt * (γ2 * x1_p3 + γ1 * x2_p3)

        Gh3(eval_gh3, h3, p3, q3, t)
        @. x2_h3 = (x2_h3 + (Δτ3/Δt) * eval_gh3 ) * dd3
        @. h3 = hprev3 + Δt * (γ2 * x1_h3 + γ1 * x2_h3)

    end

    if print_err
        Gh3(eval_gh3, h3, p3, q3, t)
        @show norm(eval_gh3 - x2_h3, Inf)
        @show t/(3600*24)
    end


    mul!(h31,Ix_h1,x2_h3)
    mul!(x2_h1,h31,Iy_h1)

    mul!(p31,Ix_p1,x2_p3)
    mul!(x2_p1,p31,Iy_p1)

    mul!(q31,Ix_q1,x2_q3)
    mul!(x2_q1,q31,Iy_q1)


   @. h1 = hprev1 + Δt * (γ2 * x1_h1 + γ1 * x2_h1)
   @. p1 = pprev1 + Δt * (γ2 * x1_p1 + γ1 * x2_p1)
   @. q1 = qprev1 + Δt * (γ2 * x1_q1 + γ1 * x2_q1)

    for i in 1:n_t1

        Gp1(eval_gp1, h1, p1, q1, t)
        @. x2_p1 = (x2_p1 + (Δτ1/Δt) * eval_gp1 ) * dd1
        @. p1 = pprev1 + Δt * (γ2 * x1_p1 + γ1 * x2_p1)

        Gq1(eval_gq1, h1, p1, q1, t)
        @. x2_q1 = (x2_q1 + (Δτ1/Δt) * eval_gq1 ) * dd1
        @. q1 = qprev1 + Δt * (γ2 * x1_q1 + γ1 * x2_q1)
        Gh1(eval_gh1, h1, p1, q1, t)
        @. x2_h1 = (x2_h1 + (Δτ1/Δt) * eval_gh1 ) * dd1
        @. h1 = hprev1 + Δt * (γ2 * x1_h1 + γ1 * x2_h1)

        Gq1(eval_gq1, h1, p1, q1, t)
        @. x2_q1 = (x2_q1 + (Δτ1/Δt) * eval_gq1 ) * dd1
        @. q1 = qprev1 + Δt * (γ2 * x1_q1 + γ1 * x2_q1)

        Gp1(eval_gp1, h1, p1, q1, t)
        @. x2_p1 = (x2_p1 + (Δτ1/Δt) * eval_gp1 ) * dd1
        @. p1 = pprev1 + Δt * (γ2 * x1_p1 + γ1 * x2_p1)

        Gh1(eval_gh1, h1, p1, q1, t)
        @. x2_h1 = (x2_h1 + (Δτ1/Δt) * eval_gh1 ) * dd1
        @. h1 = hprev1 + Δt * (γ2 * x1_h1 + γ1 * x2_h1)

    end
    if (printerr && (t<(3600*24*10)-Δt*21))
        Gh1(eval_gh1, h1, p1, q1, t)
        @show norm(eval_gh1 - x2_h1, Inf)
    end



    @. I.hprev1 = h1
    @. I.pprev1 = p1
    @. I.qprev1 = q1

    ## transfer solution to coarse grid
    mul!(h31,h1, Cy_h1)
    mul!(h3, Cx_h1, h31)

    mul!(p31,p1, Cy_p1)
    mul!(p3, Cx_p1, p31)

    mul!(q31,q1, Cy_q1)
    mul!(q3, Cx_q1, q31)

    # ##transfor stages to coarser grids, as approximations.

    mul!(h31,x1_h1, Cy_h1)
    mul!(x1_h3, Cx_h1, h31)

    mul!(p31,x1_p1, Cy_p1)
    mul!(x1_p3, Cx_p1, p31)

    mul!(q31,x1_q1, Cy_q1)
    mul!(x1_q3, Cx_q1, q31)

