UNIVERSITY
OF OSLO

Master's thesis

Basilisk CFD Simulations of Physical Wave Tank Experiments

Martin Funderud Gimse

Computational Science: Mechanics
60 ECTS study points

Department of Mathematics
Faculty of Mathematics and Natural Sciences

Spring 2025

Martin Funderud Gimse

Basilisk CFD Simulations of Physical Wave Tank Experiments

Supervisors:
Atle Jensen
Øystein Lande

Abstract

The main focus of this thesis is the comparison of physical wave tank experiments and Basilisk simulations. I have done physical experiments in a wave tank with a piston-type wave maker, generated regular Stokes waves with different steepness, and measured surface elevation. I used the computational fluid dynamics software Basilisk to simulate the waves generated in the physical experiment, with a two-phase Navier-Stokes solver and a layered solver. The simulations used for comparison were done in two dimensions, but the methods have been expanded to also work in a 3D numerical wave tank. I have tested different methods for implementing the piston and found that setting the piston speed as a boundary condition gave the best results. I compare the results generated by the 2D Basilisk simulations with the physical experiments by using surface elevation and PIV measurements from physical waves. The results show good correspondence between the physical experiments and numerical simulations for many of the waves.

Contents

List of Figures

List of Tables

Preface

Chapter 1

Acknowledgements

I would like to thank my supervisor Atle Jensen for his guidance, understanding, time, and patience during my work on this thesis. I would also like to thank Lars Willas Dreyer and Øystein Lande for many meetings and workshops with good discussions. With a special thanks to Øystein Lande for the critical help and understanding of Basilisk that allowed me to create stable simulations.

The biggest thanks of all to my wife Ingvild for picking up the slack in the rest of my life, enabling me to complete this thesis.

Part I

Introduction

1.1 Stokes waves

Stokes waves are periodic surface waves driven by gravity. This non-linear solution to the Navier-Stokes equations was proposed by Sir George Gabriel Stokes in 1847, Stokes 1847. Stokes wave theory is applicable to waves in deep to intermediate water depths like typical ocean waves. When the water depth is too shallow, they no longer yield realistic results, and other theoretical solutions are better suited. A characteristic of Stokes waves is that as they get steeper, the troughs flatten and the wave crests become sharper.

1.2 Wave tank experiments

I performed experiments in the wave tank with the same parameters as some of the runs from Jensen et al. 2001. I also performed experiments with steeper and flatter waves. The waves are generated by a piston-style wave maker. In the wave tank at the hydrodynamic lab at UIO that is 25m long and 0.5m wide.

1.3 Basilisk

Basilisk is a software package for solving partial differential equations. It is free, fully open source, and available at http://basilisk.fr. I will be using Basilisk to simulate surface gravity waves. I will be using both the solver for the full Navier-Stokes equations and the layered solver described in Stéphane Popinet 2020 where the fluid is divided into discrete layers. I will try to generate the waves from the physical experiments using different methods, both setting boundary conditions and a method of modeling a moving piston by setting the water velocity inside the simulation domain.

1.4 Particle Image Velocimetry - PIV

Particle image velocimetry-PIV is a method for measuring flow velocity where tracer particles are imaged at different times to calculate the velocity of the flow. PIV can be used to create measurements both in 2D slices and 3D volumes. In this thesis, I will perform 2D PIV on the images from Jensen et al. 2001 to be able to compare with the results from numerical simulations.

Chapter 2

Theory

2.1 Navier Stokes Equations

An incompressible viscous flow, like that of water waves, can be characterized by the variables:

- Density - ρ
- Velocity - u
- Pressure - p
- Gravitational acceleration - g
- Dynamic viscosity - μ

The fluid motion can be further described by the Navier-Stokes equations 2.1. With the incompressibility condition for divergence-free flow equation 2.2.

$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u}\cdot\nabla)\,\boldsymbol{u}\right) = -\nabla p + \rho\boldsymbol{g} + \mu\nabla^2\boldsymbol{u} \tag{2.1}$$

$$\nabla\cdot\boldsymbol{u} = 0 \tag{2.2}$$

If the viscous forces are assumed to be negligible, the Navier-Stokes equations reduce to the Euler equation 2.3.

$$\rho\left(\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u}\cdot\nabla)\,\boldsymbol{u}\right) = -\nabla p + \rho\boldsymbol{g} \tag{2.3}$$

2.2 Stokes Waves Governing Equations

Sir George Gabriel Stokes used perturbation methods to find analytical solutions Stokes 1847 to the Euler equation 2.3. In Stokes 2009 Stokes presented solutions up to the third order. Later many have expanded on the solutions using varying approaches, Rayleigh Rayleigh 1917, Levi-Cevita Levi-Civita 1925, Struik Struik 1926 up to the third order. Solutions up to the fifth order have been found by, Hunt Hunt 1953, De De 1955, Skjelbreia Skjelbreia L 1960, Fenton's fifth order solutions Fenton 1985 have long been the go-to method. New derivations of the fifth order solutions have, however, offered further improvements Zhao and Liu 2022a. I will show a derivation of the solutions to second order based on Kundu 2016 and Zhao and Liu 2022a, where the analytical solution all the way to the fifth order is presented.

2.2.1 Euler equation

The previous assumptions, of incompressibility and that the viscous forces are negligible, yield the Euler equations. If the further assumption is made that the flow is two-dimensional (plane waves - $u(x, z, t)$). With the previous assumption that the viscous forces are negligible, potential theory can be used. That is, that the velocity field is the gradient of a potential function 2.4.

$$\boldsymbol{u} = \nabla\phi \tag{2.4}$$

All these assumptions together with equation 2.3 and equation 2.2, result in equations 2.4 and 2.5. The Euler equations in potential form.

$$\rho\left(\frac{\partial\nabla\phi}{\partial t} + (\nabla\phi\cdot\nabla)\,\nabla\phi\right) = -\nabla p + \rho\boldsymbol{g} \tag{2.5}$$

$$\nabla\boldsymbol{u} = \nabla^2\phi = 0 \tag{2.6}$$

2.2.2 Boundary conditions

There are different options for fixing the position of the coordinate system.

- Setting z = 0 at the bottom of the domain
- Setting z = 0 at the still water level - SWL. The height of the water surface when water is at rest, no waves propagating.
- Setting z = 0 at the mean water level - MWL. The mean of the water level as observed at a fixed position as waves pass by.

For deep water waves, the difference between MWL and SWL is negligible Zhao and Liu 2022a. Here I will continue using the still water level as z = 0, since this is the easiest to measure in a laboratory setup.

With two-dimensional flow, the surface elevation is a function of horizontal position and time $\eta(x, t)$. Two boundary conditions arise from the free surface, the kinematic and the dynamic boundary conditions.

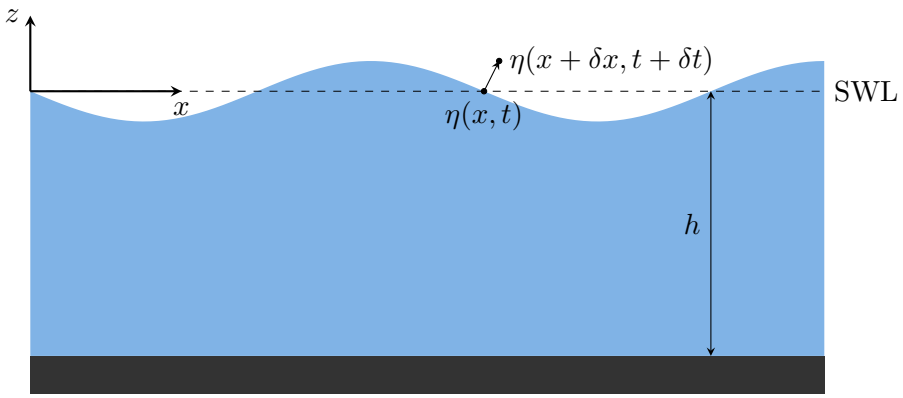

Figure 2.1: Movement of infinitesimal fluid volume as a wave moves from left to right. Still water level (SWL) - h. Used to derive the kinematic boundary condition

The kinematic boundary condition is that, at the free surface, the normal velocity of the free surface must equal the velocity of the fluid normal to the free surface. In other words, a small fluid volume at the surface stays at the surface.

At t_0, x_0 considering a small volume at the surface with elevation z_0. It will after a time δt have reached a new position $(x_0 + \delta x, z_0 + \delta z)$. At t_0, the surface elevation at x_0, is $\eta(x_0, t_0)$. Following the movement of the small fluid volume to its location at $t_0 + \delta t$ the surface elevation there is $\eta(x_0 + \delta x, t_0 \delta t)$ Summarizing this gives us equations 2.7 and 2.8

$$z_0 = \eta(x_0, t_0) \tag{2.7}$$

$$z_0 + \delta z = \eta(x_0 + \delta x, t_0 + \delta t) \tag{2.8}$$

Taylor expanding the last term in equation 2.8 as $\eta(x_0 + \delta x, t_0 + \delta t) = \eta(x_0, t_0) + \frac{\partial \eta}{\partial x}\delta x + \frac{\partial \eta}{\partial t}\delta t$ and rewriting δx as $\frac{\partial \phi}{\partial x}\delta t$ and δz as $\frac{\partial \phi}{\partial z}\delta t$ inserted in equation 2.8 together with equation 2.7 gives equation 2.9 the kinematic boundary condition.

$$\begin{gathered} \frac{\partial \eta}{\partial t} + \frac{\partial \phi}{\partial x}\frac{\partial \eta}{\partial x} = \frac{\partial \phi}{\partial z} \\ \text{at z} = \eta(x, y, t) \end{gathered} \tag{2.9}$$

The next boundary condition we get from the free surface is a condition on the pressure which is , when surface tension effects are ignored, the pressure in the air just above the free surface is equal to the pressure in the water just below the free surface. If the air pressure is then set to a constant typically 0. To find the dynamic boundary condition we can start with the Euler equation 2.5 using the identity $(\nabla\phi \cdot \nabla)\nabla\phi = \frac{1}{2}\nabla(\nabla\phi \cdot \nabla\phi) + (\nabla \times \nabla\phi) \times \nabla\phi$ and the assumption that the flow is irrotational $\nabla \times \nabla\phi = 0$ we get equation 2.10

$$\frac{\partial \nabla\phi}{\partial t} + \frac{1}{2}\nabla(\nabla\phi)^2 = -\frac{\nabla p}{\rho} - g. \tag{2.10}$$

Integrating this equation and setting $p = 0$ at $z = \eta(x, t)$ gives the Dynamic boundary condition equation 2.11

$$\begin{gathered} \frac{\partial \phi}{\partial t} + \frac{1}{2}\left(\frac{\partial \phi}{x}^2 + \frac{\partial \phi}{z}^2\right) + g\eta = 0 \\ \text{at } z = \eta(x, t) \end{gathered} \tag{2.11}$$

Finally the boundary condition for the bottom of the domain is that the vertical velocity at the bottom is 0, 2.12 no flow through the bottom boundary.

$$\begin{gathered} \frac{\partial \phi}{\partial z} = 0 \\ \text{at } z = -h \end{gathered} \tag{2.12}$$

2.2.3 Governing Equations

To summarize, the governing equations for two-dimensional Stokes waves are

$$\begin{aligned}
\rho\left(\frac{\partial \nabla\phi}{\partial t} + (\nabla\phi\cdot\nabla)\nabla\phi\right) &= -\nabla p + \rho\boldsymbol{g} \\
\nabla^2\phi &= 0 \\
\frac{\partial\phi}{\partial z} &= 0, && \text{at } z = -h \\
\frac{\partial\eta}{\partial t} + \frac{\partial\phi}{\partial x}\frac{\partial\eta}{\partial x} &= \frac{\partial\phi}{\partial z}, && \text{at z} = \eta(x,t) \\
\frac{\partial\phi}{\partial t} + \frac{1}{2}\left(\frac{\partial\phi}{x}^2 + \frac{\partial\phi}{z}^2\right) &= 0, && \text{at z} = \eta(x,t)
\end{aligned}$$

- h is the water level at rest
- $\eta(x,t)$ is the elevation of the free surface compared to the water level at rest.
- $\phi(x,z,t)$ is the velocity potential for the flow

2.3 First Order Solution

2.3.1 Waves on arbitrary depth

The linear solution is from Zhao and Liu 2022b. This first-order solution is called the Airy wave solution. Which has velocity potential equation 2.13 surface elevation equation 2.14. Equation 2.15 is the resulting horizontal velocity and equation 2.16 the vertical veloctiy.

$$\phi_1(x,z,t) = a\sqrt{\frac{g}{k\sigma}}\frac{cosh(k(h+z))}{cosh(kh)}sin(\theta) \tag{2.13}$$

$$\eta_1(x,z,t) = acos\theta \tag{2.14}$$

$$u(x,z,t) = a\sqrt{\frac{gk}{\sigma}}\frac{cosh(k(h+z))}{cosh(kh)}cos(\theta) \tag{2.15}$$

$$w(x,z,t) = a\sqrt{\frac{gk}{\sigma}}\frac{sinh(k(h+z))}{cosh(kh)}sin(\theta) \tag{2.16}$$

$$\sigma = tanh(kh) \tag{2.17}$$

$$\theta = kx - \omega t \tag{2.18}$$

2.3.2 Deep water waves

For deep water waves, the assumption is that kh is large, such that $tanh(kh) \approx 1$. This means that the wavelengths are small compared to the water depth. The velocity potential simplifies by using the relation in equation 2.19 and that $\sigma = tanh(kh) \approx 1$ inserted into equation 2.13 gives the velocity potential in equation 2.20.

$$\frac{cosh(k(h+z))}{cosh(kh)} = cosh(kz) + tanh(kh)sinh(kz) \tag{2.19}$$

$$\phi_1^{deep}(x,z,t) = a\sqrt{\frac{g}{k}}e^{kz}sin\theta \tag{2.20}$$

2.3.3 Dispersion Relation

To find the dispersion relation like in Kundu 2016. The following steps are taken. In the dynamic boundary condition equation 2.11 , the non-linear terms containing ϕ^2 are assumed small and dropped. $\frac{\partial\phi}{\partial t}$ at $z = \eta$ is also approximated with $\frac{\partial\phi}{\partial t}$ at $z = 0$. This results in the new linearized dynamic boundary condition equation 2.21:

$$\frac{\partial\phi}{\partial t} = -g\eta \qquad \text{at, } z = 0 \tag{2.21}$$

When the solutions for the potential 2.13 and surface 2.14 are inserted into equation 2.21 it allows the dispersion relation to be found 2.23

$$-a\omega\sqrt{\frac{g}{k\sigma}}\frac{cosh(k(h+0))}{cosh(kh)}cos(\theta) = -gacos(\theta) \tag{2.22}$$

$$\omega = \sqrt{gktanh(kh)} \tag{2.23}$$

The phase speed of the waves is given by $c = \frac{\omega}{k} = \sqrt{\frac{g}{k}tanh(kh)}$ Kundu 2016. This tells us that waves with longer wavelengths, lower k, will move faster than shorter waves, waves with higher k.

2.4 Second order Solution

2.4.1 Arbitrary depth

This second-order solution for waves on arbitrary depth is from Zhao and Liu 2022b, with ϕ_1 the same as in equation 2.13.

$$\phi_2 = \phi_1 + \frac{3\omega_0 a^2}{8sinh^4(kh)}cosh(2k(z+h))sin(2\theta) + C_1kx + \frac{\omega_0^2a^2}{\sigma}C_2t, \tag{2.24}$$

$$\eta = \frac{ka^2}{4}\left(\frac{3-\sigma^2}{\sigma^3}cos(2\theta) + \frac{\sigma^2-1}{\sigma}\right) - ka^2C_2, \tag{2.25}$$

$$\sigma = tanh(kh), \tag{2.26}$$

$$\omega_0^2 = gk\sigma, \tag{2.27}$$

$$C_2 = \frac{\sigma^2-1}{4\sigma} \tag{2.28}$$

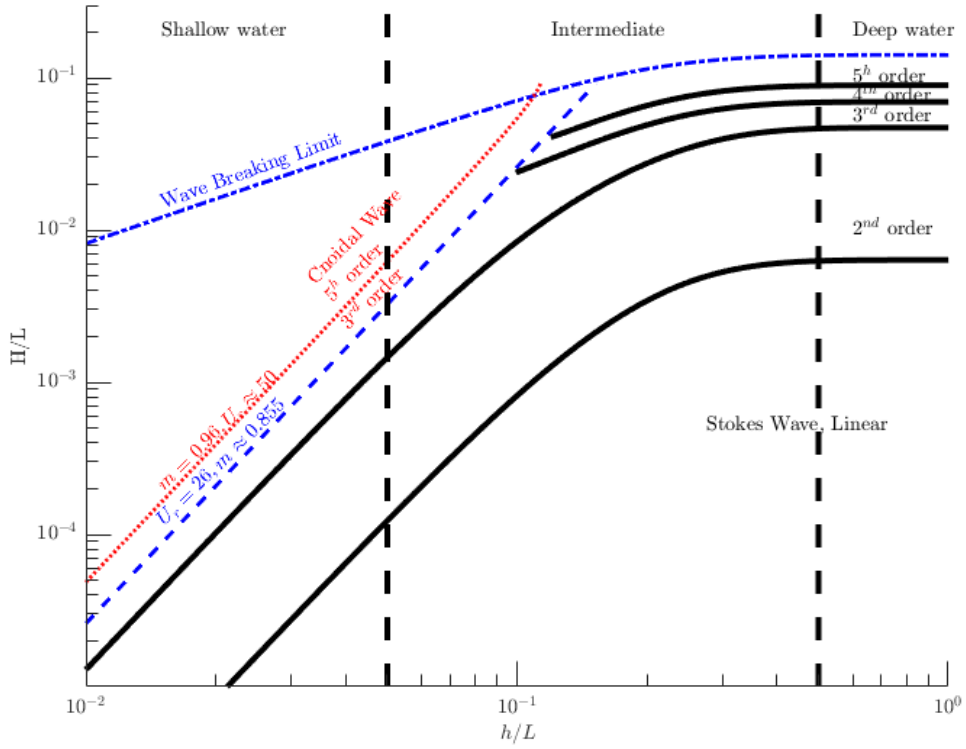

Figure 2.2: A guide to selecting the appropriate analytical solution based on wave height H, wave length L and water depth h. Figure generated using the code from Zhao, Wang and Liu 2024 where they update the figure from Bernard Le Mehaute Le Mehaute 1976 to fit their updated 5th order wave theory Zhao and Liu 2022a. H is wave height, L wave length and h water depth.

2.4.2 Deep water waves

C_1 is a constant representing steady uniform horizontal drift, which can be set to zero if desired. The mean of the free surface is 0 from the way we set the coordinate system, which from 2.25 leads to $C_2 = \frac{\sigma^2-1}{4\sigma}$. When we now apply the deep water assumption that kh is large all the terms disappear except for ϕ_1, leaving us with the same velocity potential as for the linear case 2.13 in equation 2.29.

$$\phi_2^{Deep} = \phi_1^{Deep} = \frac{ga}{\omega} e^{kz} sin\theta \tag{2.29}$$

2.5 Selecting Wave Theory

As the order of the solutions increases, the complexity also increases significantly. So how high order Stokes wave theory must be used to get good results is discussed and shown in a figure in Le Mehaute 1976. Zhao et al. have updated the figure for their updated 5th-order theory Zhao and Liu 2022a. I have included the figure from Zhao, Wang and Liu 2024 in Figure 2.2 . This figure shows which order of Stokes waves theory is needed to get results for the crest height, within 1%. It is based on wave height H, wave length L and water depth h.

2.6 Basilisk

Basilisk is a program for numerical solution of partial differential equations. That can be used to solve the incompressible variable density Navier-Stokes equations. In my numerical simulations, I will be using the Navier-Stokes and multilayer solvers, with various extensions.

2.6.1 Navier Stokes Solver

The Navier-Stokes solver for incompressible flow in Basilisk Stephane Popinet 2025c, is a collection of methods to find a numerical approximation to the equations 2.30 - 2.32, from Stephane Popinet 2025c. Where D is the deformation tensor and $\mathbf{a}$ represents external forces.

$$\partial_t \mathbf{u} + \nabla \cdot (\mathbf{u} \otimes \mathbf{u}) = \frac{1}{\rho}\left[-\nabla p + \nabla \cdot (2\mu \mathbf{D})\right] + \mathbf{a} \tag{2.30}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2.31}$$

$$\mathbf{D} = [\nabla \mathbf{u} + (\nabla \mathbf{u})^T]/2 \tag{2.32}$$

Bell-Colella-Glaz Advection Scheme

The Bell-Colella-Glaz advection scheme Bell, Colella and Glaz 1989, is a projection method that is second order accurate provided $\Delta t = O(\Delta x, \Delta y)$. It starts with utilizing Hodge decomposition, which states that "a vector field V can be uniquely decomposed into a divergence-free component V^d ... and the gradient of some scalar ϕ" Bell, Colella and Glaz 1989. $\mathbf{P}$ is then an orthogonal projection, which projects to a divergence-free vector space such that $\mathbf{P}V = V^d$. Rewriting 2.30 without external forces, normalizing density, rewriting the viscous forces and utilizing the projection $\mathbf{P}$ gives the equations 2.33 and 2.34 from Bell, Colella and Glaz 1989.

$$\partial_{\mathbf{t}} \mathbf{u} + \nabla p = \epsilon \Delta \mathbf{u} - (\mathbf{u} \cdot \nabla)\mathbf{u} \tag{2.33}$$

$$\partial_{\mathbf{t}} \mathbf{u} = \mathbf{P}(\epsilon \Delta \mathbf{u} - (\mathbf{u} \cdot \nabla)\mathbf{u}) \tag{2.34}$$

The Bell-Colella-Glaz scheme uses the temporal discretization 2.35.

$$\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} + \nabla p^{n+1/2} = \frac{\epsilon}{2}\Delta(\mathbf{u}^n + \mathbf{u}^{n+1}) - [(\mathbf{u} \cdot \nabla)\mathbf{u}]^{n+1/2} \tag{2.35}$$

Where u^n is $\mathbf{u}$ at time step n. It is then assumed that the half time step values for u are explicitly computable to second order accuracy from the velocity field at t^n and the current approximation to ∇p Bell, Colella and Glaz 1989. Equations 2.33 and 2.34 are then used with 2.35 to construct the following equation for the right-hand side of 2.35

$$(\mathbf{I} - \mathbf{P})(\frac{\epsilon}{2}\Delta(\mathbf{u}^n + \mathbf{u}^{n+1}) - [(\mathbf{u} \cdot \nabla)\mathbf{u}]^{n+1/2}) = \nabla p^{n+1/2} \tag{2.36}$$

This equation is then solved by an iterative approach which is second order accurate after only one iteration Bell, Colella and Glaz 1989.

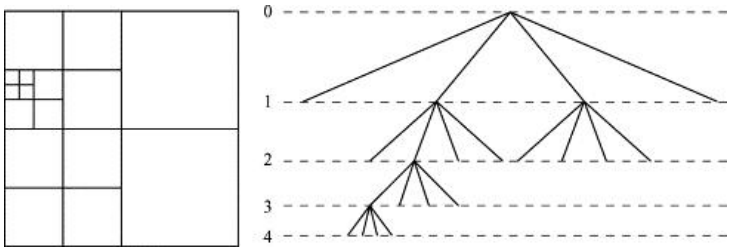

Figure 2.3: Example of quadtree discretization from Stéphane Popinet 2003

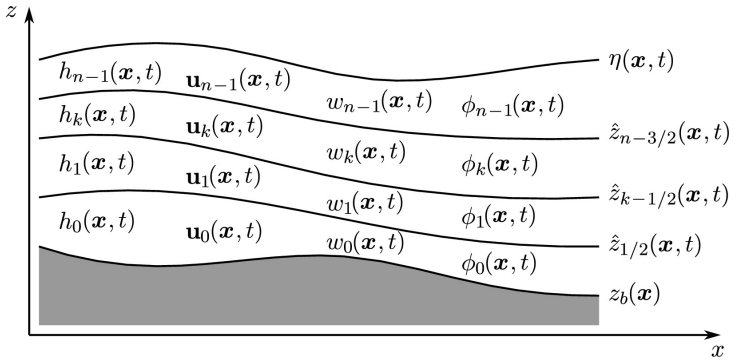

Figure 2.4: Fluid divided into n layers. Figure from Stéphane Popinet 2020

Basilisk Tree Domain Discretization

To be able to use adaptive meshes I used the tree mesh spatial discretization from Basilisk described closer in Stéphane Popinet 2003. Basilisk uses quadtrees for two dimensions and octrees for three dimensions. The following is for the 2D case. The whole domain is considered as a cell of level 0. This cell can then be divided into four equally sized child cells which have level 1, which then can be divided into smaller child cells of each individual cell. Which will be of level 2. This procedure can then be repeated up to the maximum level of refinement. An example of a mesh with maximum level 4 in Figure 2.3. In Basilisk three following rules from Stéphane Popinet 2003 are imposed on the levels of the cells.

- The levels of directly neighbouring cells cannot differ by more than one.
- The levels of diagonally neighbouring cells can not differ by more than one.
- All the cells directly neighbouring a mixed cell must be at the same level.

Each cell has four (in 2D) boundaries to the neighbouring cells called faces. Vectors and scalars in Basilisk can be defined at either the cell center, the cell faces or the vertexes of the cells.

2.6.2 Multilayer Solver

The multilayer solver assumes a fluid divided into n horizontal layers, with no mass exchange between the layers. The fluid is bounded by the free surface $\eta(\mathbf{x},t)$,and a bottom topography $z_b(\mathbf{x})$, as seen in Figure 2.4. Each layer has a height $h_k(\mathbf{x},t)$, horizontal velocity field $\mathbf{u}_k(\mathbf{x},t)$, vertical velocity $w_k(\mathbf{x},t)$, non-hydrostatic pressure $\phi_k(\mathbf{x},\mathbf{t})$ all dependent on only horizontal position $\mathbf{x}$ and t.