    # #  ##transfor stages to coarser grids, as approximations.

    mul!(h31, x2_h1, Cy_h1)
    mul!(x2_h3, Cx_h1, h31)

    mul!(p31, x2_p1, Cy_p1)
    mul!(x2_p3, Cx_p1, p31)

    mul!(q31, x2_q1, Cy_q1)
    mul!(x2_q3, Cx_q1, q31)


    @. I.hprev3 = h3
    @. I.pprev3 = p3
    @. I.qprev3 = q3

    nothing
end
```

B
Applying Pseudo-Time-Stepping to Crank-Nicolson- and BDF2-schemes

B.1. Applying Pseudo-Time-Stepping Techniques to Various Implicit Methods

In this ection we explain how pseudo-time-stepping can be applied to Crank Nicolson and BDF2-methods.

Applying Pseudo-Time-Stepping to the Crank-Nicolson Scheme

Remark: From this point forward, we adopt the following notation to simplify Equations (7.8)–(7.10):

$$\frac{\partial h}{\partial t} = F_h(p) + G_h(q), \tag{B.1}$$

$$\frac{\partial p}{\partial t} = F_p(h, p, q, t), \tag{B.2}$$

$$\frac{\partial q}{\partial t} = F_q(h, p, q). \tag{B.3}$$

Applying the Crank-Nicolson method from (7.3) to Problem (B.1) - (B.3) yields:

$$h^{n+1} = h^n + \frac{\Delta t}{2}(F_h(p^{n+1}) + G_h(q^{n+1}) + F_h(p^n) + G_h(q^n)), \tag{B.4}$$

$$p^{n+1} = p^n + \frac{\Delta t}{2}\left(F_p(h^{n+1}, p^{n+1}, q^{n+1}, t^{n+1}) + F_p(h^n, p^n, q^n, t^n)\right), \tag{B.5}$$

$$q^{n+1} = q^n + \frac{\Delta t}{2}\left(F_q(h^{n+1}, p^{n+1}, q^{n+1}) + F_q(h^n, p^n, q^n)\right). \tag{B.6}$$

Similarly to the previous section, in Equations (B.4)–(B.6), the variables h^{n+1}, p^{n+1}, and q^{n+1} are approximated by ξ_h, ξ_p, and ξ_q, respectively that are again made dependent on the pseudo-time variable τ. Furthermore, a τ-dependent derivative is introduced, leading to the following pseudo-time systems:

$$\frac{\partial \xi_h}{\partial \tau} = \frac{1}{2}\left(F_h(\xi_p) + G_h(\xi_q) + F_h(p^n) + G_h(q^n)\right) - \frac{\xi_h - h^n}{\Delta t} \tag{B.7}$$

$$\frac{\partial \xi_p}{\partial \tau} = \frac{1}{2}\left(F_p(\xi_h, \xi_p, \xi_q, t^{n+1}) + F_p(h^n, p^n, q^n, t^n)\right) - \frac{\xi_p - p^n}{\Delta t}, \tag{B.8}$$

$$\frac{\partial \xi_q}{\partial \tau} = \frac{1}{2}\left(F_q(\xi_h, \xi_p, \xi_q) + F_q(h^n, p^n, q^n)\right) - \frac{\xi_q - q^n}{\Delta t}. \tag{B.9}$$

The equilibrium solutions of Equations (B.7)-(B.9) are the solutions h^{n+1}, p^{n+1} and q^{n+1} in the next time step $n + 1$.

CN_IM: Applying explicit pseudo time-integration For solving Equations (B.7)-(B.9) numerically an explicit Forward Euler method is used. We continue computing time-interations, until τ-independet solutions are found.

Note, in our study we choose to implicitly integrate the $\frac{\xi_\mathbf{h}}{\Delta t}$, $\frac{\xi_\mathbf{p}}{\Delta t}$ and $\frac{\xi_\mathbf{q}}{\Delta t}$-components from Equations (B.7) - (B.9), respectively.