For the hydrostatic case, the governing equations for the layered system are equations 2.37, 2.38 and 2.39 Stéphane Popinet 2020.

$$\partial_t h_k + \nabla \cdot (h\mathbf{u})_k = 0 \tag{2.37}$$
$$\partial_t (h\mathbf{u})_k + \nabla \cdot (h\mathbf{u}\mathbf{u})_k = -gh_k \nabla(\eta) \tag{2.38}$$
$$\nabla \cdot (h\mathbf{u})_k + [w - \mathbf{u} \cdot \nabla z]_k = 0 \tag{2.39}$$

And for the non-hydrostatic case equation 2.40-2.43 with non-hydrostatic pressure ϕ.

$$\partial_t h_k + \nabla \cdot (h\mathbf{u})_k = 0 \tag{2.40}$$
$$\partial_t (h\mathbf{u})_k + \nabla \cdot (h\mathbf{u}\mathbf{u})_k = -gh_k \nabla(\eta) - \nabla(h\phi)_k + [\phi \nabla z]_k \tag{2.41}$$
$$\partial_t (hw)_k + \nabla \cdot (hw\mathbf{u})_k = -[\phi]_k \tag{2.42}$$
$$\nabla \cdot (h\mathbf{u})_k + [w - \mathbf{u} \cdot \nabla z]_k = 0 \tag{2.43}$$

2.7 Particle Image Velocimetry - PIV

PIV is an optical measurement method used to measure velocity. PIV relies on a tracer that is assumed to follow the fluid motion. The tracer can either be naturally occurring or added to the flow. It is important that the tracer follows the flow well for the quality of the results. The tracer should be neutrally buoyant. For particles, they should be small enough that their movement is not dominated by inertia or drag. However, the particles must be large enough to reflect enough light to the camera. For 2D PIV the tracer particles are illuminated by a light sheet, created by a laser and optics or other light source. Images are taken of the illuminated particles, creating a time series of images. Like the two images in the top row of Figure 2.5.

The calculation of the Eulerian velocity field is done by subdividing the image into smaller subwindows, typically with overlap. A subwindow from each of the images in the top row of Figure 2.5 is shown in Figure 2.6. When examining the two subwindows, one can see patterns of particles changing position. To find the change in position, the pattern from the first subwindow is attempted matched to a pattern in the same vicinity in the second image. The subwindow in the second image is moved around in a search range to find the best match to the subwindow from the first image. The change in position of the pattern is used to find the mean velocity inside the subwindow. How well the subwindows match is often quantified by using the cross-correlation, equation 2.44. Where the intensity of the pixels at the same location in the subwindow is multiplied together to quantify how well the images match. Different image manipulation techniques like increasing contrast, normalizing intensity, or other image filtering techniques and other correlation methods can also be used WILLERT and GHARIB 1991.

$$R_{cc} = \sum_{k=0}^{n} \sum_{l=0}^{n} I_0(k,l) I_1(k,l) \tag{2.44}$$

In HydrolabPIV Kolaas 2016, which I used for my calculations, additional methods to speed up the calculations, like FFT based cross-correlation, is used.

HydrolabPIV also uses sub-pixel interpolation to further refine the resulting velocity. This is done by fitting a polynomial to the peak of the cross-correlation function Nobach and Honkanen 2005. Such that the change in position can be estimated with greater

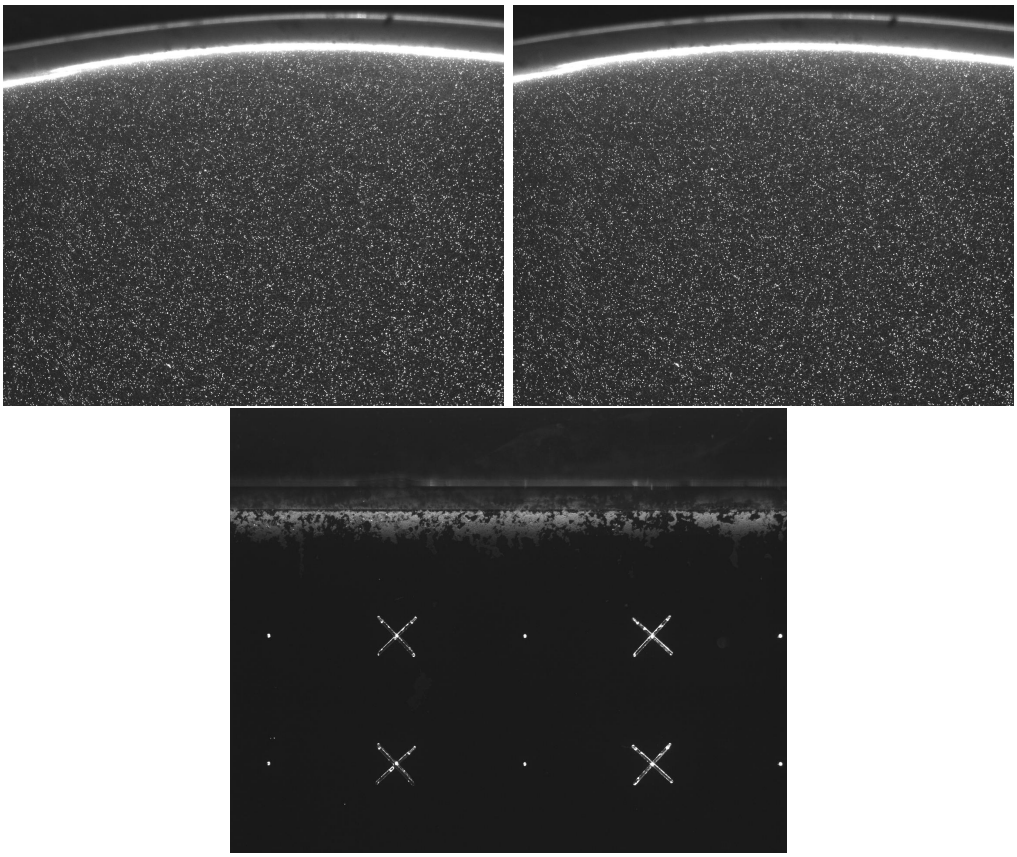

Figure 2.5: PIV Images from Jensen et al. 2001. Top row: Images taken of the tracer particles at two different time 0.012s apart. Bottom row: The image used for the coordinate system calibration

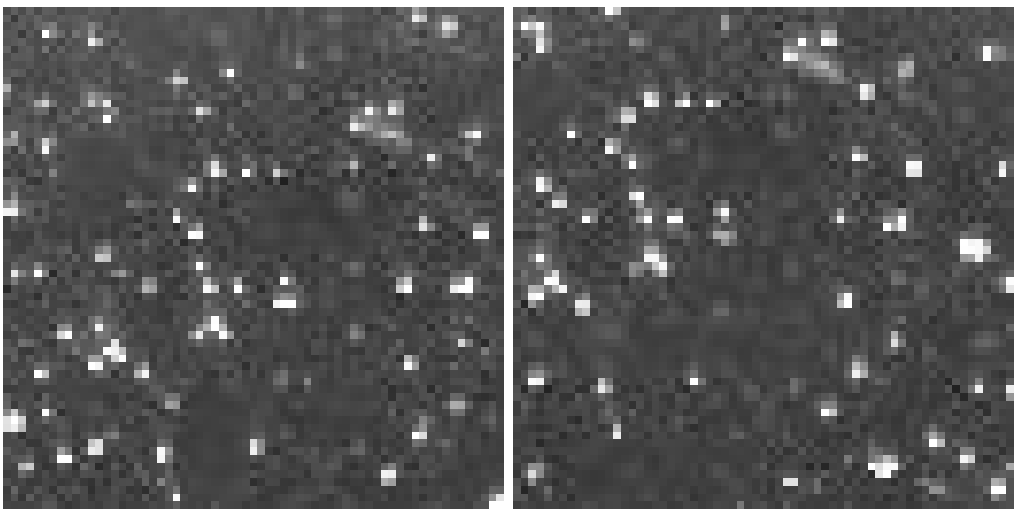

Figure 2.6: 64x64 pixel subwindows from the same location in the images in Figure 2.5. One can see the patterns of tracer particles change position between the images.

accuracy than the pixel size. This works well as long as the tracer particle size is 2-3 pixels or larger, otherwise it could lead to pixel-locking or peak-locking Raffel et al. 2018. To reduce outliers in the resulting velocity field it is possible to first perform a PIV pass, run the whole PIV procedure, with a larger subwindow. Then with a smaller subwindow, run the procedure again and use the displacement found from the first pass as an initial guess for the displacement.

Part II

Experiments and Simulations

Chapter 3

Experiments

3.1 Wave tank setup

The wave tank used for the experiments has the following dimensions.

- Length L = 24.6 m from the piston to the wave damper
- Width w = 0.5 m
- water depth h = 0.6 m

In the end of the wave tank, opposite the piston, there is a semi-permeable beach to dampen the reflections of the waves.

In the software controlling the piston, the piston movement is provided as the position of the piston, equation 3.1.

$$0.044U \tanh(t) \sin(2\pi f t)m \tag{3.1}$$

Where U is the voltage supplied to the piston and f the frequency. This movement does not fit entirely with either the supplied voltage or piston movement measured by the software, Figure 3.1, there is a difference in the start of the movement. Because of this, I used the measured position of the piston in the Basilisk simulations instead of the formula for the position and speed.

I started with running the piston with the same parameters as Jensen et al. 2001. Frequency 1.425Hz and water depth 0.6m and piston amplitude 0.308 Volts as input to the piston. I also ran the piston with double and half amplitude. Between each run of the piston, I waited at least 5 minutes, to let the motion in the wave tank settle. The first time I did the experiments, I had the surface probes at 3.1 and did 5 runs with parameters Table 3.2. I ran the experiments again 18/09/2024 and moved the surface probe closest to the piston to a distance of 1.5m from the piston, Table 3.3, and did 9 runs, parameters in Table 3.4.

For the wave tank experiments, The resulting measurements from the experiments are divided into many files. The ones I used are on the form

- 1/fil1.dat - The position of the piston measured in volts.
- 1/fil3.dat - The position of the piston measured by a probe in cm.
- 1/padle_ut.dat - The signal sent to the piston measured in volts.

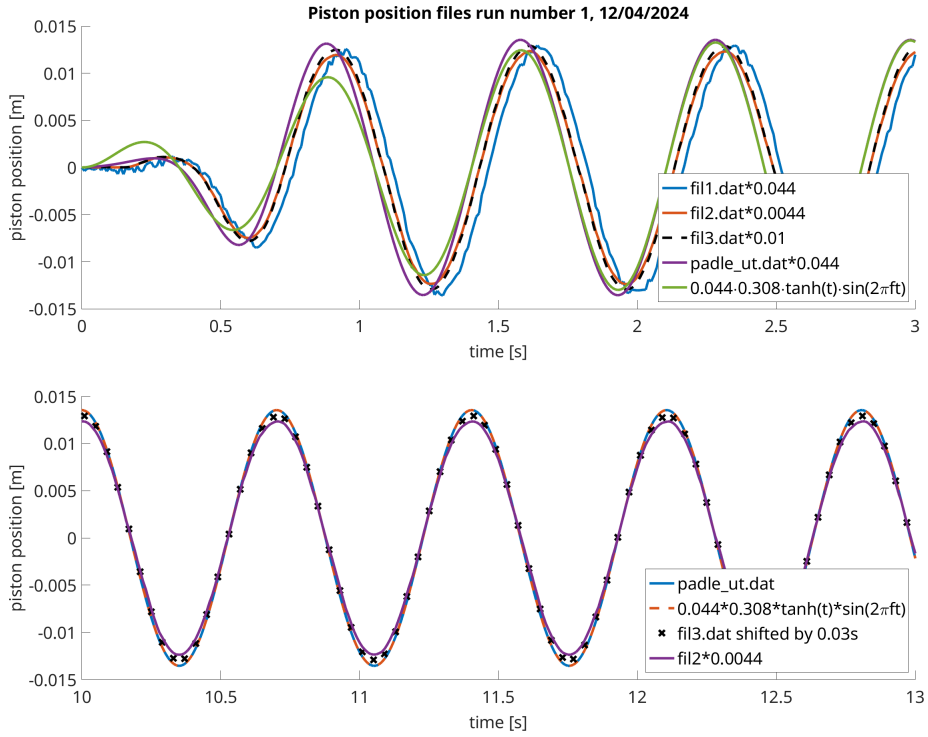

Figure 3.1: Plot of the files that log the signal sent to the piston and the measured position of the piston. Top from 0 to 3 seconds. Bottom from 10 to 13 s

Probe number	distance from piston[m]
1	8.00
2	10.05
3	10.75
4	11.50

Table 3.1: The distance from the piston at rest to the surface probes for the experiments I ran 12/04/2024

Run number	Piston amplitude [V]	f[Hz]	Piston amplitude[cm]	ak
1	0.308	1.425	1.36	0.18
2	0.308	1.425	1.36	0.18
3	0.308	1.425	1.36	0.18
4	0.616	1.425	2.71	0.31
5	0.154	1.425	0.68	0.09

Table 3.2: Piston parameters for the experiments I ran 12/04/2024. Also includes wave steepness: ak, which is calculated using Zhao and Liu 2022a and the results from the surface probes.

Probe number	distance from piston[m]
1	1.50
2	10.05
3	10.75
4	11.50

Table 3.3: The distance from the piston at rest to the surface probes for the experiments I ran 18/09/2024

Run number	Piston amplitude [V]	Piston amplitude[cm]	f[Hz]
1	0.308	1.36	1.425
2	0.308	1.36	1.425
3	0.308	1.36	1.425
4	0.616	2.71	1.425
5	0.616	2.71	1.425
6	0.616	2.71	1.425
7	0.154	0.68	1.425
8	0.154	0.68	1.425
9	0.154	0.68	1.425

Table 3.4: Piston parameters for the experiments I ran 18/09/2024. Also includes wave steepness: ak, which is calculated using Zhao and Liu 2022a and the results from the surface probes.

- 1/tid.dat - The time corresponding to the measurements in fil1.dat, fil3.dat and padle_ut.dat.
- f1425_a0_308_r1.csv - the measurements of the surface elevation at the probe positions, with one column for time with f1425 indicating 1.425 hz, a0_308 the amplitude of the signal sent to the piston and r1 the run number.

The results for all runs from both days I ran experiments are available at Gimse 2024. The position of the piston from the file fil1.dat, fil3.dat and padle_ut.dat converted to meters is shown in Figure 3.1. Here it can be seen that the signal to the piston is not quite $\tanh(t)\sin(2\pi ft)$ which is visible in the startup phase of the piston movement. The measured position of the piston also lags behind the signal by about 0.03s for fil3.dat. In the bottom plot in Figure 3.1 it can be seen that the signal to the piston from padle_ut.dat and the expected position are virtually equal. The position of the piston from fil3.dat is offset by about 0.03 seconds from the signal sent in to the piston, I assume that this simply is the delay from the piston receives a signal until it has moved. When this is taken into account the measurement of the piston position aligns very well with the position that is expected. Unfortunately, the measuring probe for fil3.dat did not work for the experiments run on 18/09/2024.

Chapter 4

Basilisk Simulations

4.1 Initialised Stokes wave

The first thing I did with Basilisk after completing installation and the tutorial was attempting to initialise a Stokes wave with the wave parameters from Jensen et al. 2001.

- water depth 0.6m
- frequency 1.425 Hz
- wave amplitude 0.0205m
- wavenumber k 7.95

I calculated the wave number using the dispersion relation solver for Fenton's 5th-order Stokes waves from Zhao and Liu 2022a

4.1.1 Multilayer solver

The first solver that I tried was the multilayer solver. I created a script based on Stephane Popinet 2024c and Stephane Popinet 2024a and matched the wave parameters from Jensen et al. 2001. I created a 2D domain and used periodic boundary conditions for the left and right boundaries of the domain such that the wave re-enters the boundary from the left as it exits the right boundary. I also set the mesh to be finer vertically closer to the surface as seen in the right-hand side of Figure 4.1. This image shows 8 layers while the later energy calculations in Figure 4.2 were done with 30 layers vertically and 512 cells horizontally.

The evolution of the kinetic and gravitational potential energy for roughly 35 wave periods can be seen in Figure 4.2. These results are similar to the basilisk results for the three-dimensional case with an initialised breaking wave before the breaking occurs Stephane Popinet 2024a. With a slight decline in total energy and a variation in kinetic and potential energy mirroring each other. I then test different combinations of number of layers and horizontal cells. In figure 4.3, it can be seen that increasing the horizontal cells decreases the loss of energy. Increasing from 10 to 20 layers also reduces the energy decay, but doubling the layers again to 40 does not give as much of an improvement.

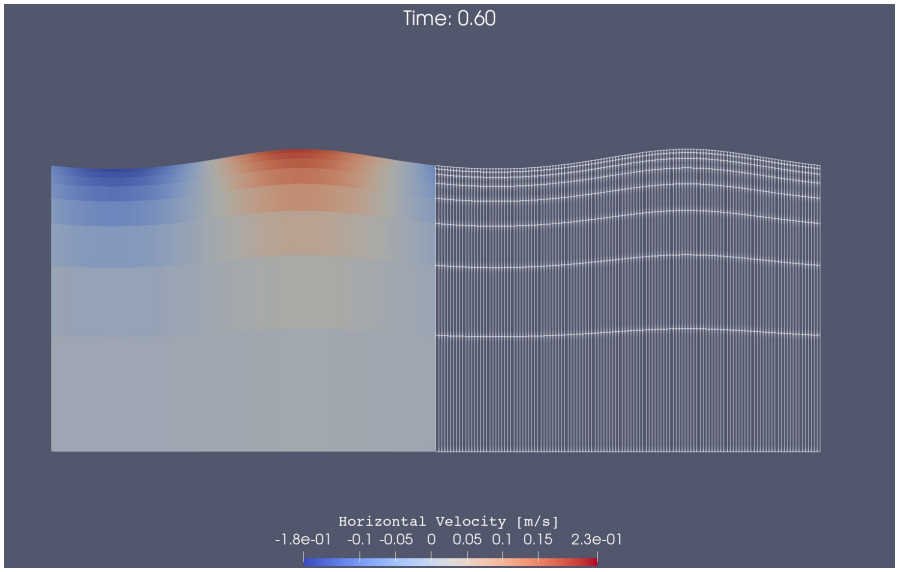

Figure 4.1: Initialised Stokes wave with periodic boundaries. Horizontal Velocity to the left and mesh cells to the right

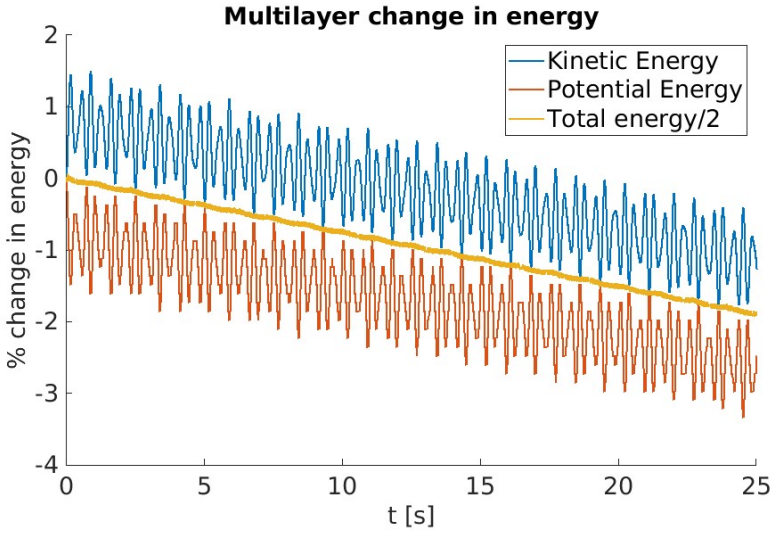

Figure 4.2: Kinetic and Potential energy for multilayer initialised wave. Percentage change in energy compared to the kinetic energy at t=0. 512 cells in x direction, 30 layers.

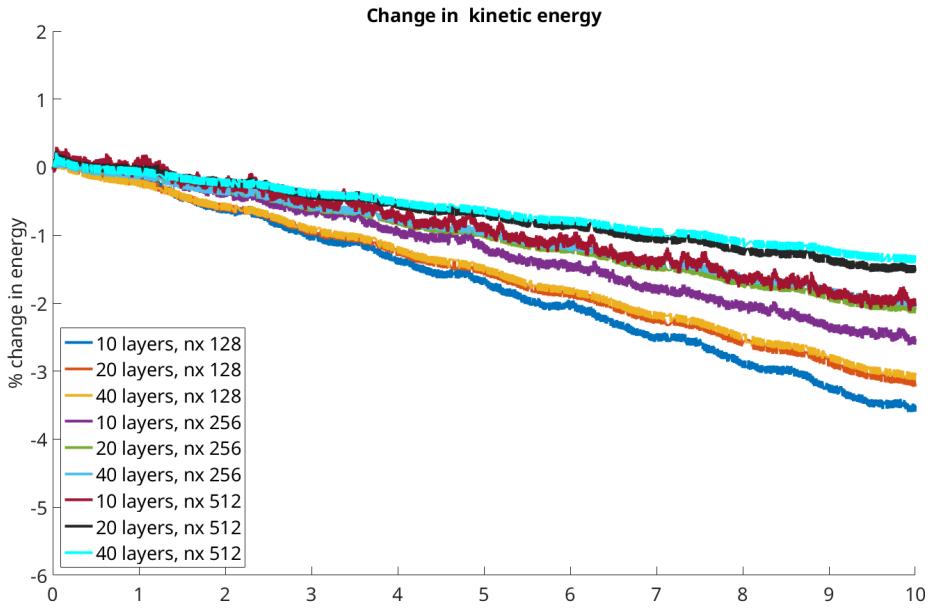

Figure 4.3: Kinetic and Potential energy for multilayer initialised wave. Percentage change in energy compared to the kinetic energy at t=0. 30 layers and 512 cells horizontally. There is a larger difference when changing from 10 to 20 than from 20 to 40 layers. Doubling the resolution in x helps the solution more than additional layers when there are 20 layers.

4.1.2 Incompressible Navier-Stokes solver

I then initialized the same wave in the Incompressible Navier–Stokes solver Stephane Popinet 2025c I used the 2D breaking wave test case stokes-ns.c Stephane Popinet 2024b as a starting point. Since it is the two-phase equivalent to the multilayer case I used previously. I also changed the mesh adaptation function to Øystein's "Adapt_wavelet_leave_interface.h" Øystein Lande 2018 which keeps higher piston refinement in the area close to the air-water interface. The resulting mesh is shown in 4.4 where it can be compared to the mesh with no mesh adaptation.

I had some problems with the stability of the solver, the amplitude would grow as the simulation progressed. I tried to mitigate this by limiting the maximum time step, this increased simulation times but helped the amplitude stay more stable. A time step of the same magnitude as the size of the smallest cells worked well. The evolution of the total energy for different time steps is shown in Figure 4.5.

4.2 Numerical Wave Tank with Piston

To be able to compare with physical experiments, I wanted to create a numerical wave tank with a piston wave generator. A vertical paddle on the left side of the domain creating waves travelling towards the right.

4.2.1 Stephane's Piston Trick

My first attempt at simulating the wave tank and piston in 2D was to use what is nicknamed "Stephane's trick", as implemented in Hooft 2018 by Anton van Hooft. Which is a solution by Stephane Popinet to implement a moving piston. It is done by defining

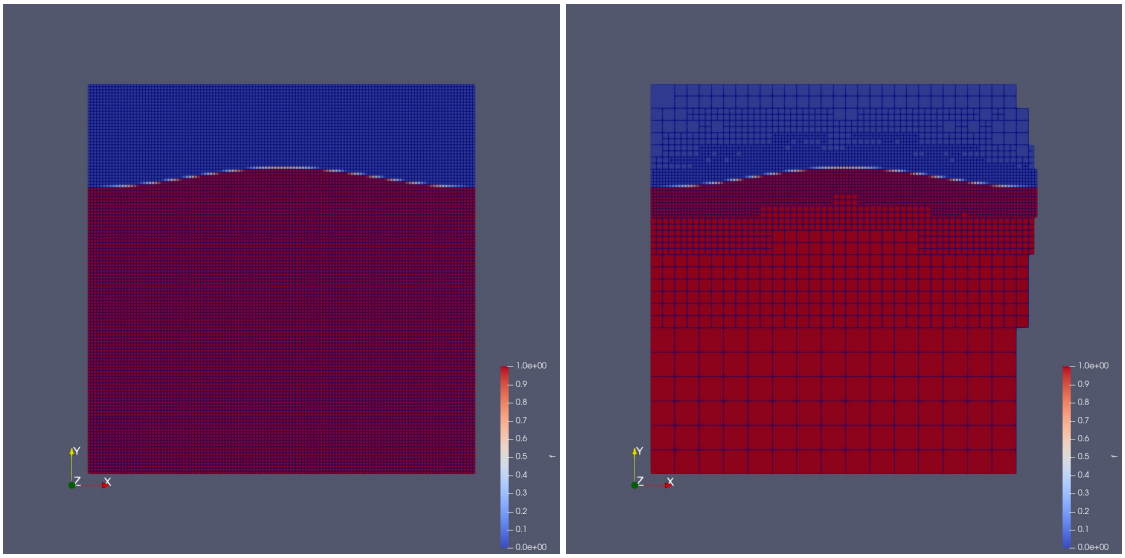

Figure 4.4: The difference in the resulting mesh when using an adaptive mesh. Regular mesh on the left. Adaptive mesh in the right image. The cells on the rightmost boundary are hidden in the render since they connect back to the left boundary.

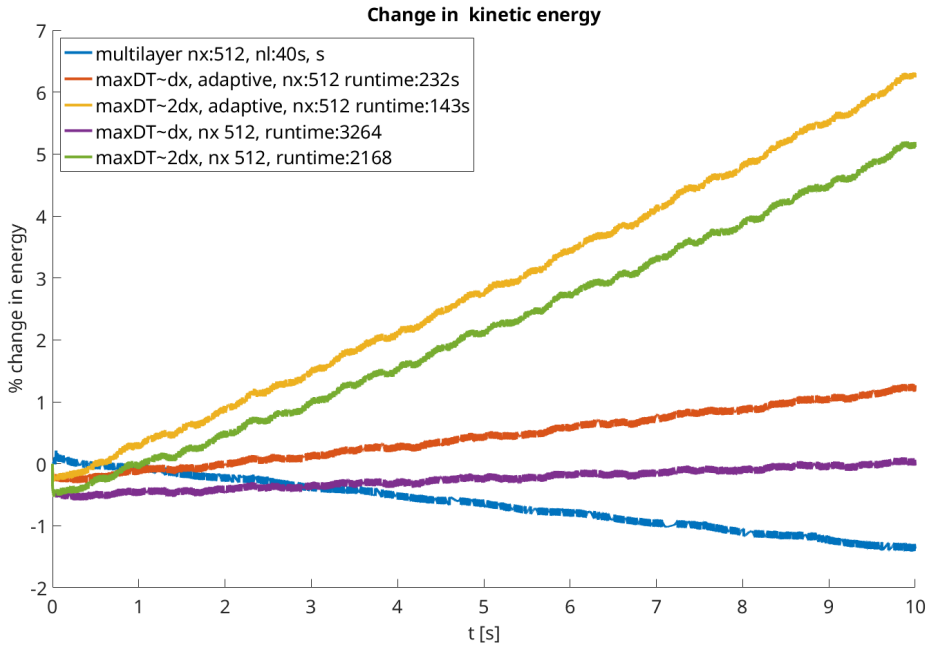

Figure 4.5: Plot of the change in the kinetic energy in the water of the initialised wave with periodic boundaries. Using adaptive mesh refinement causes the change in energy to be slightly above the change in energy when not using it. The Change in energy is much smaller when the time step is limited to be of the same magnitude as the size of the smallest cells in the mesh.

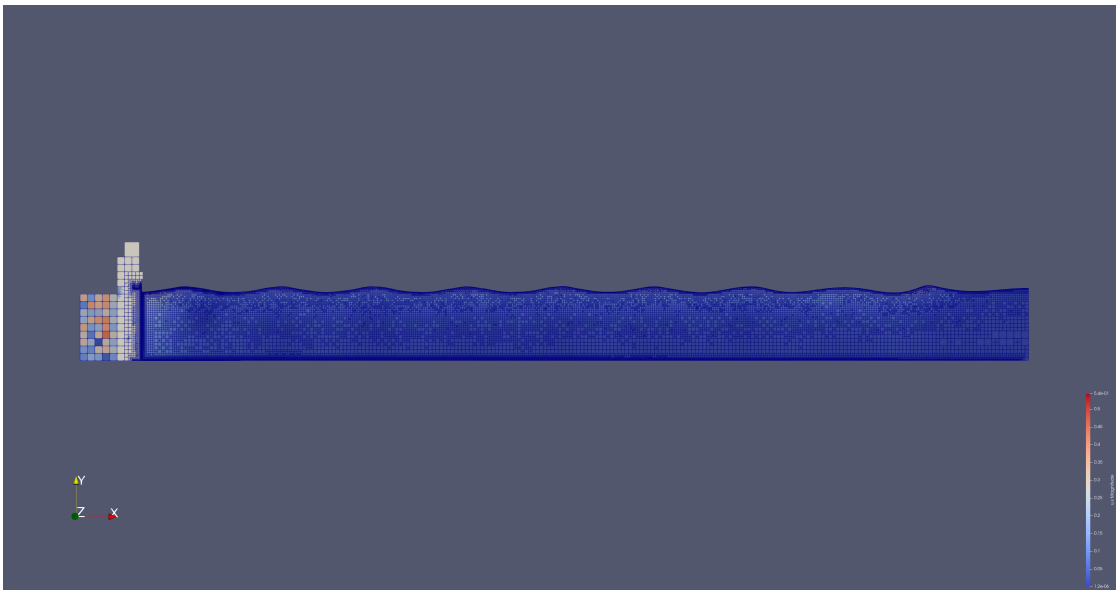

Figure 4.6: My initial implementation of the moving piston. Piston in grey, and the area to the left of the piston containing water but with a coarser mesh.

a scalar field inside the domain with the value 1 denoting inside the piston and 0 outside of the piston. The velocity field of the area inside the piston is then set to coincide with the movement speed of the piston.

Then for every time step the piston area is moved and the velocity inside the piston updated. This does, however, result in a "leaky" piston. Some water will enter and exit the piston area, however it can be reduced by increasing the refinement level of the mesh. I also made changes to adapt_wavelet_leave_interface.h Øystein Lande 2018 and created my own adapt_wavelet_leave_interface_two_levels.h which allows setting the refinement level around the piston higher than the rest of the mesh. It did help the leaking, but the smaller time step needed resulted in no significant speed up over increasing the mesh refinement on the whole domain. 4.8. Only reducing the time step also reduces the leaking from the piston.

My initial implementation also had an area behind the piston where water was free to move as in Figure 4.6. To avoid spending too much simulation time on calculating the flow behind the piston, where a lot of splashing occurred, I unrefined the mesh so it was coarse behind the piston. When I moved the piston all the way to the left boundary, the simulations would no longer be stable, as the solution for the pressure would not converge. The reasons for the pressure not converging are further discussed in section 4.3. Further details of the piston implementations are discussed in section A.1.

4.2.2 Set Boundary velocity

Next I tried simply setting the velocity on the boundary for every time-step. This had some unintended consequences like the pressure not converging for the solution and "checkerboard" or "striped" velocity as seen in Figure 4.9. Thankfully Øystein Lande explained the problem and helped me with a solution, explained in section 4.3.

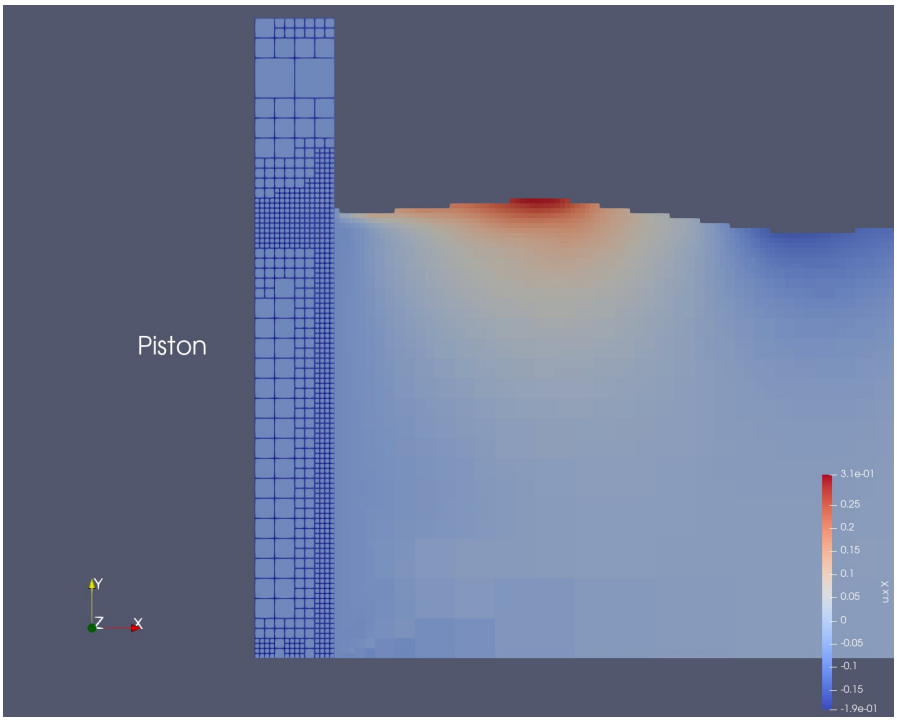

Figure 4.7: Moving piston with the piston all the way at the left boundary. Piston area to the left where the velocity is set every time step

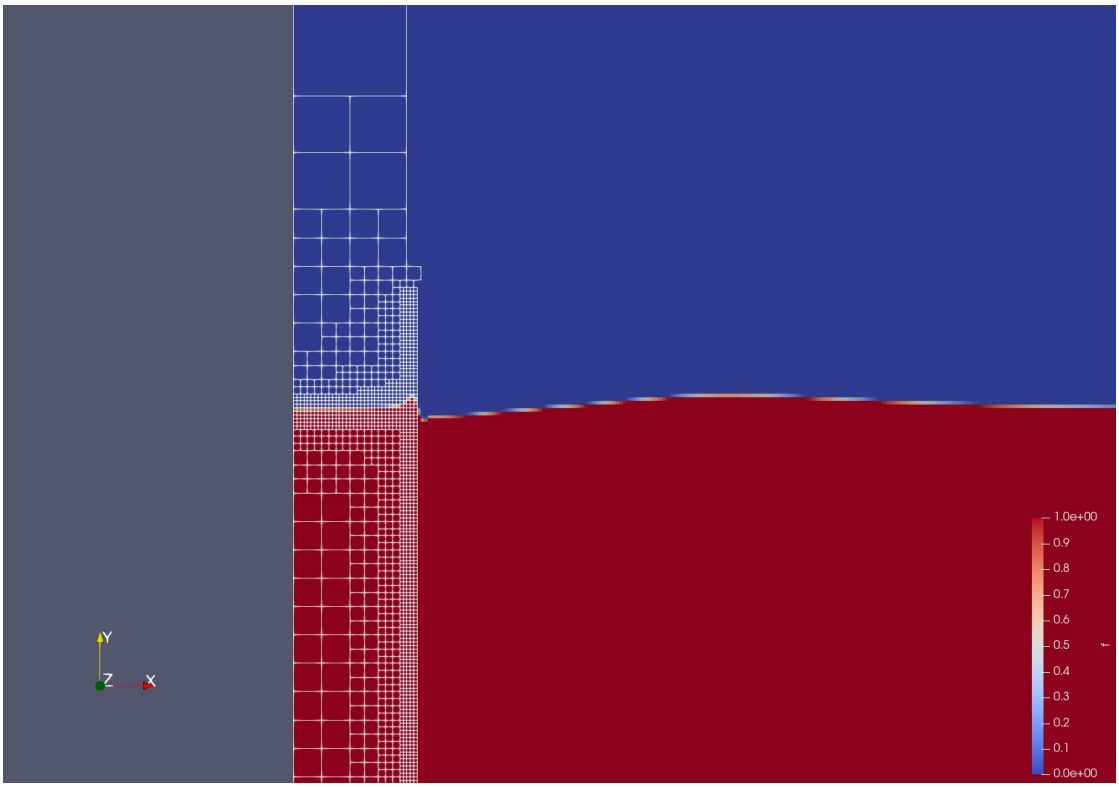

Figure 4.8: The piston area is denoted by the white wireframe. Water in red, air in blue. More water has entered the piston close to the piston boundary. Water also leaks out of the piston as it moves towards the left.

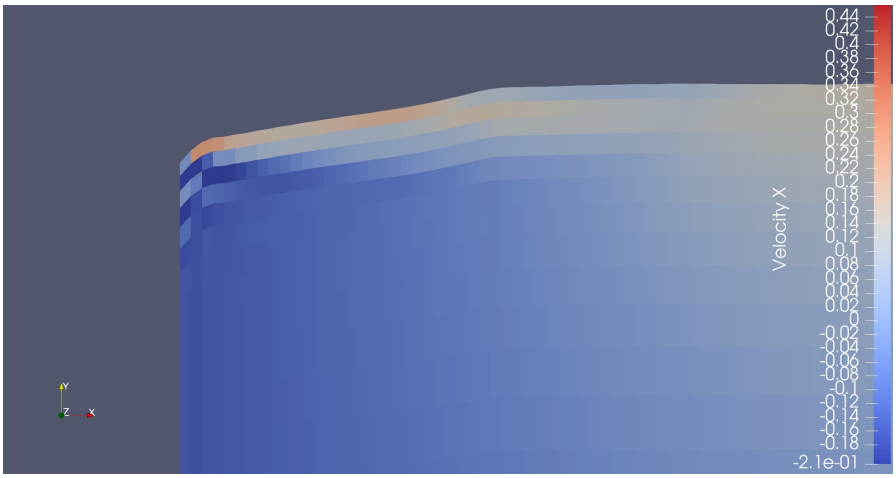

Figure 4.9: Strange pattern in the velocity at the piston boundary with the multilayer solver. Similar pattern arises with the NS solver. With both horizontal and vertical "stripes" in the velocity.

4.2.3 Multilayer solver

My first attempt at implementing a piston in the multilayer solver was to simply set the velocity on the boundary using a Basilisk event at every time step. This did work for certain piston movements and mesh/layer configurations. However, it would often lead to a chequerboard or striped pattern for the velocity close to the piston boundary as seen in Figure 4.9. I also changed the velocity set on the boundary to be a linear interpolation of the velocity at the discrete time steps from the piston speed file.

4.3 Correct Handling of the Piston Boundaries

The strange patterns in the velocity caused me much head-scratching and had me exploring many different possible solutions to avoid the problem. I eventually realized that since it didn't happen when the moving piston was placed away from the boundary, that my handling of the boundary conditions was a probable source. Luckily, Øystein Lande knew the solution to my problems and it was related to the boundary conditions. The problems arose because the default pressure boundary condition on the boundaries is a symmetric Neumann boundary condition in Basilisk. This works well for constant velocity or velocities that don't change much in time. But it does, however, lead to problems when the velocity fluctuations are larger. Calculating the boundary condition for the pressure based on the piston velocity was the solution. Another problem with my piston implementation was the way I set the velocity to a constant for all my piston implementations.

```
1 event piston (i++) {
2     u.n[left] = dirichlet(ux);
3 }
```

This doesn't allow Basilisk to update the value for the half steps in the simulation like in equation 2.35. The solution to this was to wrap the piston speed in a function dependent on t, such that Basilisk can update the velocity at the boundary at half steps. In section A.1 I go into more detail on the implementations of the piston methods.

To test the evolution of the energy over time with this piston implementation and solver, to compare with the initialized wave case. I created a piston movement with the

function 4.1 for the piston position. I ran this piston movement with f=1.425, a=0.0125m and t_1 equal to 5 wave periods.

$$piston\_position(t) = \begin{cases} a\sin(2\pi ft)\tanh(t)\tanh(t_1 - t), & t < t_1 \\ 0 & t > t_1 \end{cases} \tag{4.1}$$

This resulted in the energy plots in Figure 4.10. The stability of the kinetic energy is very dependent on the timestep. The change in energy behaved the same way when using the NS boundary method and the NS moving piston method, as they were the same apart from the piston implementation.

4.4 3D simulation

I also wanted to make a simulation of a 7x21m wave tank with 14 pistons. The options I had were limited when I started in 2023, but the Basilisk updates in 2025 has greatly increased the ways in non-cubic domains can be achieved Stephane Popinet 2025a.

4.4.1 Multilayer solver

With the multilayer solver it is possible to stack square or cubic subdomains to create rectangular domains. This means that the dimensions of the domain must be

$$l_x = n_x l, \qquad l_y = n_y l \tag{4.2}$$

Where l is the side length of the subdomains and n_x and n_y is the number of subdomains in each direction. Which can be run with OpenMP, MPI or also on GPUs. The processes used for the simulation must however be a multiple of the number of domains. This allowed me to make a 7x21m wave tank with the layered solver. Figure 4.11 shows the end of the simulation domain with the pistons coloured by velocity perpendicular to the piston face. And the mesh is visible on the side. Figure 4.12 shows the surface after 5 seconds coloured by the magnitude of the velocity.

When I tried running this on a GPU it was only possible with one layer for the layered solver. However in 2025 the Basilisk GPU capabilities were improved to allow for multiple layers.

4.4.2 Navier Stokes solver

The same method of stacking domains is not yet possible with tree grids. But the masking method I used in two dimensions works, so that it was possible to make rectangular domains to be run with OpenMP. It is possible to run non-cubic domains with MPI or on GPUs but not with tree grids, so adaptive mesh is not yet. An example of a short piston movement for 14 pistons offset by 0.1 seconds is shown in Figure 4.13.

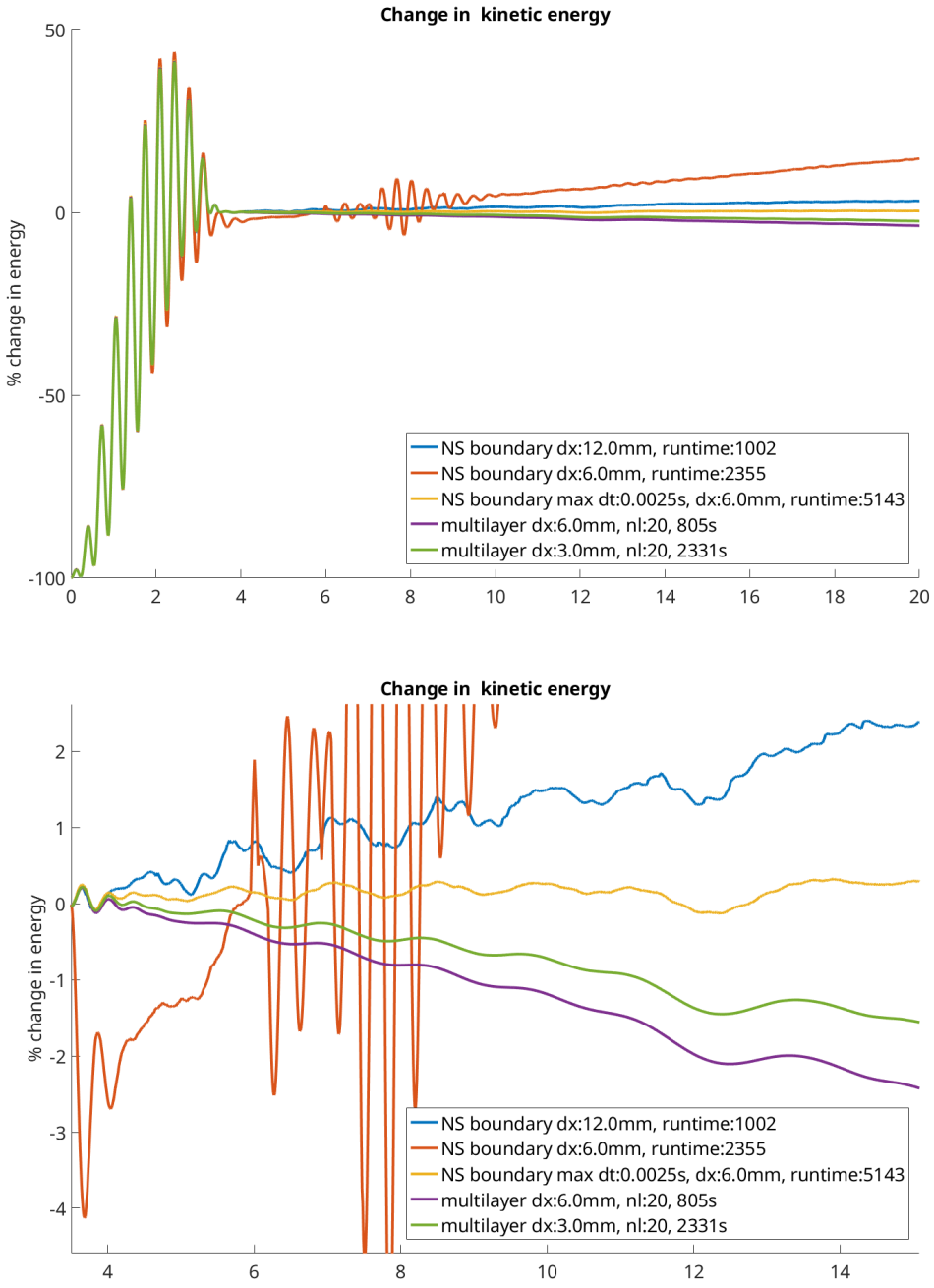

Figure 4.10: The evolution of the kinetic energy in the wave tank using a short piston movement, lasting 3.5s. The kinetic energy of each solution is compared to its own kinetic energy at 3.5s, at the end of the piston movement. Top image shows the evolution from 0 to 20 seconds where the increase in energy because of the piston movement can be seen. The bottom image shows the evolution of the kinetic energy after the piston has stopped moving. The multilayer simulations show a slight decline in kinetic energy, roughly 2% over the course of 10 seconds. With the NS solver the energy increases slightly when using when the smallest cells are 12mm but for smaller cells the solution is much less stable. However this can be helped by reducing the maximum size of the timestep, which leads to the most stable energy, but does increase the runtime. The multilayer solver does not exhibit the same effect.

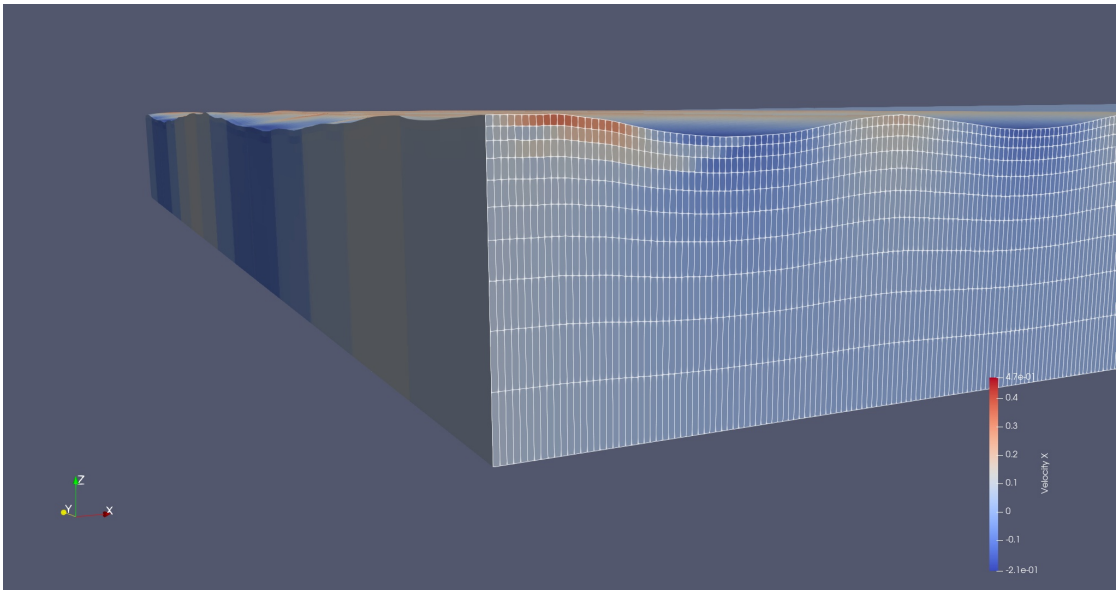

Figure 4.11: 3d multilayer solver with 14 pistons. View of the boundary with the pistons. Coloured based on the velocity perpendicular to the piston faces. The mesh can be seen on the side of the domain.

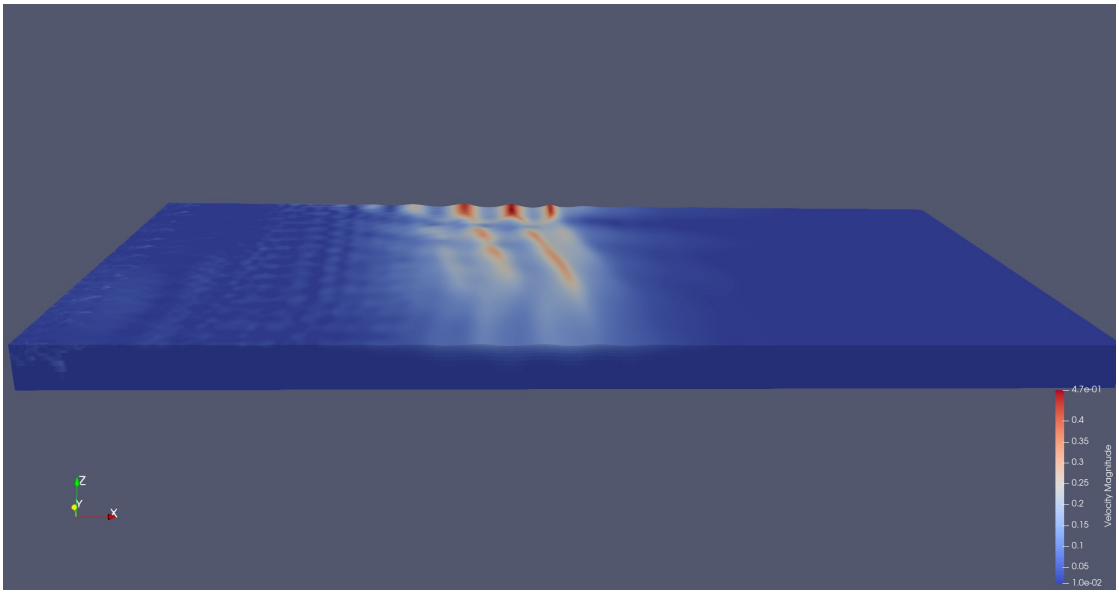

Figure 4.12: multilayer 3d simulation domain with 14 pistons. The dimensions of the domain are 7x14m

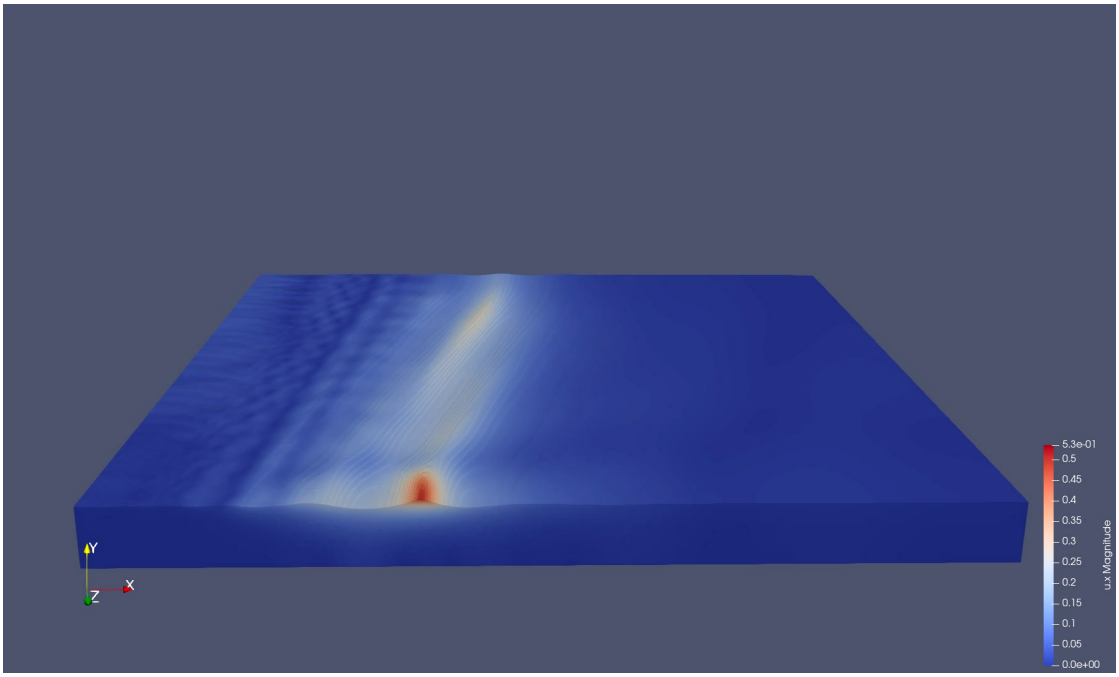

Figure 4.13: An image of the resulting wave field when using the boundary piston method in three dimensions with a 7mx7m domain. The movement of the pistons is delayed 0.1s between each of the 14 pistons.