To solve the system, with h^n, p^n, and q^n known from the previous time step, we first compute the following function evaluations, which depend solely on these known variables. The stages $k_{1,h}$, $k_{1,p}$, and $k_{1,q}$ are then computed as follows:

$$k_{1,h} = F_h(p^n) + G_h(q^n), \quad k_{1,p} = F_p(h^n, p^n, q^n, t^n), \quad k_{1,q} = F_q(h^n, p^n, q^n). \tag{B.10}$$

Furthermore, applying a pseudo-time integration with respect to pseudo-time step τ, yields:

$$\xi_h^{k+1} = \xi_h^k + \Delta\tau \left(\frac{1}{2} k_{1,h} + \frac{1}{2}\left(F_h(\xi_p^k) + G_h(\xi_q^k)\right) - \frac{\xi_h^{k+1} - h^n}{\Delta t} \right), \tag{B.11}$$

$$\xi_p^{k+1} = \xi_p^k + \Delta\tau \left(\frac{1}{2} k_{1,p} + \frac{1}{2} F_p(\xi_h^k, \xi_p^k, \xi_q^k, t^{n+1}) - \frac{\xi_p^{k+1} - p^n}{\Delta t} \right), \tag{B.12}$$

$$\xi_q^{k+1} = \xi_q^k + \Delta\tau \left(\frac{1}{2} k_{1,q} + \frac{1}{2} F_q(\xi_h^k, \xi_p^k, \xi_q^k) - \frac{\xi_q^{k+1} - q^n}{\Delta t} \right). \tag{B.13}$$

The initial approximations, ξ_h^0, ξ_p^0, and ξ_q^0, are set equal to h^n, p^n, and q^n, respectively. These approximations correspond to the solutions computed in the previous time iteration at time t^n.

Since our goal is to iterate till a stationary solution of (B.7)-(B.9) is found, we define the following stopping criterion:

$$\left\| \frac{1}{2}(F_h(\xi_p^{k+1}) + G_h(\xi_q^{k+1}) + k_{1,h}) - \frac{\xi_h^{k+1} - p^n}{\Delta t} \right\|_\infty < 1e - 8.$$

If this stopping criterion is satisfied, the solutions at the next time step, i.e., at t^{n+1}, will be:

$$h^{n+1} = \xi_h^{k+1}, \quad , p^{n+1} = \xi_p^{k+1}, \quad q^{n+1} = \xi_q^{k+1}.$$

CN_IMEX: Applying Semi-Explicit Pseudo-Time-Integration In the fully explicit pseudo time-solver we updated the pseudo-time values with function evaluations by values computed in the previous pseudo-time step. To improve the stability properties of the pseudo-time solver,one can choose to firstly update only the ξ_p^{k+1} variable and use this value to update ξ_q^{k+1}. Then, the update for ξ_h^{k+1} can be done entirely implicitly with only function evaluations by the latest updates ξ_p^{k+1} and x_q^{k+1}.

To ensure symmetry between the updates of ξ_p and ξ_q, the value of ξ_q^{k+2} is first updated using the values of ξ_h^{k+1}, ξ_p^{k+1}, and ξ_q^{k+1}. Subsequently, ξ_p^{k+2} is updated with the values of ξ_h^{k+1}, ξ_p^{k+1}, and ξ_q^{k+2}. Finally, the update for ξ_h^{k+2} is performed fully implicitly. The following equations represent this semi-explicit scheme:

$$\xi_p^{k+1} = \xi_p^k + \Delta\tau \left(\frac{1}{2} k_{1,p} + \frac{1}{2} F_p(\xi_h^k, \xi_p^k, \xi_q^k, t^{n+1}) - \frac{\xi_p^{k+1} - p^n}{\Delta t} \right),$$
$$\xi_q^{k+1} = \xi_q^k + \Delta\tau \left(\frac{1}{2} k_{1,q} + \frac{1}{2} F_q(\xi_h^k, \xi_p^{k+1}, \xi_q^k) - \frac{\xi_q^{k+1} - q^n}{\Delta t} \right),$$
$$\xi_h^{k+1} = \xi_h^k + \Delta\tau \left(\frac{1}{2} k_{1,h} + \frac{1}{2} \left(F_h(\xi_p^{k+1}) + G_h(\xi_q^{k+1}) \right) - \frac{\xi_h^{k+1} - h^n}{\Delta t} \right),$$
$$\xi_q^{k+2} = \xi_q^{k+1} + \Delta\tau \left(\frac{1}{2} k_{1,q} + \frac{1}{2} F_q(\xi_h^{k+1}, \xi_p^{k+1}, \xi_q^{k+1}) - \frac{\xi_q^{k+2} - q^n}{\Delta t} \right),$$
$$\xi_p^{k+2} = \xi_p^{k+1} + \Delta\tau \left(\frac{1}{2} k_{1,p} + \frac{1}{2} F_p(\xi_h^{k+1}, \xi_p^{k+1}, \xi_q^{k+2}, t^{n+1}) - \frac{\xi_p^{k+2} - p^n}{\Delta t} \right),$$
$$\xi_h^{k+2} = \xi_h^{k+1} + \Delta\tau \left(\frac{1}{2} k_{1,h} + \frac{1}{2} \left(F_h(\xi_p^{k+2}) + G_h(\xi_q^{k+2}) \right) - \frac{\xi_h^{k+2} - h^n}{\Delta t} \right).$$

Applying Pseudo-Time-Stepping to the BDF2-Scheme

In this subsection the process of solving the linear system corresponding to the BDF2-method given by Equation (7.4) is described.

First of all, applying the BDF2-method from Equation (7.4) to the time-dependent problem (B.1)-(B.3), yields the following equations:

$$h^{n+1} = \frac{4}{3} h^n - \frac{1}{3} h^{n-1} + \frac{2}{3} \Delta t F_h(p^{n+1}) + \frac{2}{3} \Delta t G_h(q^{n+1}), \tag{B.14}$$
$$p^{n+1} = \frac{4}{3} p^n - \frac{1}{3} p^{n-1} + \frac{2}{3} \Delta t F_p(h^{n+1}, p^{n+1}, q^{n+1}, t^{n+1}), \tag{B.15}$$
$$q^{n+1} = \frac{4}{3} q^n - \frac{1}{3} + \frac{2}{3} \Delta t F_q(h^{n+1}, p^{n+1}, q^{n+1}). \tag{B.16}$$

Similar to the cases of Backward Euler and Crank-Nickelson, the unknown variables h^{n+1}, p^{n+1}, and q^{n+1} are approximated by the τ-dependent variables ξ_h, ξ_p, and ξ_q, respectively. Furthermore, derivatives with respect to τ are introduced in Equations (B.17)–(B.19).

$$\frac{\partial \xi_h}{\partial \tau} = \frac{2}{3} \left(F_h(\xi_p) + G_h(\xi_q) \right) - \frac{\xi_h - \frac{4}{3} h^n + \frac{1}{2} h^{n-1}}{\Delta t}, \tag{B.17}$$
$$\frac{\partial \xi_p}{\partial \tau} = \frac{2}{3} F_p(\xi_h, \xi_p, \xi_q, t^{n+1}) - \frac{\xi_p - \frac{4}{3} p^n + \frac{1}{2} p^{n-1}}{\Delta t}, \tag{B.18}$$
$$\frac{\partial \xi_q}{\partial \tau} = \frac{2}{3} F_q(\xi_h, \xi_p, \xi_q) - \frac{\xi_q - \frac{4}{3} q^n + \frac{1}{2} q^{n-1}}{\Delta t}. \tag{B.19}$$