Part III

Results and Conclusion

Chapter 5

Results

5.1 Surface Elevation

The easiest measurement to compare between the physical experiments and the basilisk simulations was the surface elevations, so that is where I started.

5.1.1 Wave Tank Experiment

I chose to use the results from fil3.dat for the majority of my Basilisk simulations, but also tested the results from using fil1.dat and fil2.dat.

Measurements from the four surface probes for different runs with the same piston amplitudes are shown in Figures 5.1, 5.2 and 5.3. When plotting the measured surface elevation from the same location in different runs, the surface elevation is most consistent between runs for the piston movements with the smallest amplitude.

The measured surface elevation from the probe at 10.5m is plotted in Figure 5.4 together with the analytical solution for 3 runs with different amplitudes. The wave steepness is calculated using the Matlab functions from Zhao and Liu 2022a. The difference in surface elevation of Zhao and Fenton's solution is very small for these waves Table 5.1 as expected from Zhao and Liu 2022a. Figure 5.5 is a plot of the surface elevation from the probe the closest to the piston. For the smaller waves run 1 and run 7, Table 3.4, the measurements coincide well with the theoretical solutions. For run 4, Table 3.4, with larger waves the measured waves have both sharper crests and flatter troughs than the analytical solution. It is also interesting to note how the wave shapes are closer to the shape of regular Stoke's waves as they have moved further away from the piston.

For the different steepnesses of the waves in my experiments, the difference in the theoretical amplitude between Fenton's solution and the the solution from Zhao and Liu 2022a is shown in 5.1. The difference in the amplitude is minor for these waves.

ak	Crest height difference, Zhao vs Fenton
0.18	-0.06%
0.31	-0.72%
0.09	0.00%

Table 5.1: The difference in surface height at the crest of Zhao's analytical solution compared to Fenton. The difference is small as expected from Zhao and Liu 2022a.

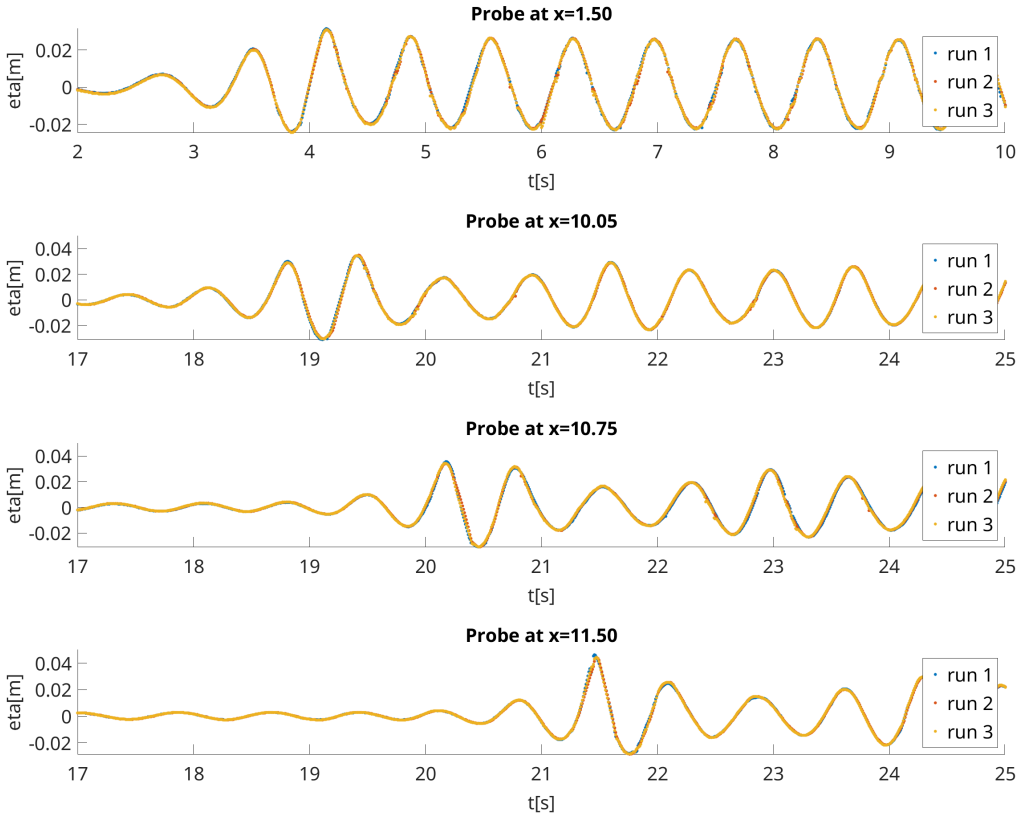

Figure 5.1: Plot of the measured surface elevation for the runs with the same piston parameters. The measurements from the same surface probes in the same plots. The measurements are from runs 1, 2 and 3 from 18/09/2024 Table 3.4. The measurements are very consistent between runs. However some difference can be spotted for the sensors furthest away from the piston, especially at crest of the wave with the highest amplitude

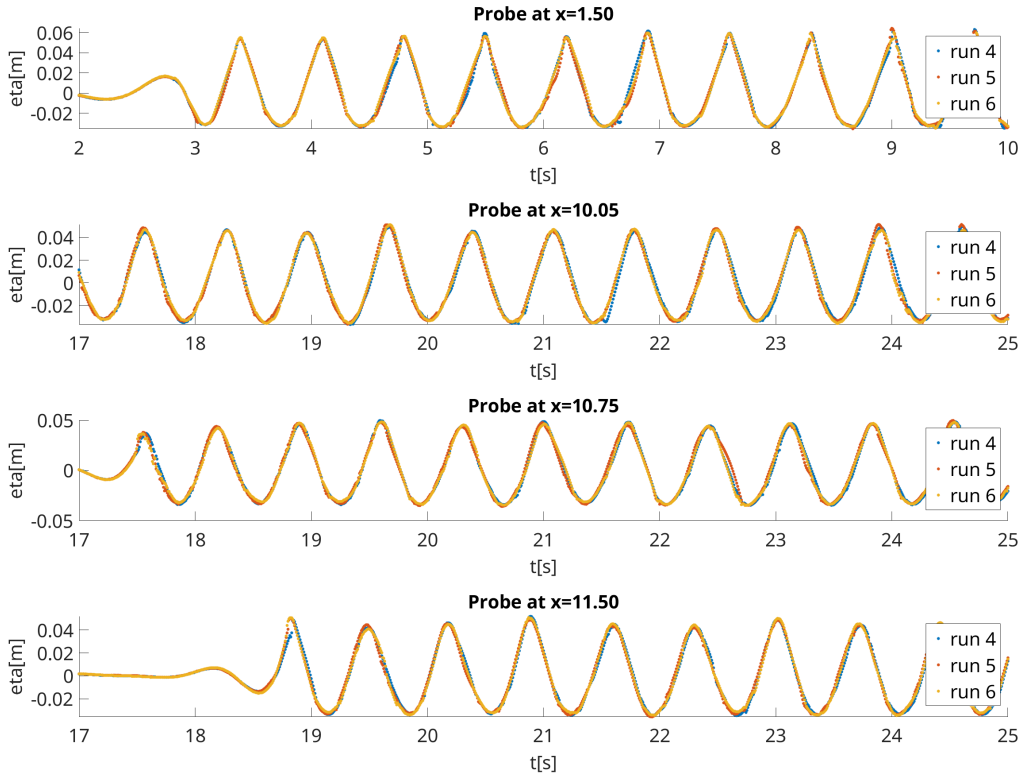

Figure 5.2: Plot of the measured surface elevation for the runs with the same piston parameters. The measurements from the same surface probes in the same plots. The measurements are from runs 4, 5 and 6 from 18/09/2024 Table 3.4. The measurements are not as consistent as the measurements from the runs with half of this piston amplitude in Figure 5.1. The differences between runs are also apparent closer to the piston than for the runs with smaller amplitudes.

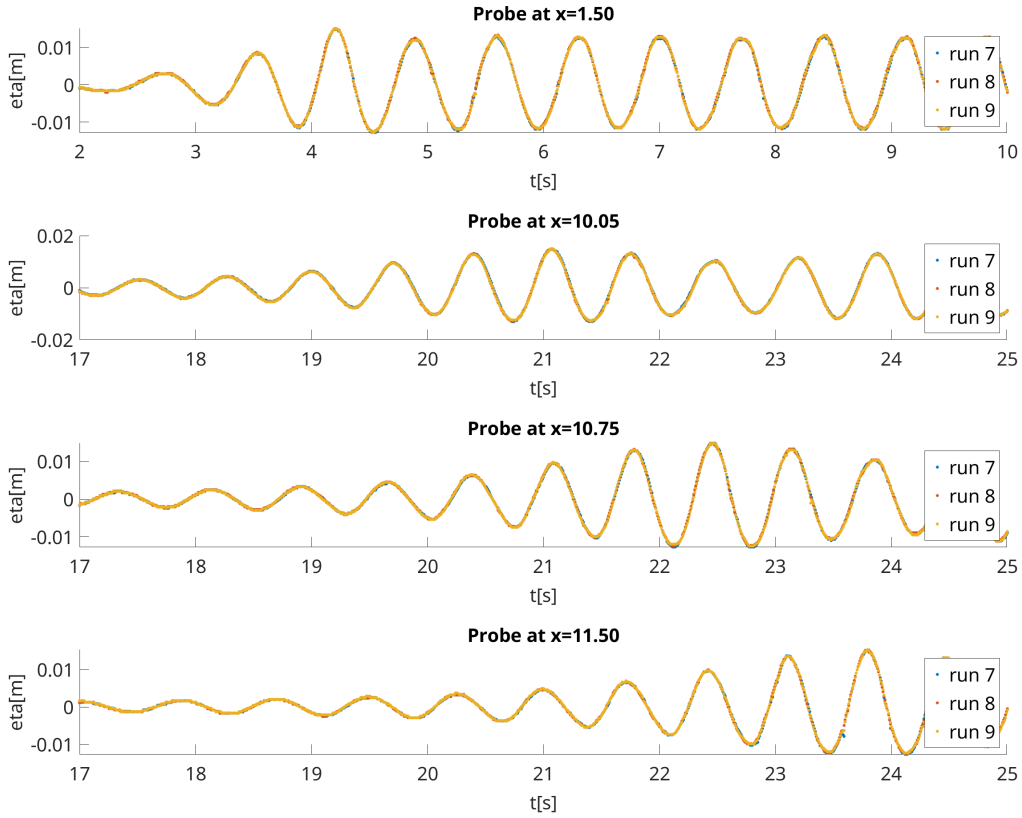

Figure 5.3: Plot of the measured surface elevation for the runs with the same piston parameters. The measurements from the same surface probes in the same plots. The measurements are from runs 7, 8 and 9 from 18/09/2024 Table 3.4. These measurements of the piston movement with the smallest amplitude are more consistent between runs compared to both double and quadruple the piston amplitude in Figures 5.1 and 5.2.

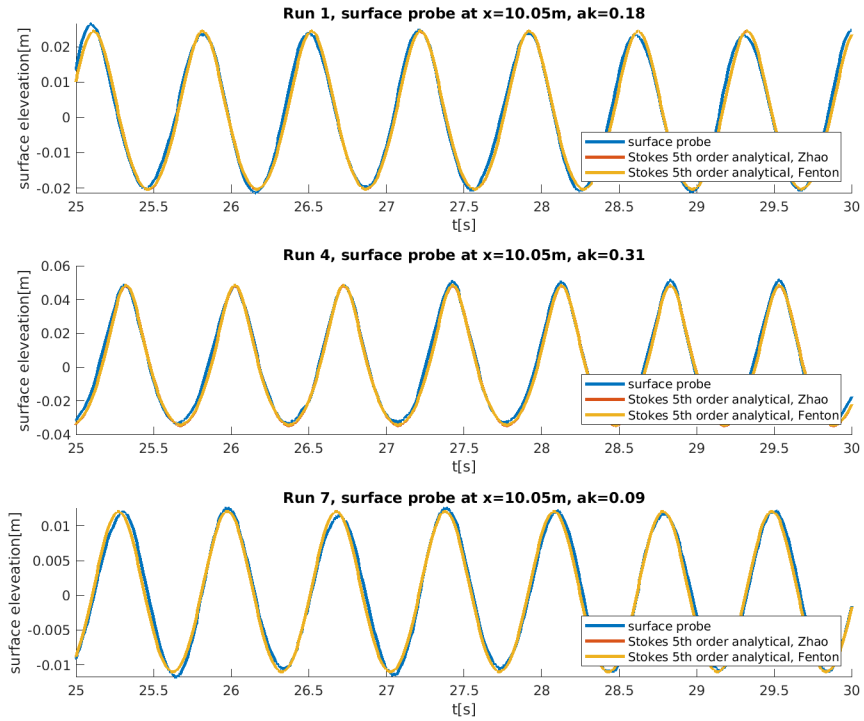

Figure 5.4: Measured surface and analytical surface calculated using Zhao and Liu 2022a method for Fenton's solutions and their own solution. The results are from 18/09/2024. The measured surface elevation is generally the same shape as the analytical solutions.

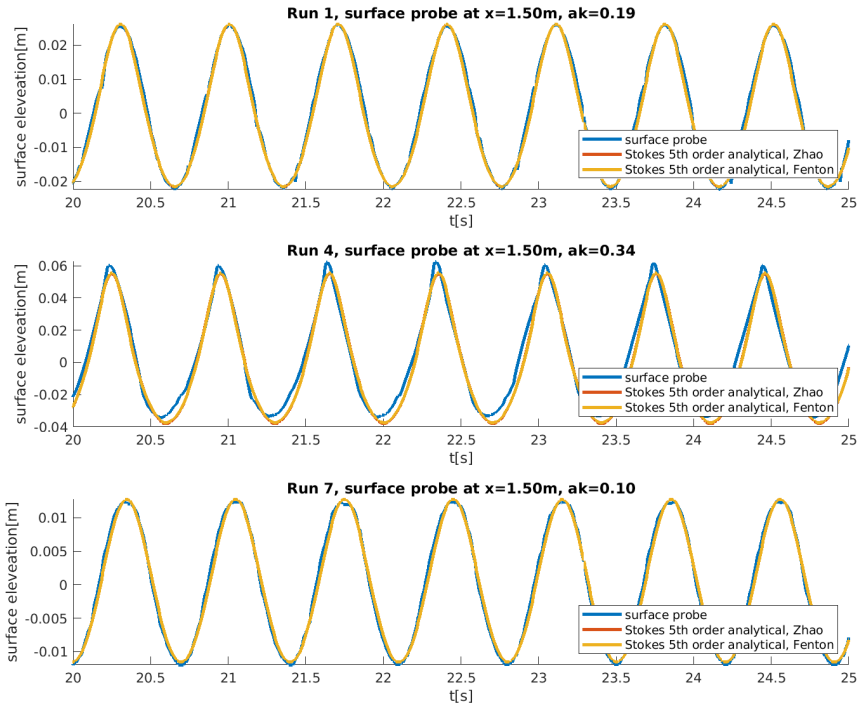

Figure 5.5: Measured surface and analytical surface calculated using Zhao and Liu 2022a method for Fenton's solutions and their own solution. The results are from 18/09/2024. For runs 1 and 7 with the lowest amplitudes the the surface elevation is already close in shape to the analytical solutions at 1.5 m from the piston. For run 4, with the largest amplitude in the piston movement, the measured waves are both sharper at the crests and flatter in the troughs than the analytical solution for Stokes waves with the same parameters.

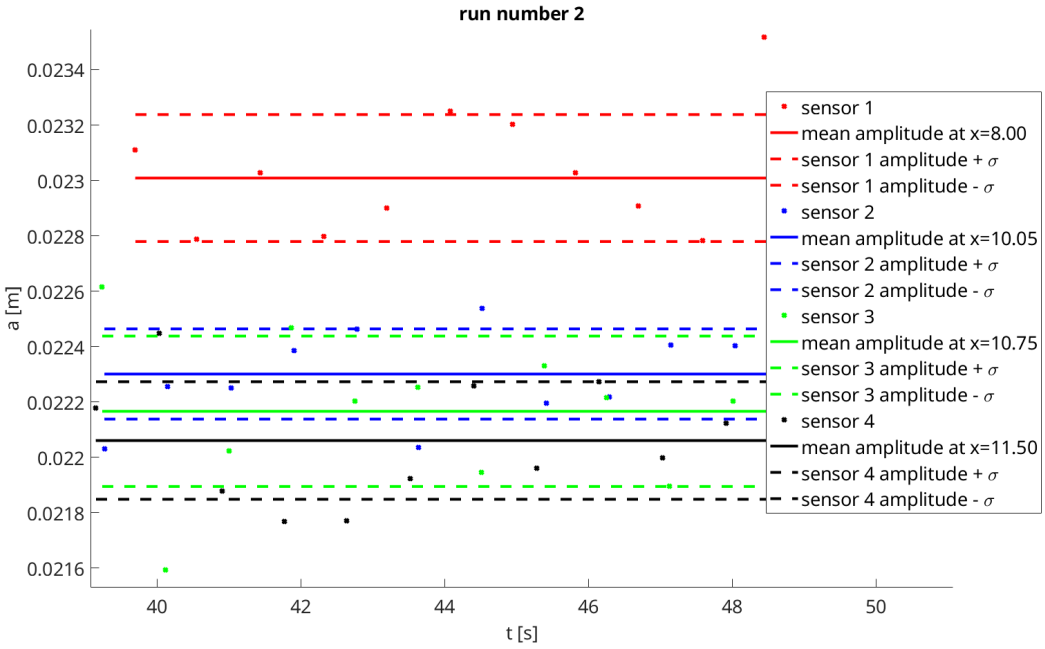

Figure 5.6: The amplitude of the waves as they pass by the sensors for run number 2 from 2/04/2024.

I also made a MATLAB script that calculates the amplitudes of the waves as they pass the probes measuring the surface elevation.

The results shown in Figure 5.7, show that the measured amplitude decreases further away from the piston wave maker. For the measured amplitudes from the steeper waves Figure 5.8 and 5.9, the amplitude also decreases as a general trend. However the measured amplitude increases between the fourth and fifth sensor for some of the runs. This could be due to non-linear effects. For the experiments done on 18/09/24 for runs 4 and 5 some of the sensors measuring the surface elevation did not work correctly and yielded very noisy data so these results have been omitted from these plots.

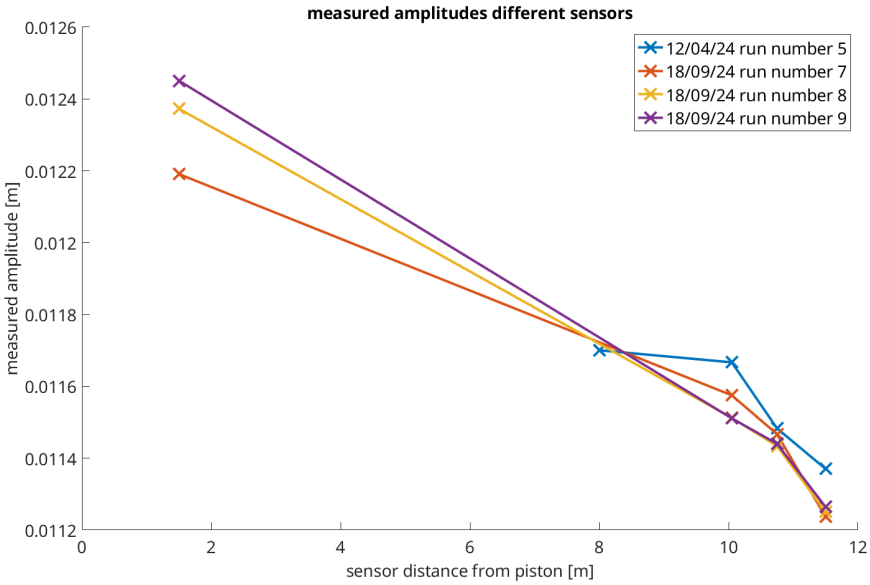

Figure 5.7: Plot of the amplitudes calculated from the surface elevation probes. For the runs with ak=0.09. The reduction in amplitude is small $\approx 1mm$ between the measurements at 1.5m and 11.5m from the piston.

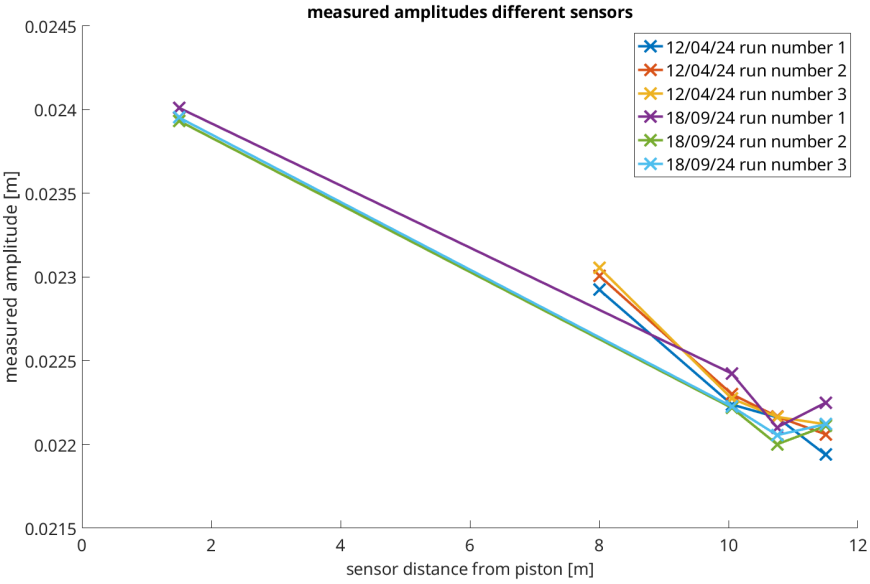

Figure 5.8: Plot of the amplitudes calculated from the surface elevation probes. For the runs with ak=0.18. The reduction in amplitude is $\approx 2mm$ between the sensors at 1.5m and 11.5m. This is roughly the same relative decay in amplitude as seen in Figure 5.7.

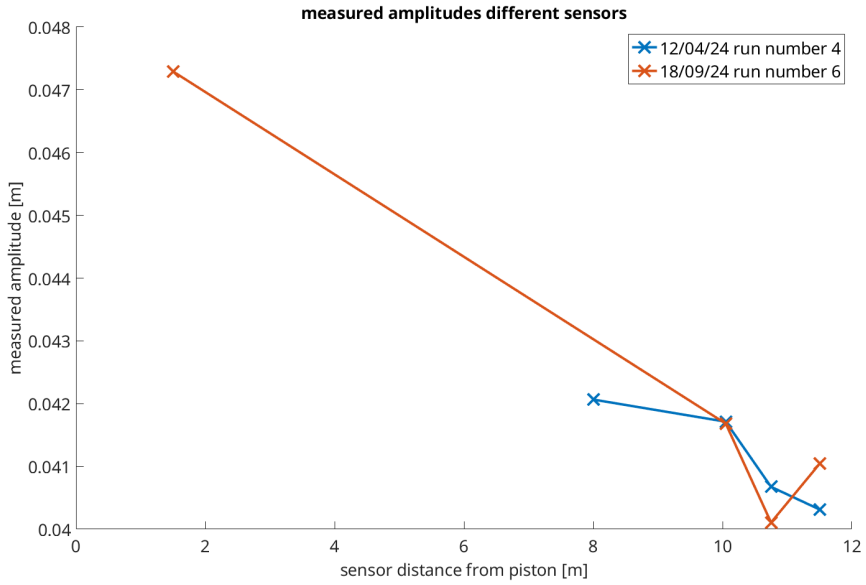

Figure 5.9: Plot of the amplitudes calculated from the surface elevation probes. For the runs with ak=0.31. The reduction in amplitude is $\approx 7mm$ between the first and last probe. This is a larger relative decay in amplitude than what is seen for the smaller waves in Figure 5.7 and Figure 5.8.

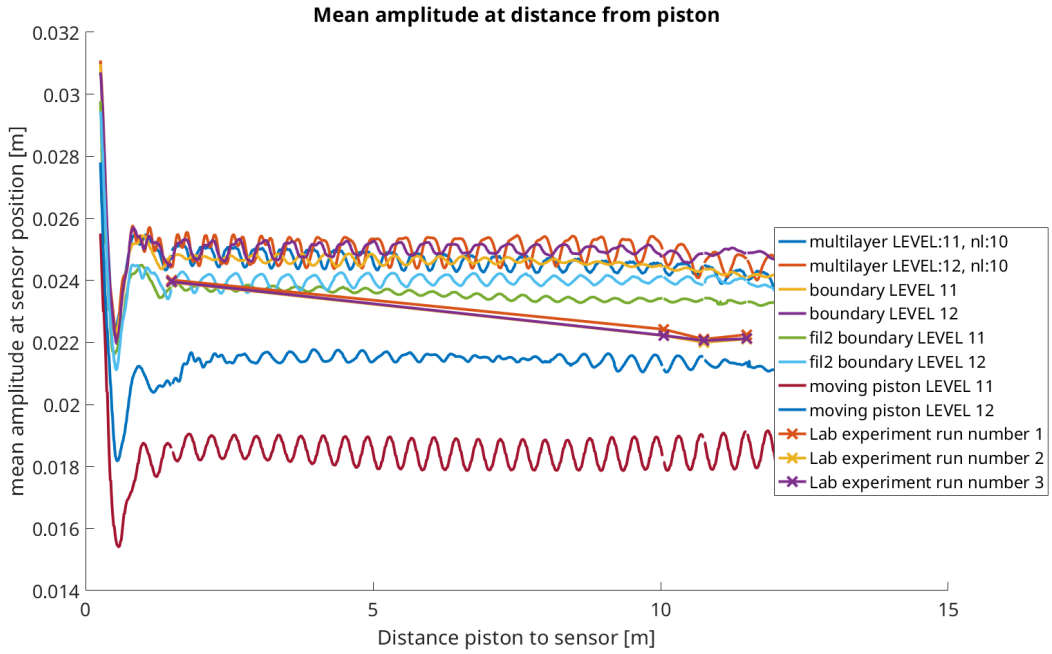

Figure 5.10: Plot of the mean amplitude at a fixed distance from the piston. The piston movement is from run 1 from Table 3.2. The measurements start a few seconds after the start of the wave train has passed the position. None of simulation the methods have the same decay in amplitude as the physical experiments. The multilayer method and the method of setting the veloctiy on the boundary show the best correspondence with the physical experiments at the sensor closest to the piston. But since the amplitude decays in the physical experiments they end up closer to the amplitudes yielded by the moving piston method at the measurements further away from the piston.

5.1.2 2D Basilisk simulations

I used the piston positions from fil3.dat for my simulations. I smoothed the data by using a FFT and setting all the frequencies above a cut-off frequency, 8Hz, to 0 before applying a reverse FFT. I then took the derivative of the smoothed data to get the piston speed used for the boundary conditions and for setting the water velocity when using Stephane's trick. I used the piston positions from 12/04/24 when the sensor for fil3.dat was working for all my basilisk simulations.

Navier-Stokes solver - Stephane's trick

The results for the amplitudes of the waves as they pass by a position a fixed distance from the piston are shown in Figures 5.10, 5.11 and 5.12. The results for the moving piston, which is using the piston trick, are lower than the measured amplitudes from the experiments. This is a result of the piston being leaky. Increasing the refinement level improves the solution. In Figure 5.13, Figure 5.14 and Figure 5.15.

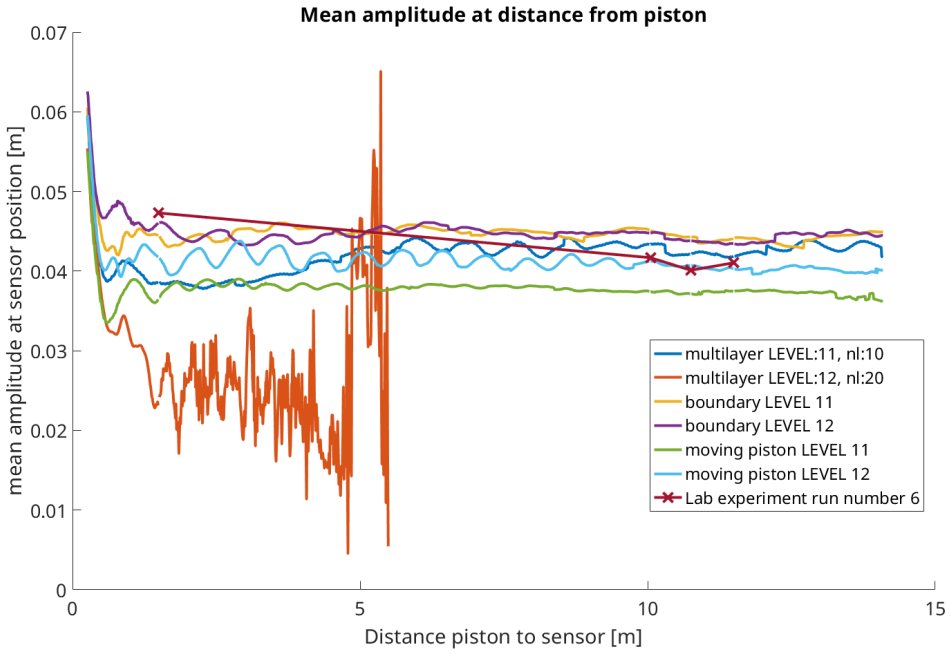

Figure 5.11: Plot of the mean amplitude at a fixed distance from the piston. The piston movement is from run 4 from Table 3.2. The measurements start a few seconds after the start of the wave train has passed the position. For these waves that are close to breaking the multilayer solution is not stable without further adjustments when increasing the mesh refinement level. For the other Basilisk simulations the amplitude does not decay as it is measured further away from the piston. This leads to the methods where the velocity is set on the boundary are closest to the right amplitude closest to the piston, while further away the leaky moving piston is closer to the right amplitude since it yield lower amplitudes.

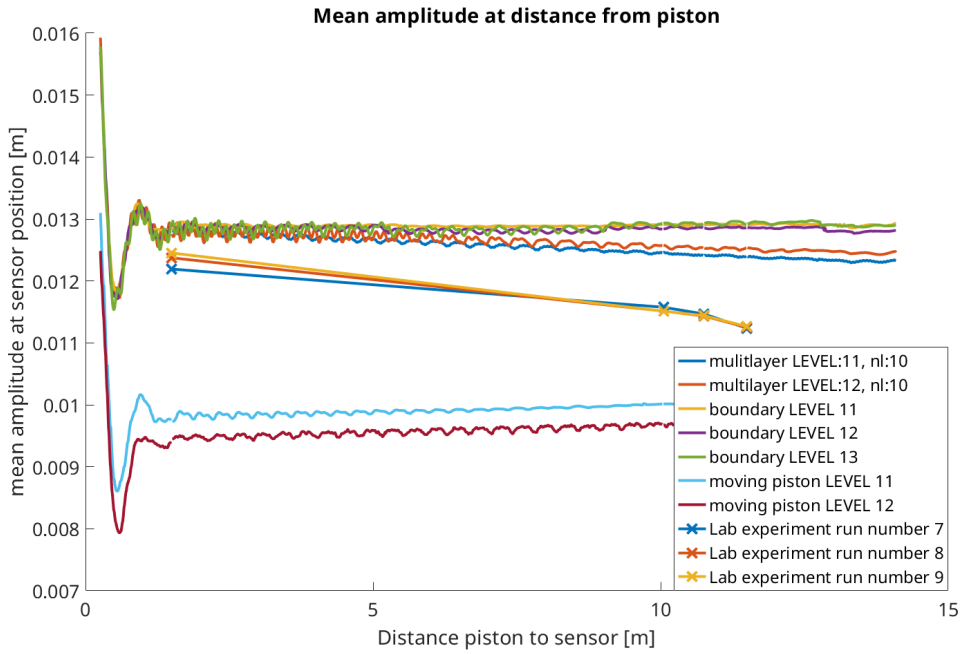

Figure 5.12: Plot of the mean amplitude at a fixed distance from the piston. The piston movement is from run 5 from Table 3.2. The measurements start a few seconds after the start of the wave train has passed the position. When setting the velocity on the boundary with the multilayer solver and the NS solver the amplitudes are a little high at 1.5m and do not decay in the same way as the measured amlitude from the physical experiments. The measured amplitudes using the moving piston method are lower than the amplitudes from the physical experiments. This is as excted since the moving piston is leaky.

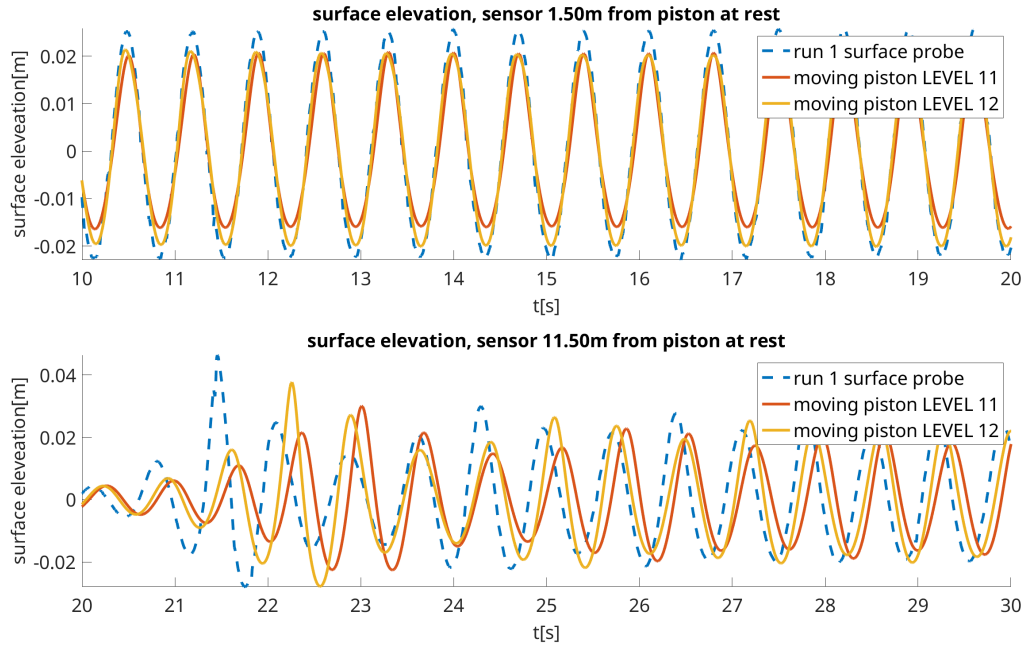

Figure 5.13: Plot of the surface elevation from physical experiments and Basilisk simulations using the moving piston method. The amplitude is lower in the Basilisk simulations as can also be seen in Figure 5.10. The lower amplitude results in a difference in phase as the waves travel further away from the piston.

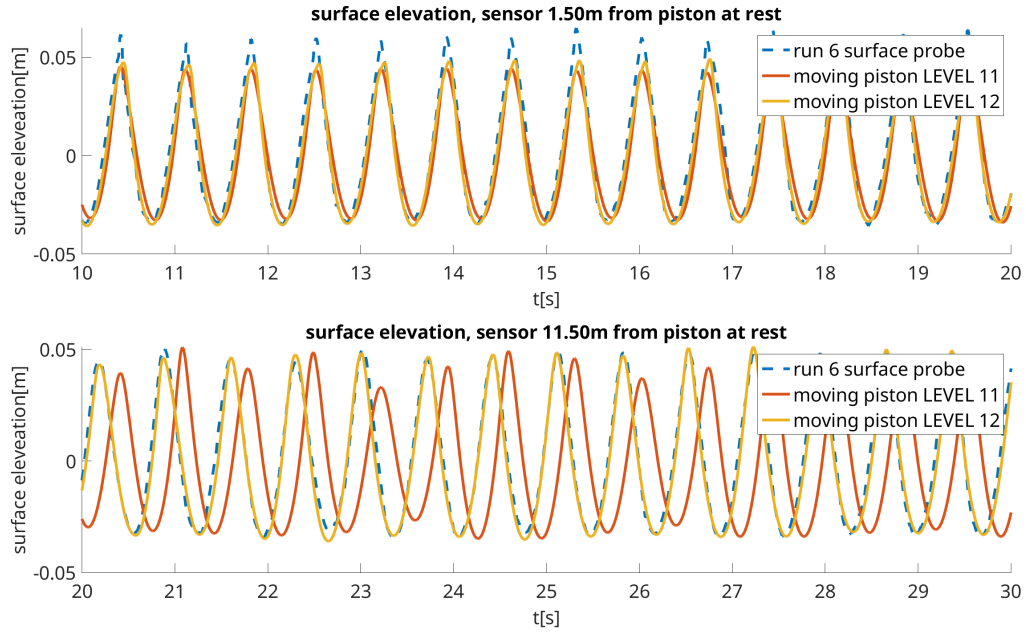

Figure 5.14: Plot of the surface elevation from physical experiments and Basilisk simulations using the moving piston method. At the sensor closest to the piston the amplitudes are smaller for the Basilisk simulations compared to the physical experiments. The amplitudes are closer at the sensor 11.5m away from the piston as can also be seen in Figure 5.11. The phase shift in the waves is relatively large for the LEVEL 11 simulation while the simulation using a finer mesh is close in phase.

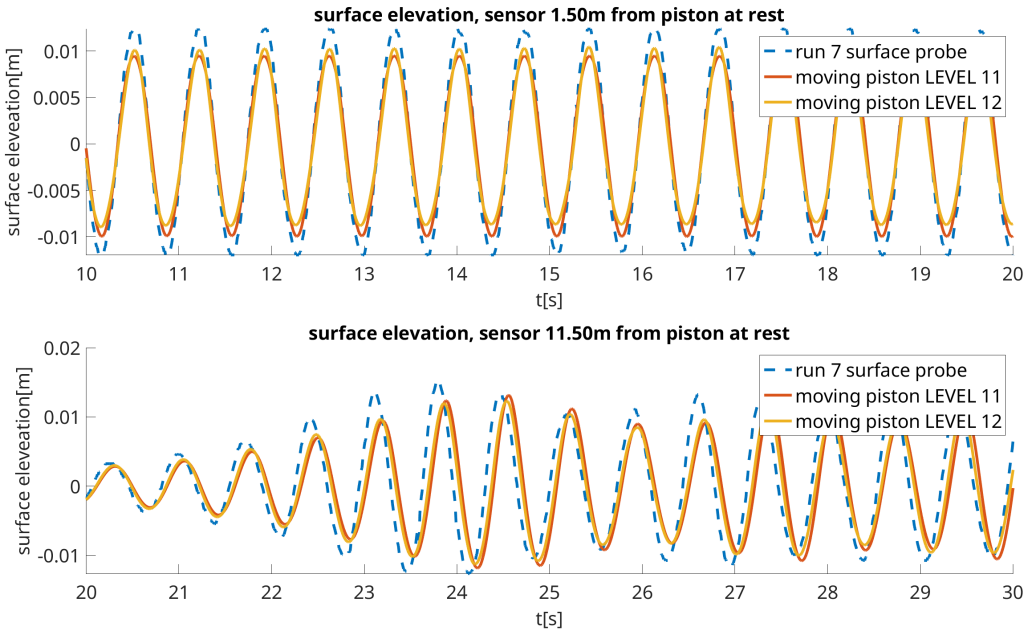

Figure 5.15: Plot of the surface elevation from physical experiments and Basilisk simulations using the moving piston method. At the sensor closest to the piston the amplitudes are smaller for the Basilisk simulations compared to the physical experiments. The amplitude is smaller for both simulations at the furthest sensor as well. This can also be seen in Figure 5.12. The phase is also shifted at the furthest sensor.

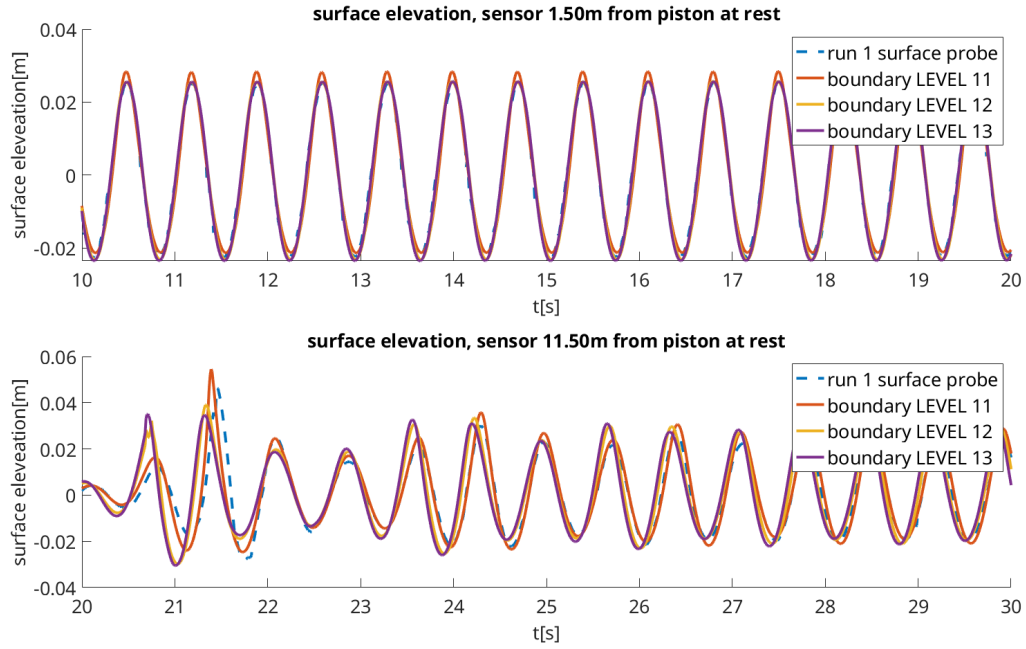

Figure 5.16: Plot of the surface elevation from the physical experiments and Basilisk simulations setting the velocity on the boundary. The amplitude is slightly higher in the simulations at both measurement locations as can also be seen in Figure 5.10. There is some breaking evident at the sensor at 11.5m when the first waves with the highest amplitudes arrive. The phase is much closer to the experiments at 11.5m than the results from the moving piston method.

Navier-Stokes solver - Setting velocity on boundary

The calculated amplitudes from setting the velocity on the boundary in the Navier-Stokes solver do not decay as they do in the experiments. For the smaller amplitude experiments Figure 5.12 the method produces waves that are too large. As the piston amplitude in the experiments increases, the basilisk simulations result in waves that are closer to the physical experiments and eventually smaller than the results from the physical experiments, Figure 5.11.

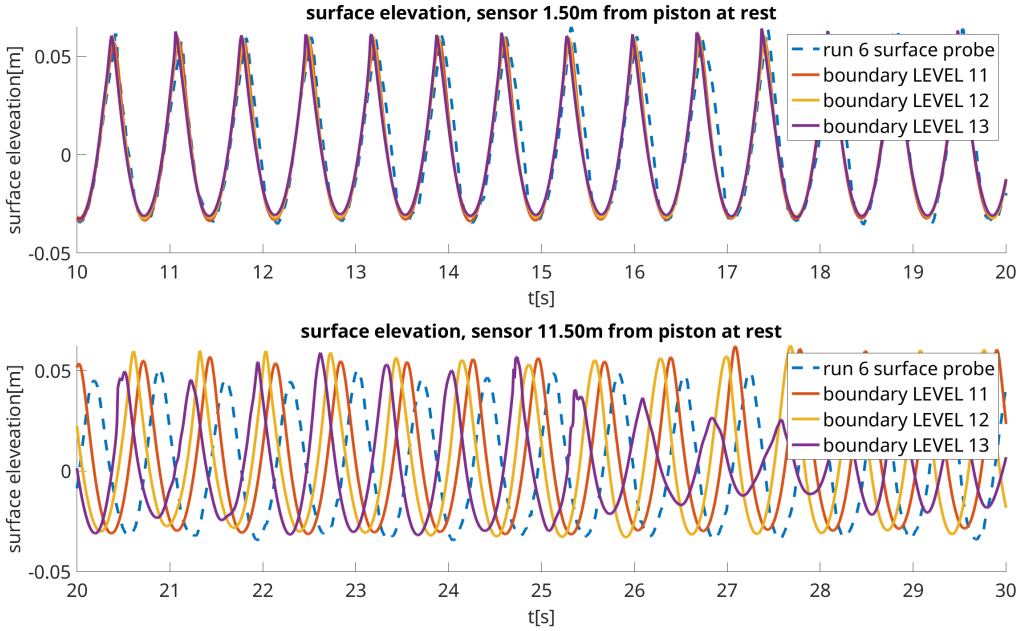

Figure 5.17: Plot of the surface elevation from physical experiments and Basilisk simulations setting the velocity on the boundary. The amplitude is close at the first sensor as can also be seen from Figure 5.9. However there is a slight difference in phase already at the first sensor which is much larger at the sensor at 11.5m. Much of this difference seems to be the result of the first steep waves generated breaking in the Basilisk simulation and not in the physical experiments.

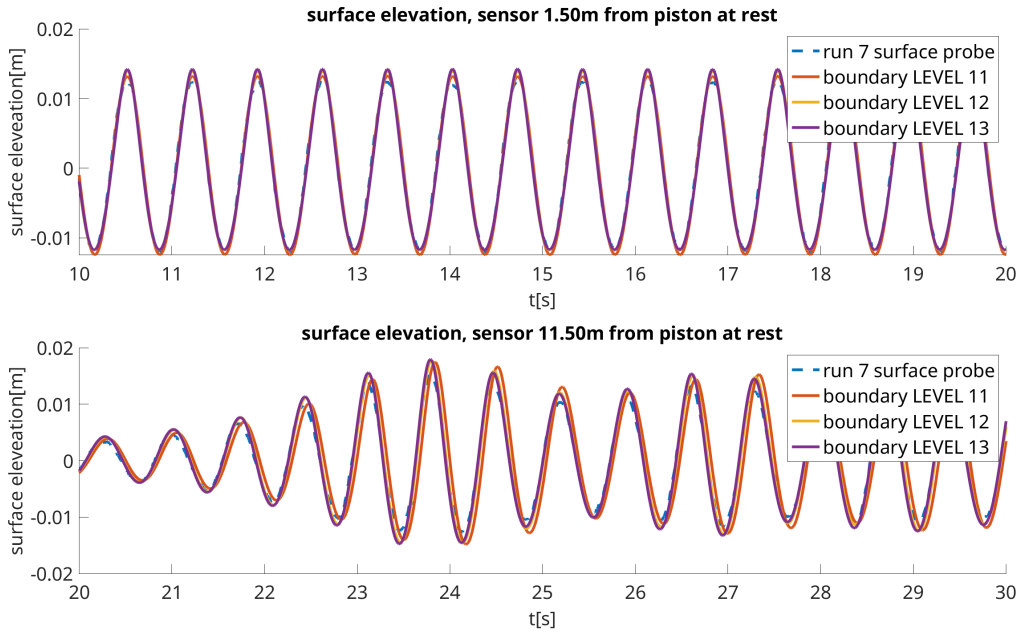

Figure 5.18: Plot of the surface elevation from physical experiments and Basilisk simulations setting the velocity on the boundary. For these smaller waves the amplitudes in the Basilisk simulations is slightly larger than the experimental results at both locations. The resulting phase shift is not as large however as for the steeper waves.

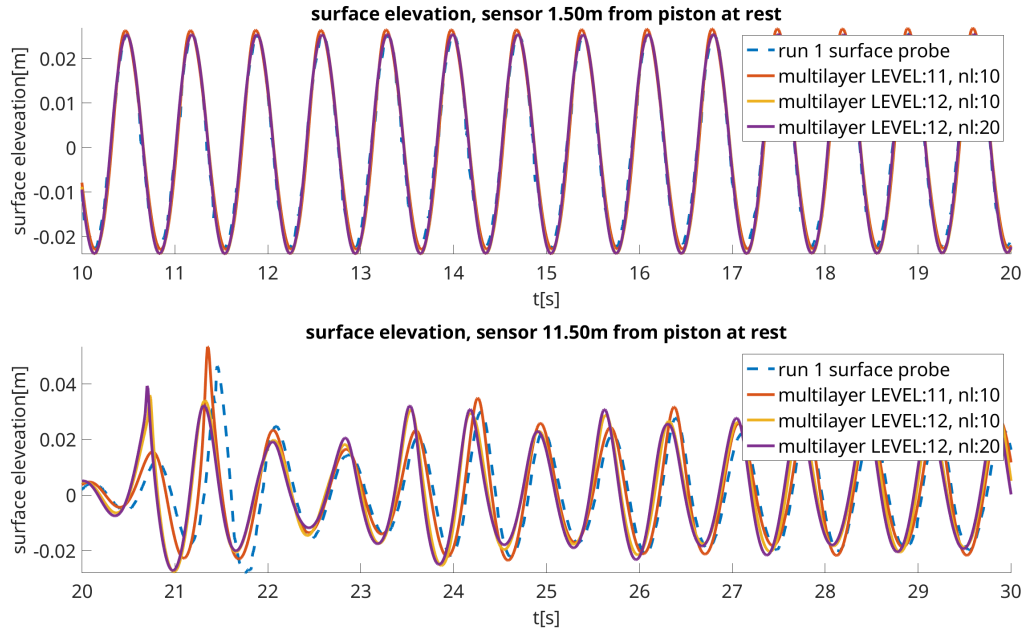

Figure 5.19: Plot of the surface elevation from physical experiment and Basilisk simulations with the multilayer solver. There is a slight change in phase between the physical experiment and the Basilisk simulation between the two probe locations at 1.5m and 11.5m. The change is phase is very similar to the change in phase when setting the velocity on the boundary using the NS solver,Figure 5.16. This is as expected since the amplitudes are similar, as seen in Figure 5.10. The largest difference is in the start of the wave train where the waves are closest to breaking.

a [m]	ak	u difference at the crest, Zhao vs Fenton
0.0225	0.18	1.77%
0.0410	0.31	6.39%
0.0116	0.09	0.46%

Table 5.2: The difference in horizontal velocity at the crest of Zhao's analytical solution compared to Fenton Zhao and Liu 2022a. For the steepest waves the difference is significant.Water depth h=0.6m, frequency=1.425Hz

Multilayer solver

The resulting amplitudes from simulations using the Basilisk multilayer solver are very close to the results achieved with the Navier Stokes solver. The simulation speed of the multilayer solver is in some cases significantly faster than the Navier Stokes solver. Simulation times for comparable mesh refinement can be seen in Figure 5.34 and Figure 5.33.