Similar to the previous subsections as $\tau \to \infty$, $\xi_h \to h^{n+1}$, $\xi_p \to p^{n+1}$ and $\xi_q \to q^{n+1}$. The equilibrium solutions of (B.17)-(B.19) in our study are obtained, by using a pseudo-time-integration scheme and performing pseudo-time-iterations till an equilibrium solution emerges.

BDF2_IMApplying explicit pseudo-time-integration For solving Equations (B.17)-(B.19) numerically an explicit Euler Forward method is used. We continue computing time-interations, until τ-independet solutions are found.

Note that, in our study, we choose to implicitly integrate the $\frac{\xi_\mathbf{h}}{\Delta t}$, $\frac{\xi_\mathbf{p}}{\Delta t}$, and $\frac{\xi_\mathbf{q}}{\Delta t}$ components of Equations (B.17)–(B.19), respectively.

The explicit pseudo-iterations for solving Problem (B.17)-(B.19) are equal to:

$$\xi_h^{k+1} = \xi_h^k + \Delta\tau \left(\frac{2}{3}\left(F_h(\xi_p^k) + G_h(\xi_q^k)\right) - \frac{\xi_h^{k+1} - \frac{4}{3}h^n + \frac{1}{2}h^{n-1}}{\Delta t} \right), \tag{B.20}$$

$$\xi_p^{k+1} = \xi_p^k + \Delta\tau \left(\frac{2}{3}F_p(\xi_h^k, \xi_p^k, \xi_q^k, t^{n+1}) - \frac{\xi_p^{k+1} - \frac{4}{3}p^n + \frac{1}{2}p^{n-1}}{\Delta t} \right), \tag{B.21}$$

$$\xi_q^{k+1} = \xi_q^k + \Delta\tau \left(\frac{2}{3}F_q(\xi_h^k, \xi_p^k, \xi_q^k) - \frac{\xi_q^{k+1} - \frac{4}{3}q^n + \frac{1}{2}q^{n-1}}{\Delta t} \right). \tag{B.22}$$

The first approximations, ξ_h^0, ξ_p^0 and ξ_q^0 are equal to h^n, p^n and q^n, respectively, where values h^n, p^n and q^n are solutions from the previous time step.

Since our goal is to iterate till a stationary solution of (B.17)-(B.19) is found, we define the following stopping criterion:

$$\left\| \frac{2}{3}\left(F_h(\xi_p^{k+1}) + G_h(\xi_q^{k+1})\right) - \frac{\xi_h^{k+1} - \frac{4}{3}h^n + \frac{1}{2}h^{n-1}}{\Delta t} \right\|_\infty < 1e-8.$$

If this stopping criterion is met, the solutions in the next time step will be:

$$h^{n+1} = \xi_h^{k+1}, \quad p^{n+1} = \xi_p^{k+1}, \quad q^{n+1} = \xi_q^{k+1}.$$

BDF2_IMEX: Applying Semi-Explicit Pseudo-Time-Integration In the fully explicit pseudo-time solver we updated the pseudo-time values by function evaluations with values computed in the previous pseudo-time step. To improve the stability properties of the solver, one approach is to first update only the ξ_p^{k+1} variable and use this value to update ξ_q^{k+1}. Subsequently, the update for ξ_h^{k+1} can be performed fully implicitly, relying solely on function evaluations of ξ_p^{k+1} and ξ_q^{k+1}.

To ensure symmetry between the updates of ξ_p and ξ_q, the value of ξ_q^{k+2} is updated first, using the values of ξ_h^{k+1}, ξ_p^{k+1}, and ξ_q^{k+1}. Subsequently, ξ_p^{k+2} is updated with the most recent updates of ξ_h^{k+1}, ξ_p^{k+1}, and ξ_q^{k+2}. Finally, the update for ξ_h^{k+2} is performed fully implicitly. The schemes are presented below:

$$\xi_p^{k+1} = \xi_p^k + \Delta\tau \left(\frac{2}{3}F_p(\xi_h^k, \xi_p^k, \xi_q^k, t^{n+1}) - \frac{\xi_p^{k+1} - \frac{4}{3}p^n + \frac{1}{2}p^{n-1}}{\Delta t} \right), \tag{B.23}$$

$$\xi_q^{k+1} = \xi_q^k + \Delta\tau \left(\frac{2}{3}F_q(\xi_h^k, \xi_p^{k+1}, \xi_q^k) - \frac{\xi_q^{k+1} - \frac{4}{3}q^n + \frac{1}{2}q^{n-1}}{\Delta t} \right), \tag{B.24}$$

$$\xi_h^{k+1} = \xi_h^k + \Delta\tau \left(\frac{2}{3}\left(F_h(\xi_p^{k+1}) + G_h(\xi_q^{k+1})\right) - \frac{\xi_h^{k+1} - \frac{4}{3}h^n + \frac{1}{2}h^{n-1}}{\Delta t} \right). \tag{B.25}$$

$$\xi_q^{k+2} = \xi_q^{k+1} + \Delta\tau \left(\frac{2}{3}F_q(\xi_h^{k+1}, \xi_p^{k+1}, \xi_q^{k+1}) - \frac{\xi_q^{k+2} - \frac{4}{3}q^n + \frac{1}{2}q^{n-1}}{\Delta t} \right), \tag{B.26}$$

$$\xi_p^{k+2} = \xi_p^{k+1} + \Delta\tau \left(\frac{2}{3}F_p(\xi_h^{k+1}, \xi_p^{k+1}, \xi_q^{k+2}, t^{n+1}) - \frac{\xi_p^{k+2} - \frac{4}{3}p^n + \frac{1}{2}p^{n-1}}{\Delta t} \right), \tag{B.27}$$

$$\xi_h^{k+2} = \xi_h^{k+1} + \Delta\tau \left(\frac{2}{3}\left(F_h(\xi_p^{k+2}) + G_h(\xi_q^{k+2})\right) - \frac{\xi_h^{k+2} - \frac{4}{3}h^n + \frac{1}{2}h^{n-1}}{\Delta t} \right). \tag{B.28}$$

B.2. Applying Pseudo-Times by only Approximating h^{n+1} as a Pseudo-Time Variable

CN_FW: Approximating h^{n+1} in the Crank Nicolson-scheme

Applying Crank Nicolson to Problem (B.1) - (B.3) yields the following time iterations:

$$h^{n+1} = h^n + \frac{\Delta t}{2}\left(F_h(p^{n+1}) + G_h(q^{n+1}) + F_h(p^n) + G_h(q^n)\right), \tag{B.29}$$

$$p^{n+1} = p^n + \frac{\Delta t}{2}\left(F_p(h^{n+1}, p^{n+1}, q^{n+1}, t^{n+1}) + F_p(h^n, p^n, q^n, t^n)\right), \tag{B.30}$$

$$q^{n+1} = q^n + \frac{\Delta t}{2}\left(F_q(h^{n+1}, p^{n+1}, q^{n+1}) + F_q(h^n, p^n, q^n)\right). \tag{B.31}$$

Substituting p^{n+1} and q^{n+1} in Equation (B.29) yields:

$$\begin{gathered} h^{n+1} = h^n + \frac{\Delta t}{2} F_h\left(p^n + \frac{\Delta t}{2}\left(F_p(h^{n+1}, p^{n+1}, q^{n+1}, t^{n+1}) + F_p(h^n, p^n, q^n, t^n)\right)\right) + \\ \frac{\Delta t}{2} G_h\left(q^n + \frac{\Delta t}{2}\left(F_q(h^{n+1}, p^{n+1}, q^{n+1}) + F_q(h^n, p^n, q^n)\right)\right) + \frac{\Delta t}{2}\left(F_h(p^n) + G_h(q^n)\right). \end{gathered}$$

Furthermore, we approximate h^{n+1} with ξ:

$$\xi \approx h^{n+1}$$

Substitute the values for ξ, in the outer time-iterations yields:

$$\begin{gathered} \xi = h^n + \frac{\Delta t}{2} F_h\left(p^n + \frac{\Delta t}{2}\left(F_p(\xi, p^{n+1}, q^{n+1}, t^{n+1}) + F_p(h^n, p^n, q^n, t^n)\right)\right) + \\ \frac{\Delta t}{2} G_h\left(q^n + \frac{\Delta t}{2}\left(F_q(\xi, p^{n+1}, q^{n+1}) + F_q(h^n, p^n, q^n)\right)\right) + \frac{\Delta t}{2}\left(F_h(p^n) + G_h(q^n)\right). \end{gathered}$$

Finding the solution of ξ in the previous equation is equivalent to finding the root of the following equation:

$$\begin{gathered} G(\xi) = -\frac{\xi - h^n}{\Delta t} + \frac{1}{2} F_h\left(p^n + \frac{\Delta t}{2}\left(F_p(\xi, p^{n+1}, q^{n+1}, t^{n+1}) + F_p(h^n, p^n, q^n, t^n)\right)\right) + \\ \frac{1}{2} G_h\left(q^n + \frac{\Delta t}{2}\left(F_q(\xi, p^{n+1}, q^{n+1}) + F_q(h^n, p^n, q^n)\right)\right) + \frac{1}{2}\left(F_h(p^n) + G_h(q^n)\right) = 0. \end{gathered}$$

Furthermore the root of $G(\xi)$ can be found by defining ξ ad τ-dependent and adding a τ-derivative, as following:

$$\frac{d\xi}{d\tau} = G(\xi),$$

where the equilibrium solution is equal to the root of G. This ODE will be solved with a first order Euler Forward time-integration method.

With the following procedure the approximations of the unknowns in the Crank Nickelson approximation are found:

$$
\begin{aligned}
p^{n+1,k+1} &= p^n + \frac{\Delta t}{2}\left(F_p(\xi^k, p^{n+1,k}, q^{n+1,k}, t^{n+1}) + k_p\right),\\
q^{n+1,k+1} &= q^n + \frac{\Delta t}{2}\left(F_q(\xi^k.p^{n+1,k+1}, q^{n+1,k}) + k_q\right),\\
\xi^{k+1} &= \xi^k + \Delta\tau\left(-\frac{\xi^{k+1} - h^n}{\Delta t} + \frac{1}{2}\left(F_h(p^{n+1,k+1}) + G_h(q^{n+1,k+1}) + k_h\right)\right),\\
q^{n+1,k+2} &= q^n + \frac{\Delta t}{2}\left(F_q(\xi^{k+1}, p^{n+1,k+1}, q^{n+1,k+1}) + k_q\right),\\
p^{n+1,k+2} &= p^n + \frac{\Delta t}{2}\left(F_p(\xi^{k+1}, p^{n+1,k+1}, q^{n+1,k+2}, t^{n+1}) + k_p\right),\\
\xi^{k+2} &= \xi^{k+1} + \Delta\tau\left(-\frac{\xi^{k+2} - h^n}{\Delta t} + \frac{1}{2}\left(F_h(p^{n+1,k+2}) + G_h(q^{n+1,k+2}) + k_h\right)\right),
\end{aligned}
$$

where the stages k_h, k_p and k_q are function evaluations of the variables in the previous (real)-time iteration, given by Equation (B.32).

$$
k_h = F_h(p^n) + G_h(q^n), \quad k_p = F_p(h^n, p^n, q^n, t^n), \quad k_q = F_q(h^n, p^n, q^n). \tag{B.32}
$$

The initial guesses for the approximations are:

$$
p^{n+1,0} \approx p^n, \quad q^{n+1,0} \approx q^n, \quad \xi^0 \approx h^n.
$$

where $\Delta\tau$ is the pseudo-time step and $p^{n+1,k}$ and $q^{n+1,k}$ are the k-th approximations of p^{n+1} and q^{n+1}, respectively.

BDF2_FW: Approximating h^{n+1} in the BDF2-scheme

Applying an implicit BDF2 scheme to our test-case SWEs yields:

$$
h^{n+1} = \frac{4}{3}h^n - \frac{1}{3}h^{n-1} + \frac{2}{3}\Delta t F_h(p^{n+1}) + \frac{2}{3}\Delta t G_h(q^{n+1}) \tag{B.33}