5.2 Wave Kinematics

The solution for Stokes waves from Zhao and Liu 2022a, gives a larger velocity at the crest. For waves of the approximate size as the ones I considered, the difference is much larger for the steepest waves compared to the smaller waves. The results for three different wave steepnesses can be seen in Table 5.2.

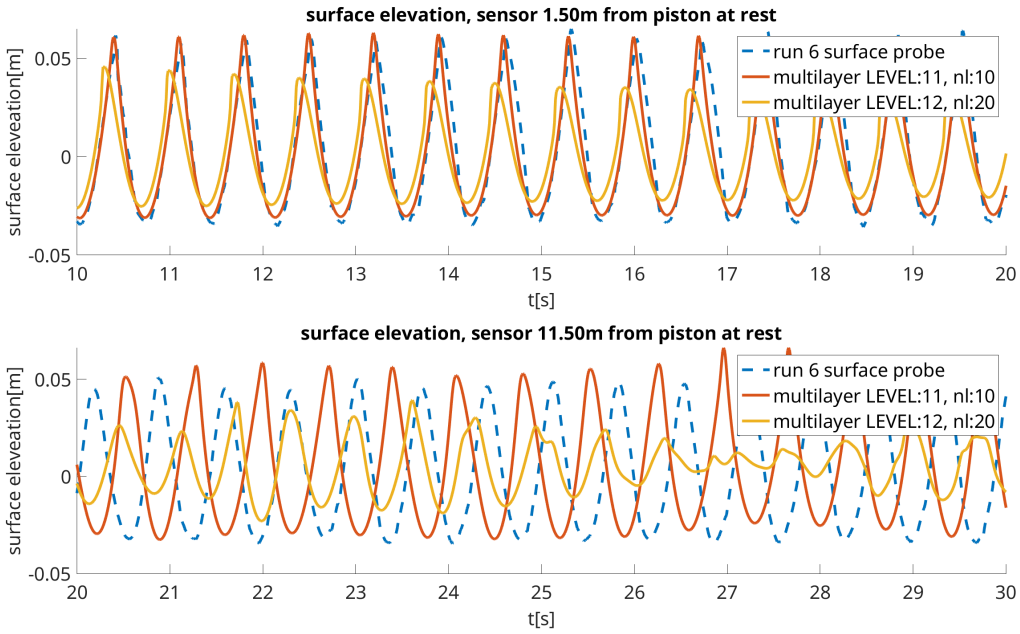

Figure 5.20: Plot of the surface elevation from physical experiment and Basilisk simulations with the multilayer solver. The multilayer solver struggles with the steeper waves that are close to breaking. Increasing the mesh resolution from LEVEL 11 to LEVEL 12 causes the waves exceed the breaking parameter at the crests faster. This causes the solution on the coarser mesh to be more stable as when the waves break in the multilayer solver a lot of high frequency noise is generated.

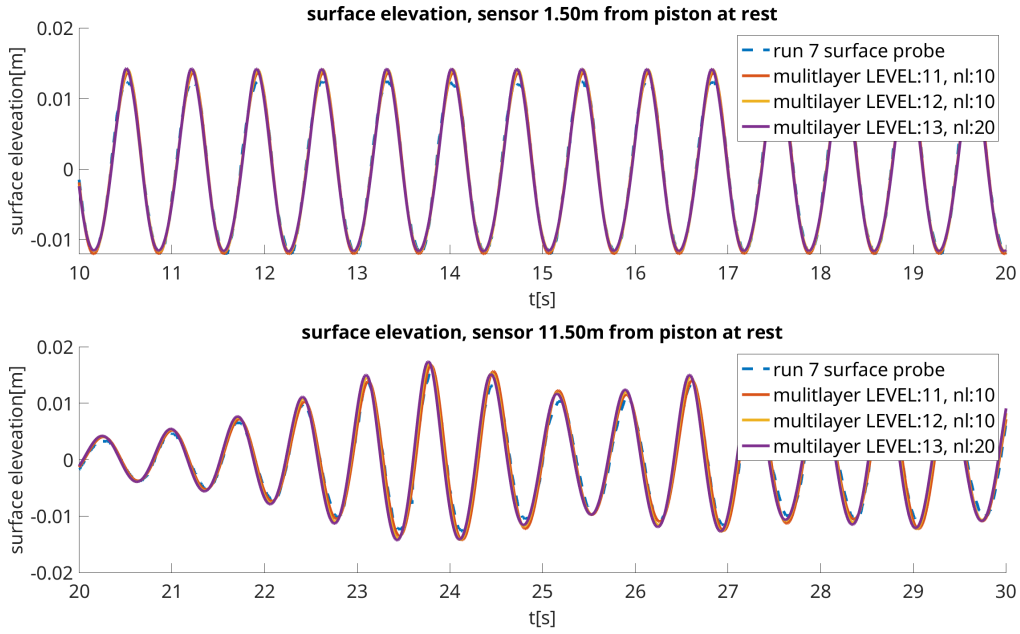

Figure 5.21: Plot of the surface elevation from physical experiment and Basilisk simulations with the multilayer solver. For the smallest waves the multilayer solver performs very well the amplitude is a little to large at both the measuring probes but the change in phase is small.

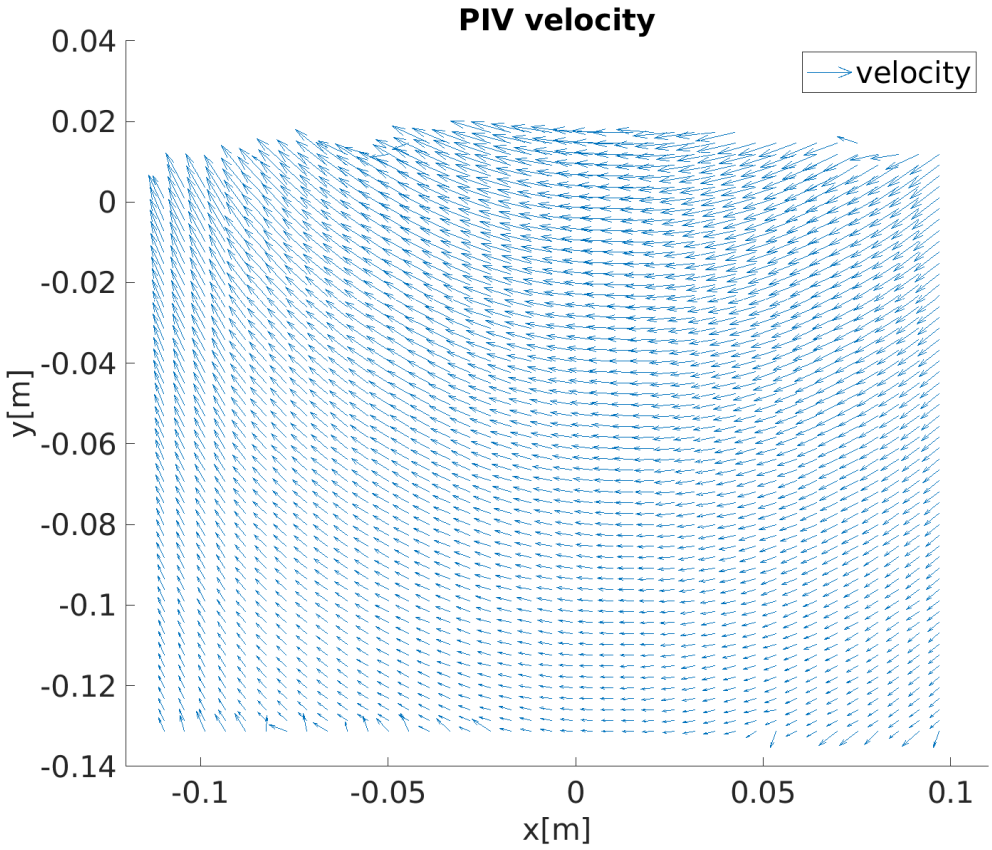

Figure 5.22: The resulting velocity field after performing PIV on the images from Jensen et al. 2001

5.2.1 PIV

I used the images from Jensen et al. 2001, Figure 2.5, for calculating the velocity field in the water. I performed two PIV passes on the images with an initial window size of 64x64 pixels. The second pass then uses the displacement from the first pass as an initial guess with window size 32x32 pixels and outlier replacement. The resulting velocity field is shown as a quiver plot in Figure 5.22.

A plot of the horizontal velocity under the crest is shown in Figure 5.23. The crest is found by searching for the horizontal position in Figure 5.22 with the smallest vertical velocity. The standard deviation in the amplitude in the plot in Figure 5.23 is estimated by calculating the amplitude from the sensor at 11.5 m from the piston and the standard deviation of those measurements over a period of 30s. This estimation is probably much higher than the true uncertainty in the measurement of the amplitude of the wave in Jensen et al. 2001.

5.2.2 Basilisk

Navier-Stokes solver moving piston

To create the same plots for the basilisk results as for the PIV results in Figure 5.23, I found the crest closest to the PIV location of 12.5 m from the piston at 40 seconds after the start of the piston movement. I used the wave amplitude from the basilisk results to

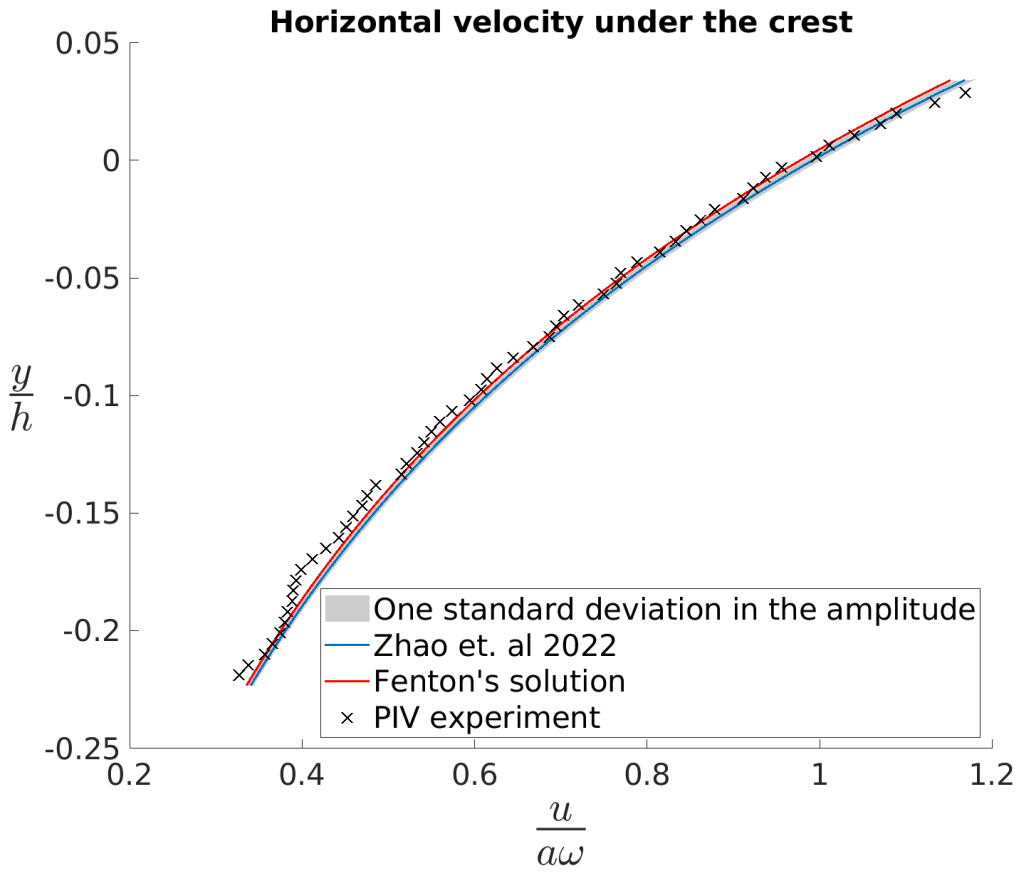

Figure 5.23: Horizontal velocity under the crest from PIV of images from Jensen et al. 2001. The horizontal velocity u is scaled by the amplitude and frequncy of the wave. The standard deviation in the amplitude is based on my measurements of the amplitudes of many passing waves from 24/04/2024, which is probably much higher than the real uncertainty in the measurement from Jensen et al. 2001.

find an analytical solution which I then compared to the Basilisk results. For Stephane's trick with a moving piston, the results for runs 1, 4, and 5 from Table 3.2, are shown in Figures 5.24, 5.25 and 5.26. From Figure 5.24

5.2.3 Navier-Stokes Solver Boundary Piston

I repeated the same procedure as with the moving piston for the results from the boundary piston. The resulting velocity profiles for runs 1, 4, and 5 from Table 3.2 are shown in Figures 5.27, 5.28 and 5.29. I also included results for run 1 when not including the conserving.h method Stephane Popinet 2025d from Basilisk. Not using conserving.h resulted in simulation results deviating more from the theoretical and experimental results.

Multilayer solver

The horizontal velocity under the crest closest to 12.5m at 40 s after the start of the piston movement is shown in Figures 5.30, 5.31 and 5.32. For runs 1, 4 and 5 respectively, from Table 3.2.

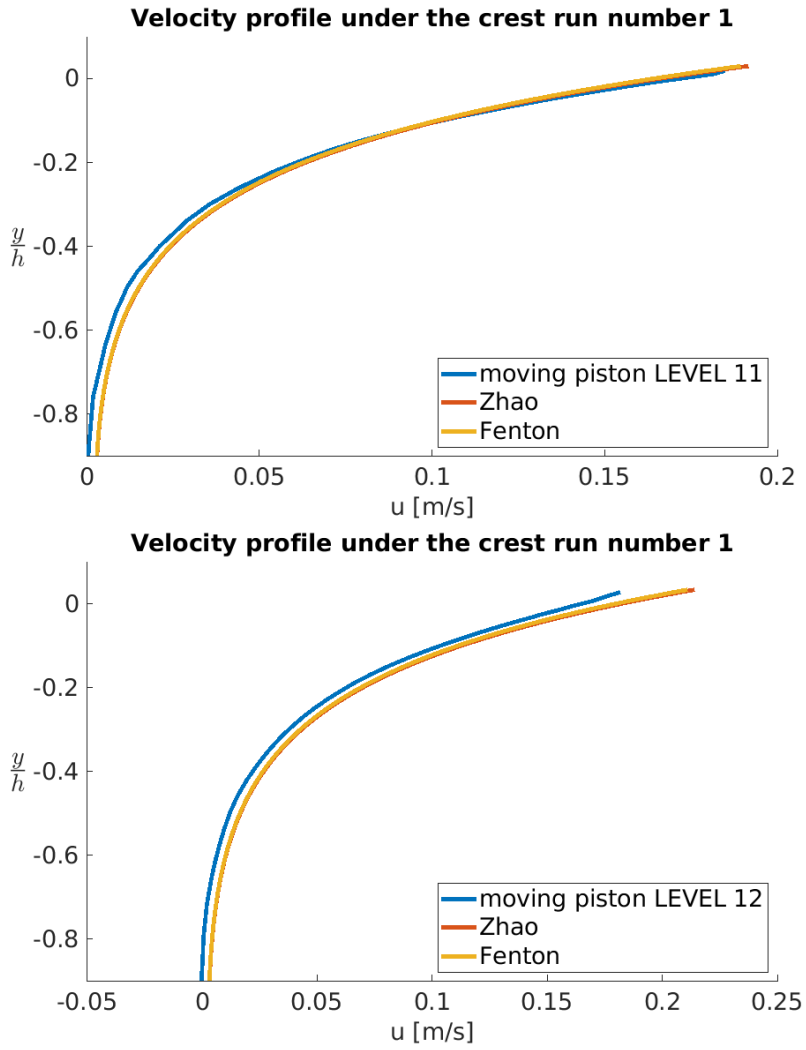

Figure 5.24: Horizontal velocity under the crest. The velocity is taken under the crest closest to 12.5m from the piston at 40s from the start of the piston movement. Velocity field generated using the moving piston method. Piston movement from run 1 from Table 3.2. The amplitude for the analytical solution is calculated using Zhao and Liu 2022a and measured amplitude from the simulation. The velocity profile does not quite follow the profile of Stokes waves for the higher refinement level simulation but fits better

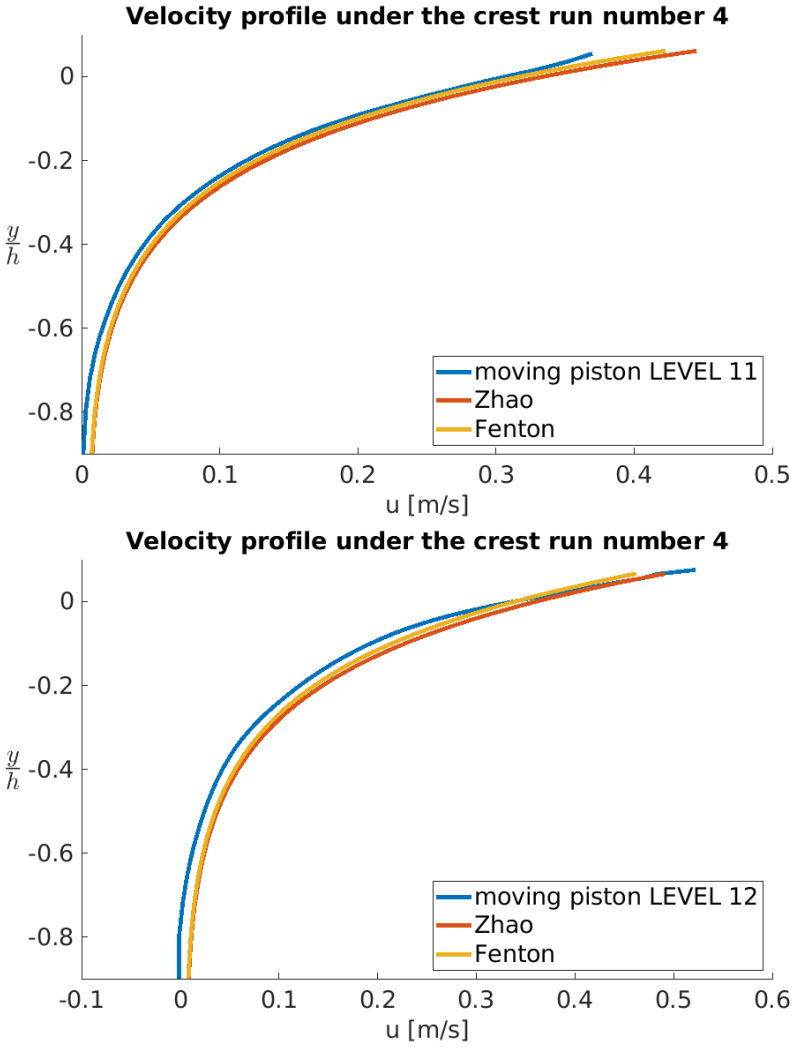

Figure 5.25: Horizontal velocity under the crest. The velocity is taken under the crest closest to 12.5m from the piston at 40s from the start of the piston movement. Velocity field generated using the moving piston method. Piston movement from run 4 from Table 3.2

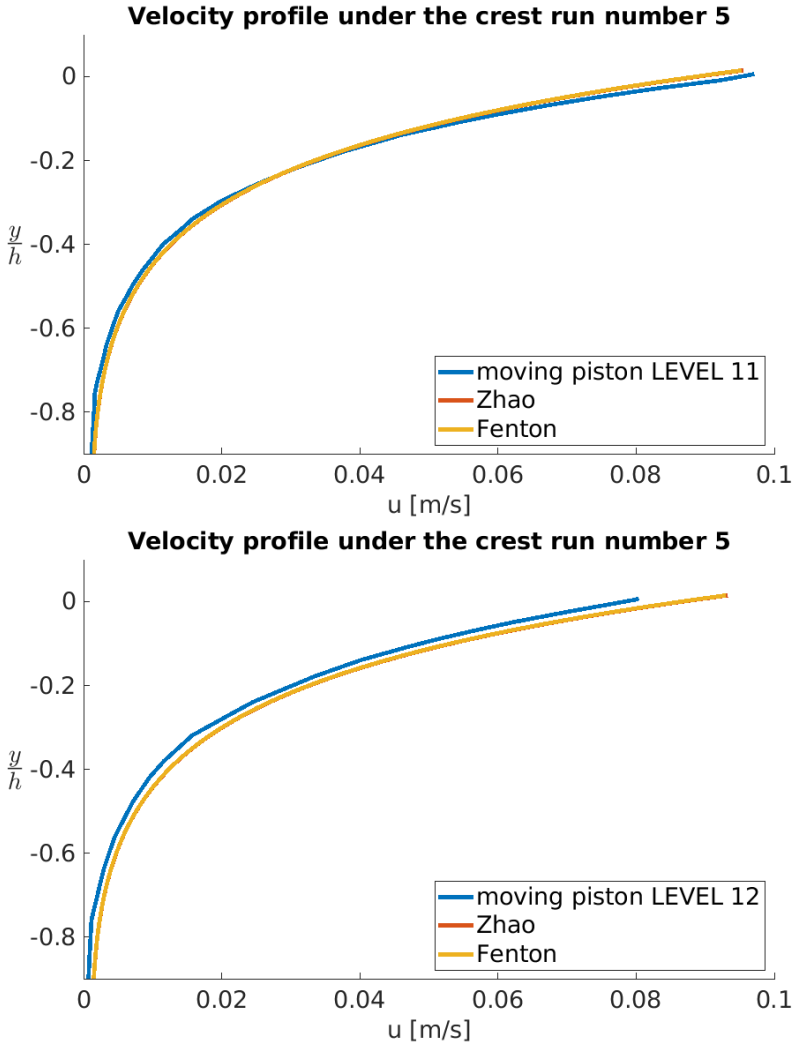

Figure 5.26: Horizontal velocity under the crest. The velocity is taken under the crest closest to 12.5m from the piston at 40s from the start of the piston movement. Velocity field generated using the moving piston method. Piston movement from run 5 from Table 3.2

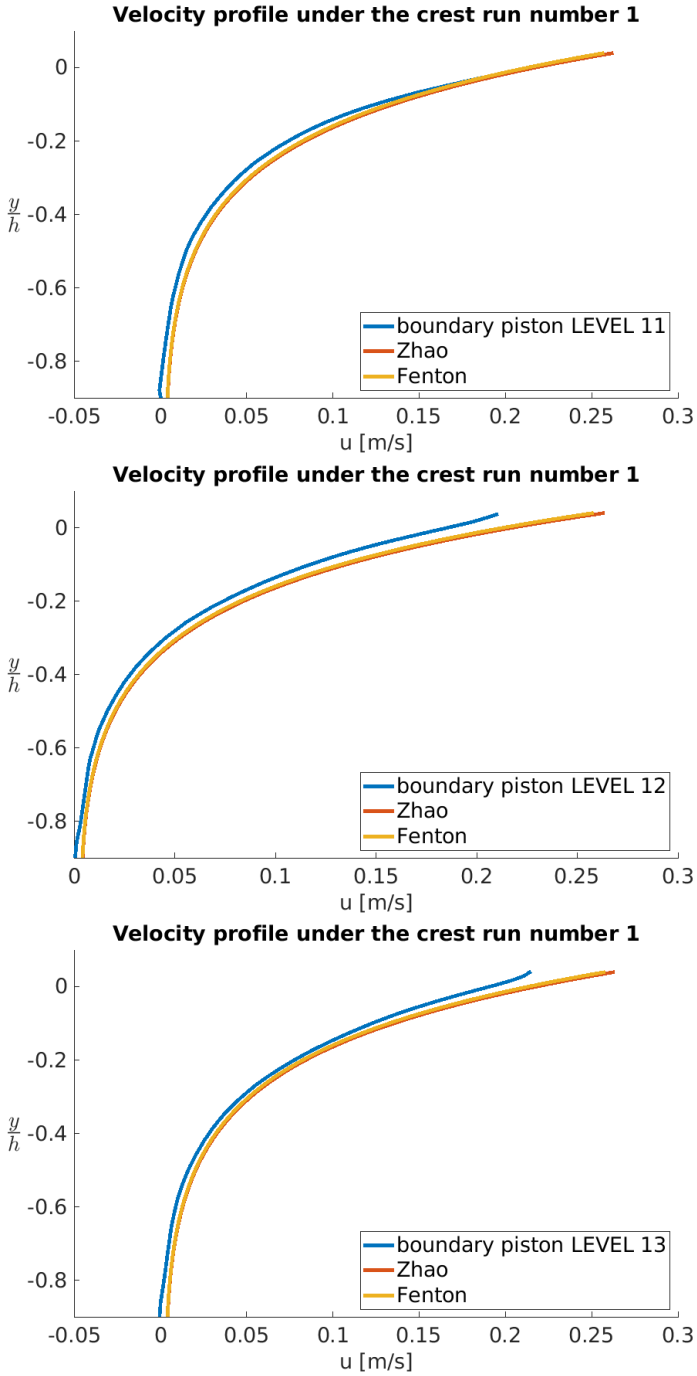

Figure 5.27: Horizontal velocity under the crest. The velocity is taken under the crest closest to 12.5m from the piston at 40s from the start of the piston movement. Velocity field generated by setting the velocity at the boundary and using the NS solver. Piston movement from run 1 from Table 3.2

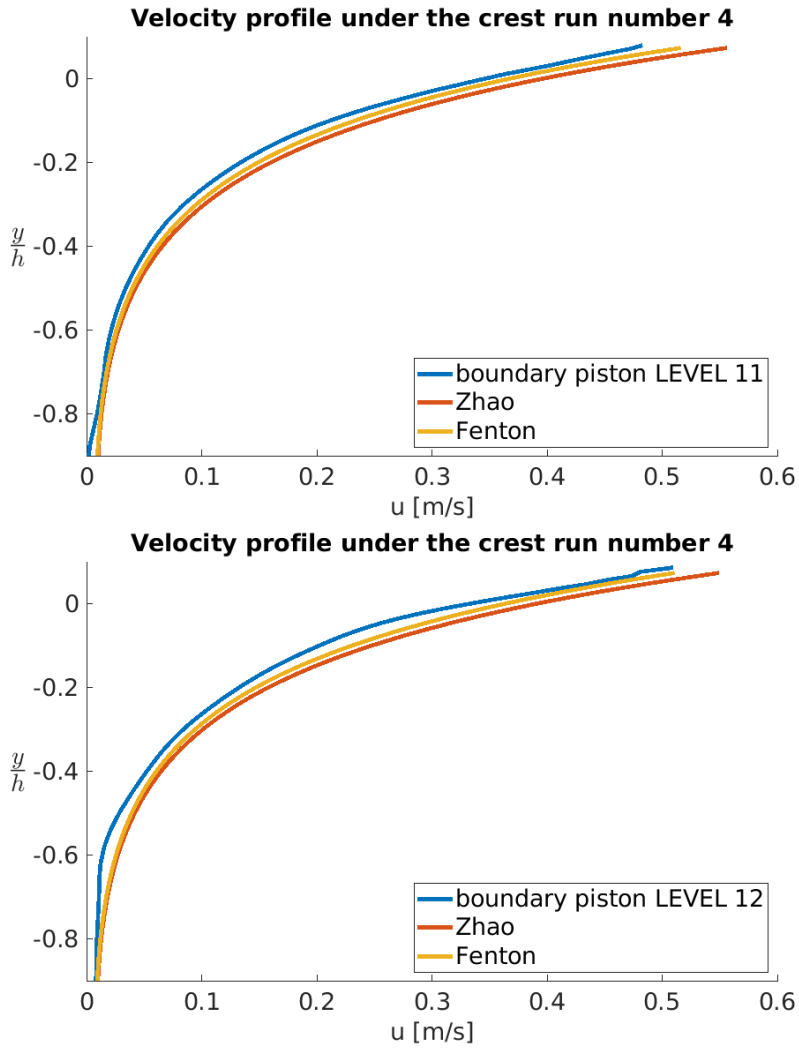

Figure 5.28: Horizontal velocity under the crest. The velocity is taken under the crest closest to 12.5m from the piston at 40s from the start of the piston movement. Velocity field generated by setting the velocity at the boundary and using the NS solver. Piston movement from run 4 from Table 3.2

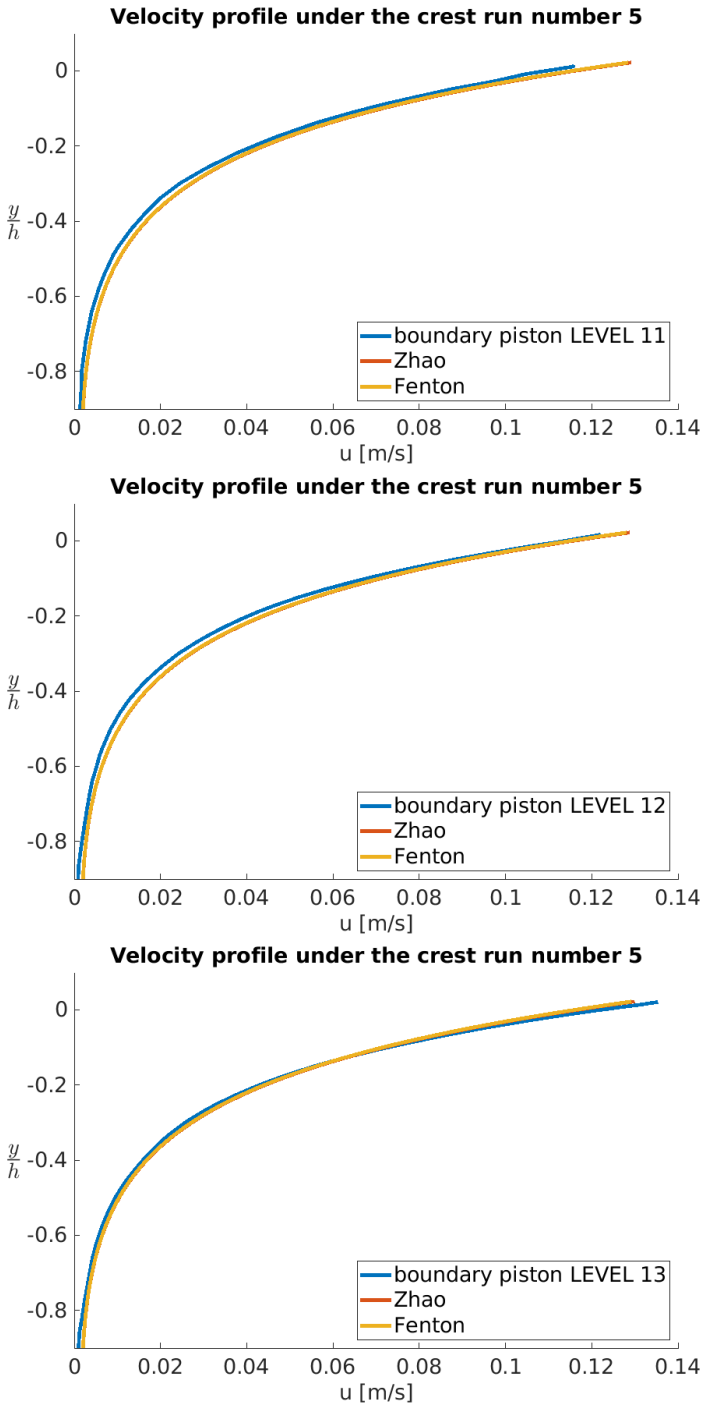

Figure 5.29: Horizontal velocity under the crest. The velocity is taken under the crest closest to 12.5m from the piston at 40s from the start of the piston movement. Velocity field generated by setting the velocity at the boundary and using the NS solver. Piston movement from run 5 from Table 3.2

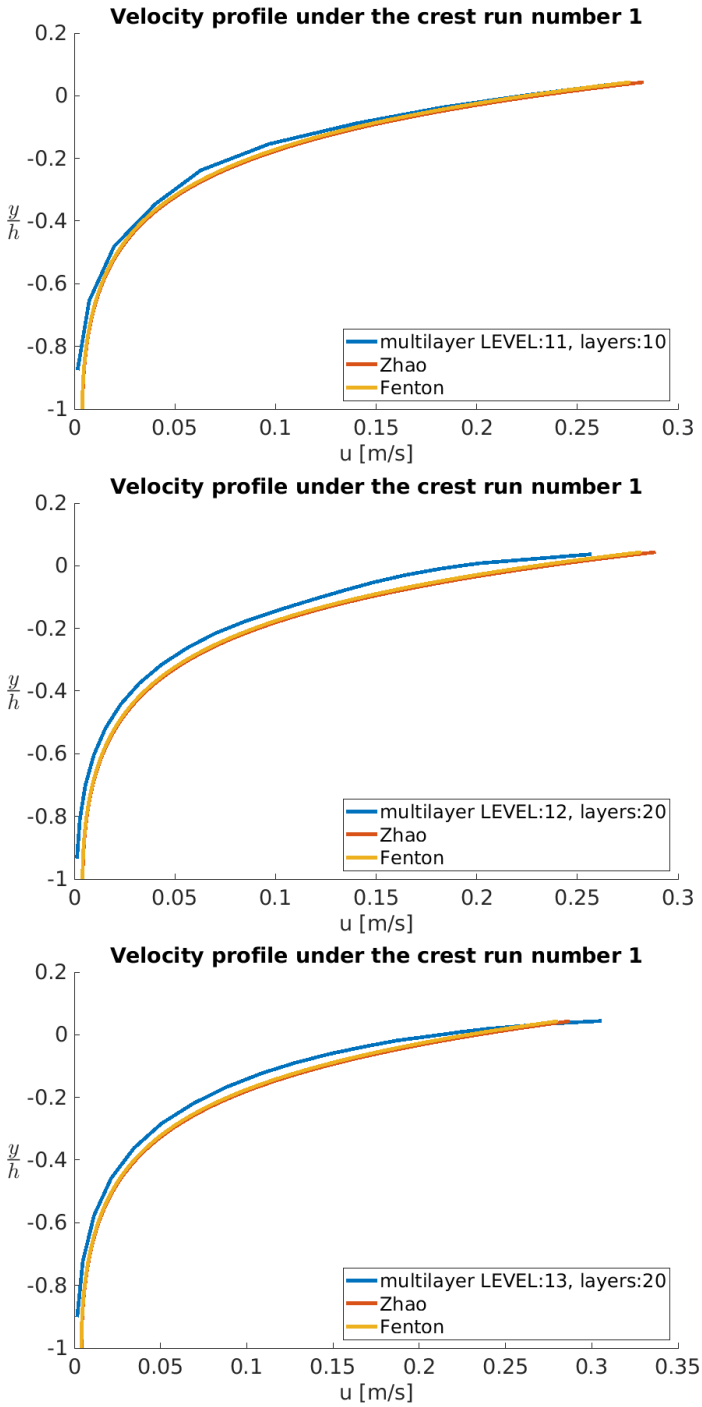

Figure 5.30: Horizontal velocity under the crest. The velocity is taken under the crest closest to 12.5m from the piston at 40s from the start of the piston movement. Velocity field generated by the Basilisk multilayer solver. Piston movement from run 1 from Table 3.2

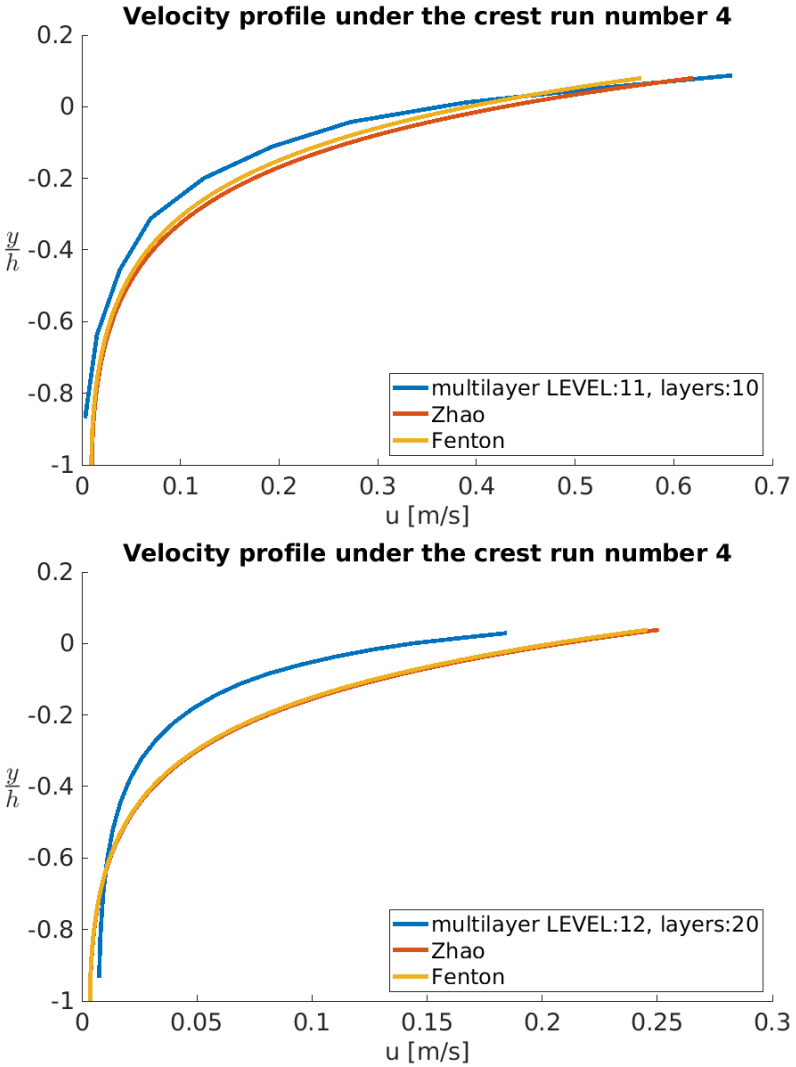

Figure 5.31: Horizontal velocity under the crest. The velocity is taken under the crest closest to 12.5m from the piston at 40s from the start of the piston movement. Velocity field generated by the Basilisk multilayer solver. Piston movement from run 4 from Table 3.2

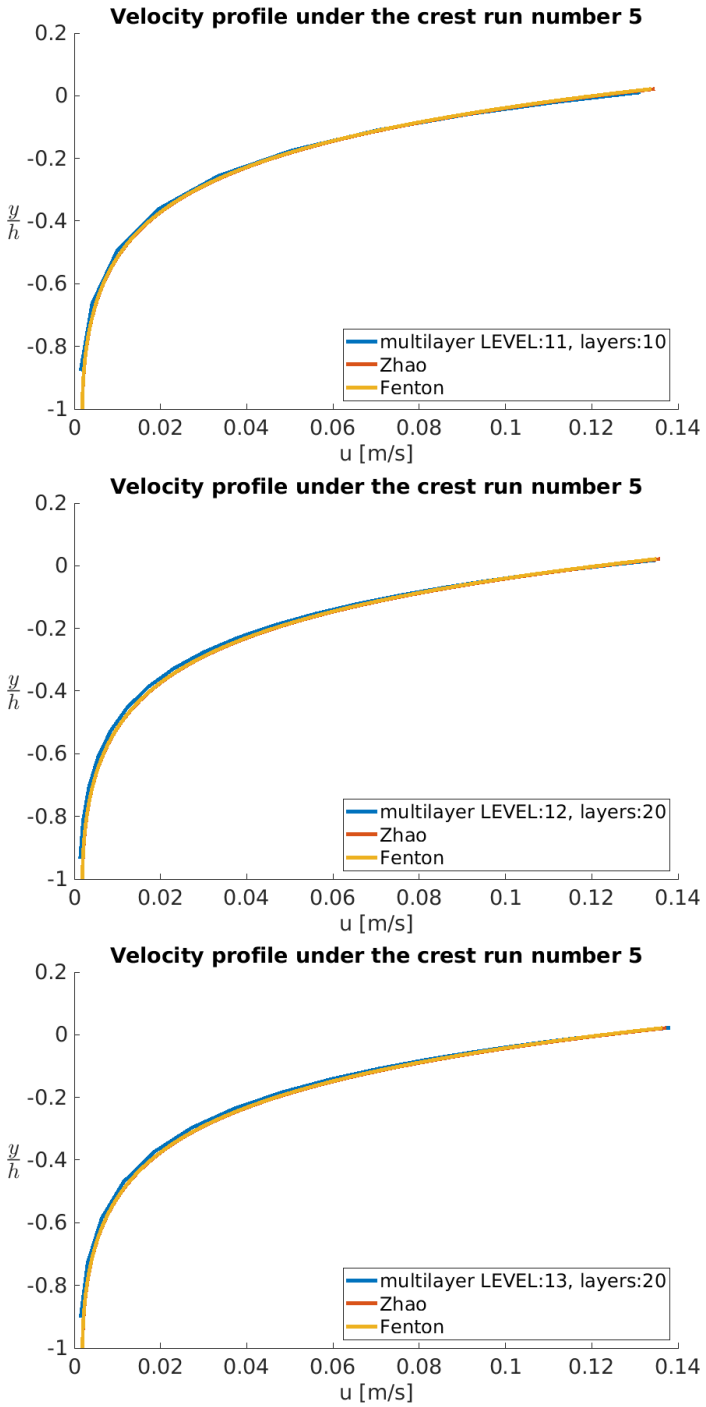

Figure 5.32: Horizontal velocity under the crest. The velocity is taken under the crest closest to 12.5m from the piston at 40s from the start of the piston movement. Velocity field generated by the Basilisk multilayer solver. Piston movement from run 5 from Table 3.2

	i7-9700K	E5-2695
Cores	8	18
L3 Cache	12MB	45MB
L2 Cache per core	256KB	256KB
L1 Cache per core	32KB	32KB
Max Frequency	4.90 GHz	3.30 GHz

Table 5.3: Some of the specs of the two processor types used for the simulation speed comparison.

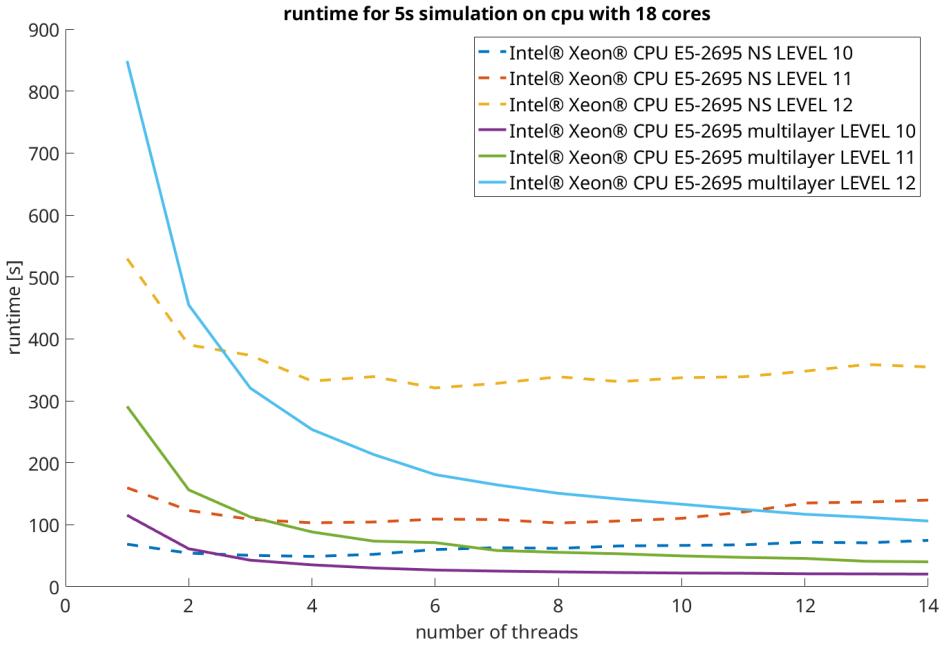

Figure 5.33: Comparison of the runtime of simulation of the same piston movement with the multilayer solver and the NS solver on the same system. For larger mesh sizes more cores help the speed of the NS solver more. The multilayer solver keeps improving when adding more cores.

5.3 Simulation speed

I ran most of these programs initially on an old laptop, but as I increased the mesh refinement, more computation power was needed. I ran some simulations on fram a linux cluster at UiT. But as I ran my simulations with openmp, I only ran them on one node at a time. As can be seen in Figure 5.33 with the simulation sizes I used for the 2d cases it does not scale much over 8 cores. This led me to running most of my simulations on my personal desktop computer or at a local machine at UIO. My machine has an Intel® Core™ i7-9700K processor while the UIO machine I used for these tests has two Intel® Xeon® CPU E5-2695 processors. In Figure 5.33 the runtimes for the multilayer solver and the NS solver are compared for similar mesh refinement using the E5-2695 processor. It is interesting to note that the multilayer solver is slower when using only one thread, but when using more threads, it overtakes the NS solver in terms of speed. The speed of the multilayer solver also keeps improving slightly when adding threads suggesting that is scales better for more threads.

Figure 5.34 shows the same plot as Figure 5.33 but for the i7-9700K. Using this processor the NS solver scales better with more threads. Interestingly with the i7-9700K

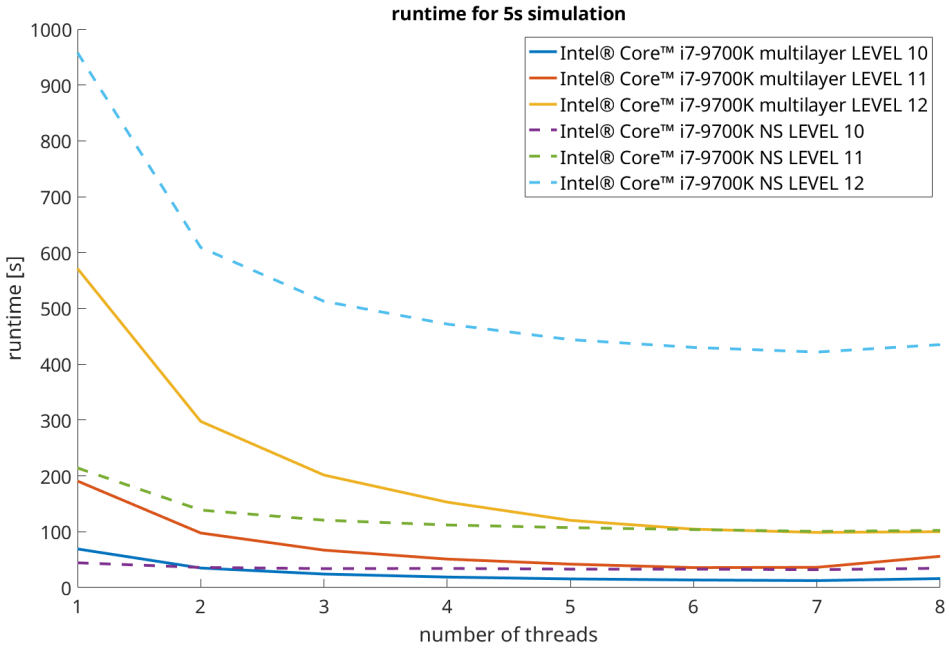

Figure 5.34: Comparison of the runtimes of the NS solver and the multilayer solver for 5 seconds simulation time. The multilayer solver offers faster simulations and keeps marginally improving after adding more cores than the NS solver.

the multilayer solver is faster, also when using only one thread, in contrast to the results in Figure 5.33. Why the multilayer solver is slower with 1 and 2 threads when using the E5-2695 processor is not immediately apparent to me and would require further examination to explain.

Figure 5.35 shows a plot of the runtimes of the multilayer solver for the to different processors. The i7-9700K is faster or comparable to the E5-2695 even when the E5-2695 uses more threads. I assume that the scalability of the multilayer solver would improve with a large mesh thus perhaps leading to the E5-2695 being faster when using more threads than are available to the i7-9700K.

In Figure 5.36 the runtimes for the NS solver are shown when run on the i7-9700K and the E5-2695 processors. The E5-2695 is faster for the larger models while the i7-9700K is faster for the smallest. I would guess that the improved speed of the E5-2695, for the largest model, comes from the larger L3 cache size. And that we would see the same effect if the multilayer solver was run with even higher mesh refinement.

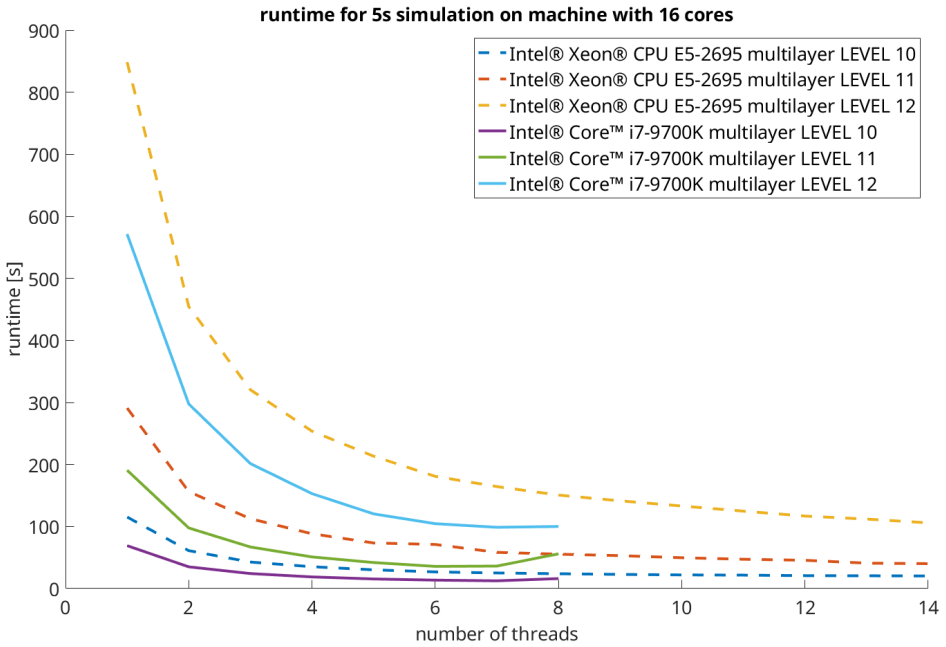

Figure 5.35: Comparison of multilayer runtimes with 8 core i7-9700K and 18 core E5-2695 processor. The i7-9700K is faster. The speed for the simulation decreases slightly when using the maximum amount of threads.