$$

$$
p^{n+1} = \frac{4}{3}p^n - \frac{1}{3}p^{n-1} + \frac{2}{3}\Delta t F_p(h^{n+1}, p^{n+1}, q^{n+1}, t^{n+1}) \tag{B.34}
$$

$$
q^{n+1} = \frac{4}{3}q^n - \frac{1}{3}q^{n-1} + \frac{2}{3}\Delta t F_q(h^{n+1}, p^{n+1}, q^{n+1}). \tag{B.35}
$$

Approximate h^{n+1} with ξ in (B.33)-(B.35):

$$
\xi \approx h^{n+1},
$$

and we obtain an equation that has to be solved for ξ

$$
\begin{aligned}
\xi = \frac{4}{3}h^n - \frac{1}{3}h^{n-1} + \frac{2}{3}\Delta t F_h\left(\frac{4}{3}p^n - \frac{1}{3}p^{n-1} + \frac{2}{3}\Delta t F_p(\xi, p^{n+1}, q^{n+1}, t^{n+1})\right) +\\
\frac{2}{3}\Delta t G_h\left(\frac{4}{3}q^n - \frac{1}{3} + \frac{2}{3}\Delta t F_q(\xi, p^{n+1}, q^{n+1})\right).
\end{aligned}
$$

Solving for ξ is equivalent with finding the root of the following function:

$$
\begin{aligned}
G(\xi) = -\frac{\xi - \frac{4}{3}h^n + \frac{1}{3}h^{n-1}}{\Delta t} +\\
\frac{2}{3}\Delta t F_h\left(\frac{4}{3}p^n - \frac{1}{3}p^{n-1} + \frac{2}{3}\Delta t F_p(\xi, p^{n+1}, q^{n+1}, t^{n+1})\right) +\\
\frac{2}{3}\Delta t G_h\left(\frac{4}{3}q^n - \frac{1}{3}q^{n-1} + \frac{2}{3}\Delta t F_q(\xi, p^{n+1}, q^{n+1})\right) = 0,
\end{aligned}
$$

to which pseudo-time-stepping can be applied such that:

$$\frac{d\xi}{d\tau} = G(\xi),$$

This ODE will be solved with a first order Euler Forward time-integration method.

With the following procedure the approximations the unknowns in the BDF2 iterations in equations (B.14)-(B.16) are found.

$$\begin{aligned}
p^{n+1,k+1} &= \frac{4}{3}p^n - \frac{1}{3}p^{n-1} + \frac{2}{3}\Delta t F_p(\xi^k, p^{n+1,k}, q^{n+1,k}, t^{n+1}),\\
q^{n+1,k+1} &= \frac{4}{3}q^n - \frac{1}{3} + \frac{2}{3}\Delta t F_q(\xi^k, p^{n+1,k+1}, q^{n+1,k}),\\
\xi^{k+1} = \xi^k + \Delta\tau &\left(-\frac{\xi^{k+1} - \frac{4}{3}h^n + \frac{1}{3}h^{n-1}}{\Delta t} + (F_h(p^{n+1,k+1}) + G_h(q^{n+1,k+1}))\right),\\
q^{n+1,k+2} &= \frac{4}{3}q^n - \frac{1}{3} + \frac{2}{3}\Delta t F_q(\xi^{k+1}, p^{n+1,k+1}, q^{n+1,k+1}),\\
p^{n+1,k+2} &= \frac{4}{3}p^n - \frac{1}{3}p^{n-1} + \frac{2}{3}\Delta t F_p(\xi^{k+1}, p^{n+1,k+1}, q^{n+1,k+2}, t^{n+1}),\\
\xi^{k+2} = \xi^{k+1} + \Delta\tau &\left(-\frac{\xi^{k+2} - \frac{4}{3}h^n + \frac{1}{3}h^{n-1}}{\Delta t} + (F_h(p^{n+1,k+2}) + G_h(q^{n+1,k+2}))\right),
\end{aligned}$$

where $\Delta\tau$ is the pseudo-time step and $p^{n+1,k}$ and $q^{n+1,k}$. are the k-th approximations of p^{n+1} and q^{n+1}, respectively. We select the following initial approximations within the inner-iterations:

$$p^{n+1,0} \approx p^n, \quad q^{n+1,0} \approx q^n, \quad \xi^0 \approx h^n.$$