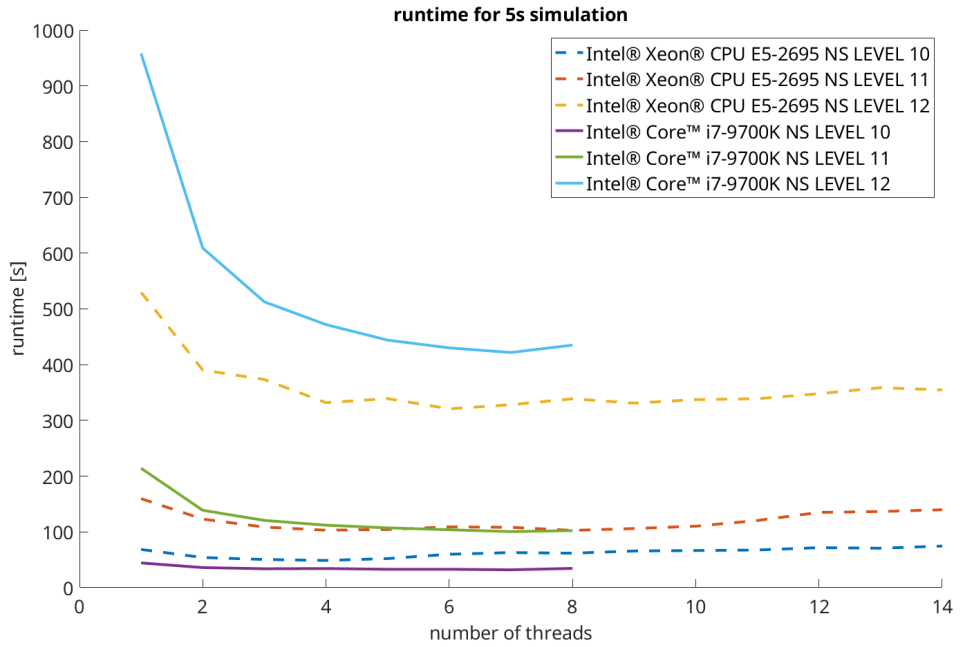

Figure 5.36: Comparison of NS runtimes with 8 core i7-9700K and 18 core E5-2695 processor. The E5-2695 is faster but gains no significant additional speed when using more than 6 cores.

Chapter 6

Conclusions and Future work

6.1 Conclusions

6.1.1 Basilisk simulations

Piston implementations

Of the three methods tested for piston implementation, the boundary methods with the NS solver and multilayer solver were the best at producing waves with amplitudes closest to the physical experiments. However, since the amplitudes of the physical experiments decayed as the waves travelled the length of the wave tank, Figure 5.8. When only examining the measurements from 8m out in the wave tank, the moving piston method often produced wave amplitudes close or closer than the other methods to the amplitudes from physical experiments. This is because the waves in the basilisk simulations do not decay as the waves in the physical wave tank. Some of the reason for the decay of the waves in the wave tank could be effects from the walls, so it would be interesting to repeat the physical experiment in a wider tank to examine if there is a difference in the way the waves decay.

6.1.2 Flow Velocity

When examining the results for the velocity profiles under the crests Figures 5.24 - 5.32, they fit relatively well for most runs with expected Stokes velocity profiles for their respective amplitude. The runs with the highest amplitudes show the largest deviation from the theoretical velocity profile as would be expected since they often have some form of breaking at the crests. The Basilisk velocity profiles both have velocities higher and lower than the theoretical profile. Some of this could be due to the fluctuations in amplitude apparent in Figures 5.10 - 5.12. The steepest waves I used had a tendency to break in the Basilisk simulations which caused the biggest deviations from the theoretical velocity profile. When examining the PIV results together with the velocity profiles from the Basilisk simulations in Figure **??**. It becomes clear that the difference between the simulations with the NS solver and the PIV measurements are not that far off Table 6.1, and some of this discrepancy could be explained by the differences in amplitudes from Figure 5.10. And the fact that the PIV measurements did not yield velocity data all the way up to the surface. However, the Multilayer solver in this instance yields much higher velocities that cannot be fully explained by the difference in amplitude.

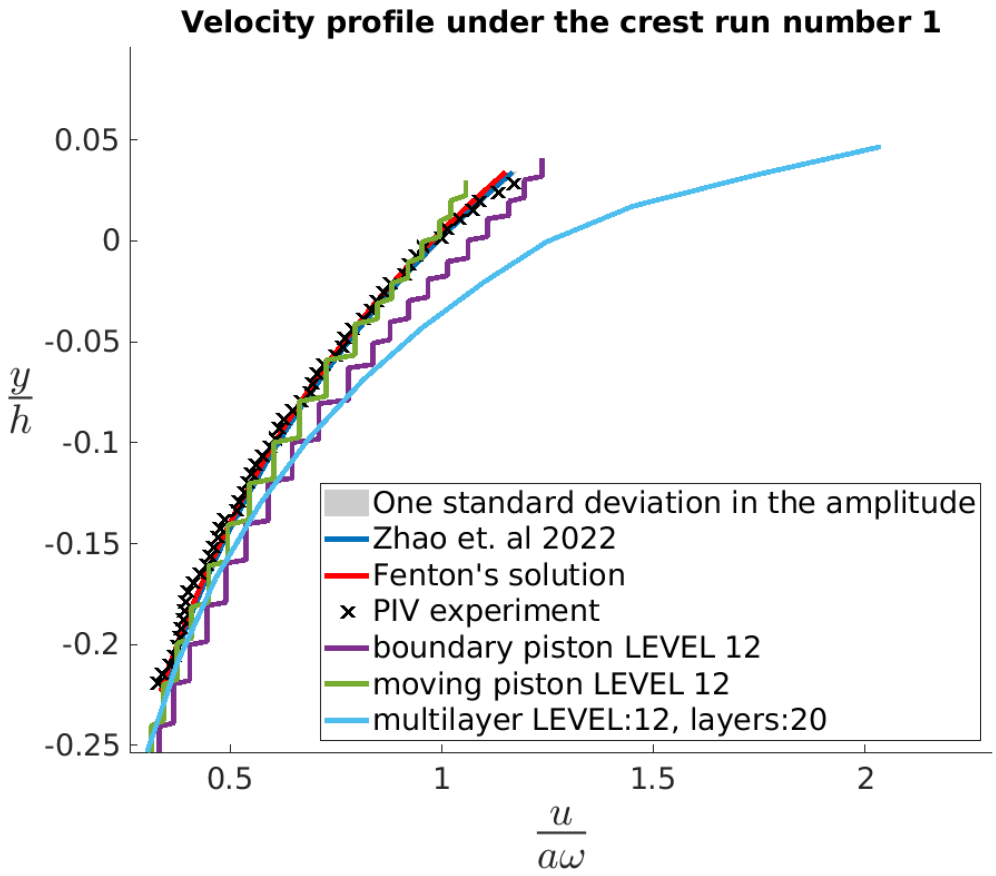

Figure 6.1: Comparison of the velocity profile under the crest for Basilisk results using run number 1, Table 3.2, for the piston movement.

Solver	Difference with PIV velocity at the crest
NS boundary piston	5.8%
NS moving piston	-9.6%
Multilayer	74.3%

Table 6.1: The difference in the velocity at the crest of the Basilisk simulations compared to piv results.

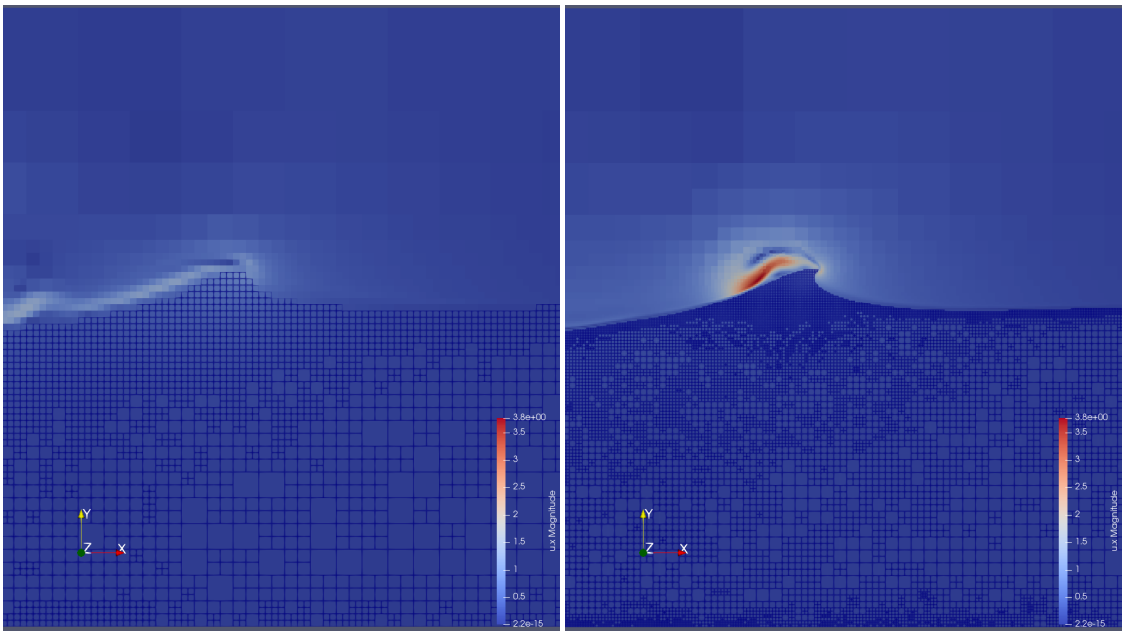

Figure 6.2: Image of the waves generated by the piston movement from run 4 from Table 3.2. The waves have a wave steepnes of roughly 0.31. On the left mesh refinement level 11, right mesh refinement level 13. The waves break sooner with higher mesh refinement. With lower mesh refinement the waves are spilling, with higher they are plunging. The velocities in the air are also much higher with higher mesh refinement.

Wave Breaking

Perlin, Choi and Tian 2013 discusses the limiting steepness before breaking occurs, and they find that in laboratory setups the limit can vary from 0.15 to 0.44. With the piston movement and laboratory setup that I used, larger piston movements quickly lead to breaking. For the basilisk simulations I ran the waves started breaking at lower wave steepness than the physical experiments. This could be due to numerical errors like rounding, too high tolerances in the code or, too high velocities in the air phase. The high velocities in the air phase when using the NS solver and higher refinement levels, can be seen when comparing the images in Figure 6.2. The piston implementation could also lead to a different wave profile and velocity than the physical experiments. It would be interesting to run the same physical experiments with a surface probe mounted to the piston to measure the surface elevation as close to the face of the piston as possible. This could be used to also set the surface elevation on the boundary in the Basilisk code and see to which degree this would change the simulation results.

Wave amplitudes

The mean measured amplitude from the basilisk experiments Figures 5.10, 5.11, 5.12 has a trough at about 0.5m from the piston. For most of the Basilisk simulations the mean amplitude also oscillates as if the surface profile contains standing waves. It could be interesting to repeat the experiments and measure the surface elevation at many additional locations to see which of these characteristics are present in the lab experiment.

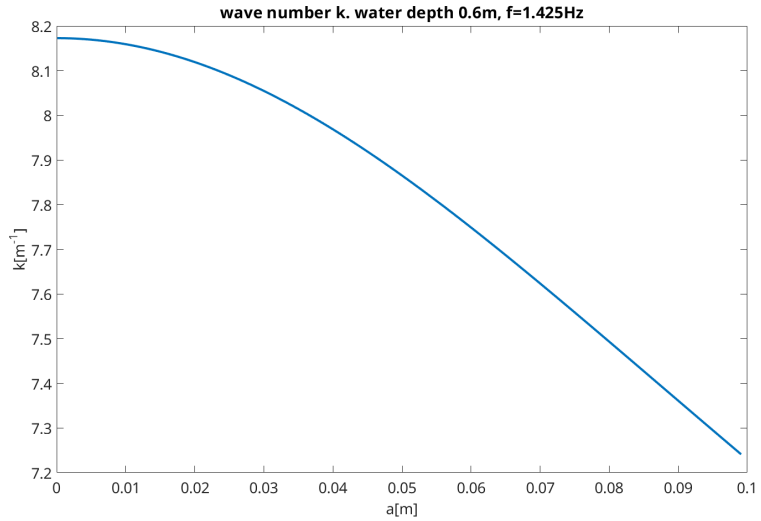

Figure 6.3: Plot of the wave number k calculated using Zhao and Liu 2022a for a range of amplitudes, with water depth h=0.6m and wave frequency f=1.425Hz.

Changes in phase

The changes seen in phase are generally larger the steeper the waves. This might be caused by breaking of the steepest waves. It can also be due to larger differences in amplitude for the larger piston movements and also the fact that the change in wavenumber is larger for the same change in wave amplitude for steeper waves. This can be seen from the plot of the wavenumber for different amplitudes in Figure 6.3. As long as the waves are not too steep, the Basilisk multilayer solver yields impressive results compared to the simulation time of the Navier-Stokes two phase solver.

6.2 Further work

6.2.1 Slave Master

After I started my work the ability to couple two different solvers has also been introduced to Basilisk. This makes it possible to use one solver close to the piston wave maker and another solver for the simulation away from the paddle. This allows for many new configurations of different solvers to possibly speed up or improve simulation accuracy. An example would be focusing waves where the area of the domain before the waves are focused is solved using the multilayer solver and the area where the waves become steep and might break is computed using the two-phase Navier-Stokes solver.

6.2.2 GPU

While I worked on this master thesis, the GPU utilization capabilities of Basilisk have increased greatly. So it has become an exciting avenue to explore for possible speed-up of simulation times.

6.2.3 Wave Tank Experiments

If the surface elevation was measured close to the piston in the physical experiments. This could possibly improve the basilisk simulations by making it possible to set the water surface level at the boundary. This could be done both for the multilayer solver and the NS solver when setting the velocity on the boundary.

Repeating the same experiments with more locations for the surface probes to get more data comparison to follow the evolution of the amplitude of the waves as they travel the length of the wave tank. To see which of the fluctuations in amplitude from Figures 5.10, 5.11 and 5.12 are physical effects and which are errors introduced in the simulation.

Repeating the experiments in a wider wave tank to investigate the effects of the side walls on the flow.

Another experiment that would be interesting is to do PIV measurements of steeper waves to see how well the results fit with the velocity profile calculated using Zhao and Liu 2022a where there is a larger difference with Fenton's solution.

Another experiment that could be done to compare with the Basilisk simulation results would be to measure the air velocity or introduce wind to the experiments.

6.2.4 3D Wave field

Perlin, Choi and Tian 2013 mentions that for spatially focusing waves arriving from different angles the wave steepness can be increased before breaking occurs. Up to double the steepness when the angle between the waves is 90 degrees. An interesting avenue for future work would be to attempt to reproduce this in both a wave tank and with CFD.

6.2.5 Comparison With Other CFD Software

It would also be interesting to run the simulations with other open source CFD software like REEF3D, OpenFOAM, FEniCS or other commercial alternatives like Ansys. This would allow comparison with Basilisk for both accuracy and efficiency.

6.2.6 PIV

If I were to do more PIV measurements, I would like to perform PIV measurements. This would make it possible to compare the flow close to the piston, from the physical experiments with Basilisk simulations. The results for the velocity closer to the piston would give better insight into the correspondence between the piston implementation methods and the resulting flow and not, be as affected by the evolution of the flow as the simulation progresses.

Another thing that would be of interest is to repeat the PIV measurements with steeper waves to possibly detect flow matching better to one of the solutions in Zhao and Liu 2022a.?

Part IV

Appendix

Appendix A

Basilisk

A.1 Piston implementation

A.1.1 Navier-Stokes Solver - Moving Piston

I wanted to use the measurements of the piston position as input to the simulations. So I read the piston position and stored it in an array and calculated the piston speed and stored it in another array. I also created a scalar field in the Basilisk program. The main lines of the piston method are in the following code snippet.

```
#define PISTON piston_position-x
scalar pstn[];

event piston (i++, first) {
  piston_counter = floor(t*100);
  piston_position = piston_positions[piston_counter];
  U_X = piston_ux[piston_counter];
  fraction (pstn, PISTON);
  foreach() {
    u.y[] = u.y[]*(1 - pstn[]);
    u.x[] = pstn[]*U_X + u.x[]*(1 - pstn[]);
  }
  u.n[left]=dirichlet(U_X);
}
```

PISTON is defined as a macro and pstn created as a scalar field in the start of the script. In the piston event that is run at every time step the piston position is updated and scalar field pstn is updated with the value 1 being inside the piston and 0 outside. Then the velocity inside the piston is changed for both the vertical and horizontal velocity. When I moved the piston was moved all the way to the boundary of the domain, I also added setting the velocity at the boundary in the last line. I later updated this method to use linear interpolation for the position and velocity of the piston, before in the final version I made of this method also wrapping the velocity and piston position in functions like I also did for the final version of the other piston methods. This method works but because the piston leaks it consistently yields small amplitudes than the physical experiments as seen in Figure 5.10, Figure 5.11 and Figure 5.12.

A.1.2 Navier-Stokes Solver - Boundary Piston

My first attempt at modelling a piston by simply setting the velocity on the boundary looked like this.

```
event piston(i++){
    u.n[left] = dirichlet(piston_ux[(int)floor(t*file_samplerate)]);
```

```
}
```

Where the velocity is set to a constant at every time step. This does work when the changes in velocity between time step are small. But for higher rates of change in the boundary velocity this leads to problems when Basilisk solves for the pressure and strange patterns in the velocity field.

Luckily Øystein Lande could help me with a solution. Some of the problem was the way I was setting the velocity on the boundary did not update the velocity for the half steps in the numerical scheme. Another problem is that the boundary condition for the pressure by default is set to neumann(0) in Basilisk, which does not match with a variable velocity at the boundary. The implementation of these solutions for the NS solver is outlined in the following code snippet.

```
#define neumann_pressure_variable(i) ((paddle_rampoff(y)*(Wave_VeloX(0,0,0,t+dt)
    -//
                                      Wave_VeloX(0,0,0,t))/dt - a.n[i])*fm.n[i]/
    alpha.n[i])

double Wave_VeloX(double x , double y, double z, double t){
  int t0_i = (int)floor(t*file_samplerate);
  return piston_ux[t0_i] + (piston_ux[t0_i+1]-piston_ux[t0_i])*(t*file_samplerate-
    t0_i);
}

double paddle_rampoff(double y){
  double ramp = 1;
  if (y>ramp_start){
    ramp = (ramp_start+ramp_distance-y)/ramp_distance;
    if (y>ramp_start+ramp_distance){
      ramp = 0;
    }
  }
  return ramp;
}

init(){
    ...
    u.n[left] = dirichlet(Wave_VeloX(0,0,0,t)*paddle_rampoff(y));
    p[left]   = neumann(neumann_pressure_variable(0));
    ...
}
```

Here the pressure gradient is calculated based on the change in piston velocity. The velocity to be set on the boundary is contained in a function dependent on t, so that Basilisk can update the value when calculating at half steps. I initially did not have a linear interpolation for the velocity in this function, but that led to errors when calculating the pressure gradient. A ramp-off function is also needed to avoid interference with the outflow boundary conditions for the top boundary. Finally, the boundary conditions now only have to be set once, in the init function instead of at every time step.

A.1.3 Multilayer Solver

My first attempt at modelling the piston in the multilayer solver was the same as I did when setting the velocity on the boundary in the NS solver.

```
event piston_update(i++){
  piston_counter = floor(t*file_samplerate);
  u.n[left] = dirichlet(piston_ux[piston_counter]);
}
```

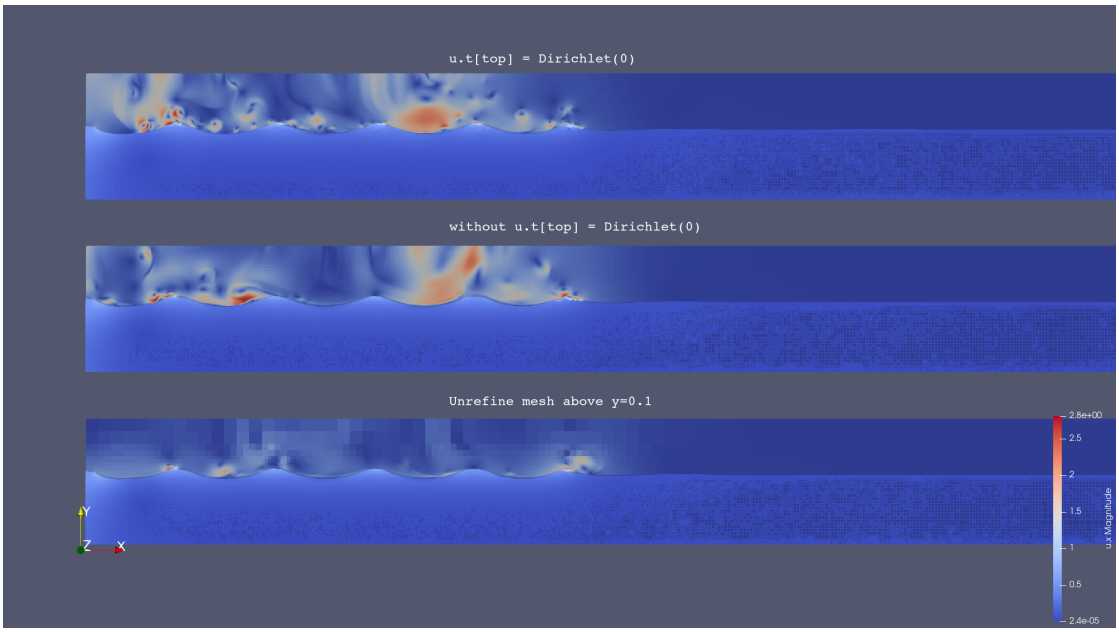

Figure A.1: Different ways of treating the top boundary, to avoid high velocities.

This led to the same problems as with the NS solver with the velocity close to the boundary. Øystein Lande helped me with the solution for the multilayer solver as well. Since the multilayer solver isn't a two phase solver, the solution is simpler. The velocity function is done in the same way. The rampoff function is no longer needed and the pressure function is simpler.

```
1 # define neumann_pressure_variable(i) (((Wave_VeloX(0, 0, 0, t+dt)-Wave_VeloX(0,
      0, 0, t))/dt))
```

The boundary conditions are set in the same way as for the NS solver in the init event.

A.1.4 Boundary Conditions - Not Piston Boundaries

The boundary conditions I have set for the boundaries in my scripts are for the right boundary either neumann(0) or dirichlet(0) corresponding to free outflow or a solid boundary. I have not done measurements trying to determine the quality of the reflections calculated by Basilisk compared to experiments. The measurements I used for comparison were before the reflection from the end of the tank returned to the measuring locations. However, I noticed that setting neumann(0) boundary conditions on the outflow boundary would lead to some reflection. So if the target was no reflections, when using Basilisk, some damping would probably be necessary at the outflow.

The domain has an open top allowing flow through the top boundary with Neumann boundary conditions. This does however cause some strange behaviour with runaway velocities at the top boundary, shown in Figure A.1. I tried different solutions for treating the boundary, to solve these issues. Like setting the tangential velocity to zero at the top boundary. The most robust solution I found, which I ended up using, was to coarsen the mesh for a set height above the water surface. This dampened the small-scale flow and removed the issues with runaway velocities at the top boundary while still allowing for flow through the boundary. A movie of the resulting simulation is available at Gimse 2025b.

A.2 Included Solvers and Functions

A.2.1 Surface probes

I have made a surface_probes event in my scripts that saves the surface elevation at selected locations. It uses the heights function from heights.h, which can give the vertical distance to the interface, for the cells closest to the surface.

A.2.2 Adapt wavelet leave interface

Based on adapt_wavelet_leave_interface.h Øystein Lande 2018 that refines and unrefines the mesh based on given tolerances and also keeps the highest refinement level around the air-water interface. I have only made one change to this function, which is to include a second refinement level to achieve higher refinement around the piston interface.

A.2.3 Navier Stokes centered

The Basilisk header file navier-stokes/centered.h Stephane Popinet 2025c contains the main methods used for solving the incompressible variable density Navier-Stokes equations, equation 2.30 and 2.31. It uses the Bell-Collela-Glaz advection scheme 2.6.1 an implicit viscosity solver for the viscous diffusion equation. And it is possible to restrict the timestep with the CFL (Courant–Friedrichs–Lewy) parameter. The CFL parameter limits the size of the timestep such that $\mathrm{CFL} u \Delta t < \Delta x$ to increase stability.

A.2.4 Two phase

The water is described by a fraction field for every cell in the grid, where a value of 1 is water and 0 is air. src/two-phase.h Stephane Popinet 2025eis a header file which includes the vof - volume of fluid methods from vof.h, to solve the advection equation Stephane Popinet 2025f.

$$\partial_t c_i + \mathbf{u}_f \cdot \nabla c_i = 0 \tag{A.1}$$

Where c_i denotes the phases with sharp interfaces.

A.2.5 Navier Stokes conserving

The Basilisk file src/navier-stokes/conserving.h Stephane Popinet 2025d implements momentum conservation for the two-phase solver. It conserves the total momentum not the momentum in each phase.

A.2.6 Reduced gravity

In reduced.h gravity is reformulated as an interfacial force Stéphane Popinet 2018

$$-\nabla p + \rho \mathbf{g} = -\nabla p' - [\rho]\mathbf{g} \cdot \mathbf{x}\mathbf{n}\delta_s \tag{A.2}$$

For some of my runs this reduced the formation of high air velocities in vortexes above the water surface, but I have not performed a full test of the impact this has after updating the piston implementations.

A.3 Basilisk Scripts

All the Basilisk scripts are also available at Gimse 2025a.

A.3.1 Multilayer Initialised Wave

This script is made using bar.c Stephane Popinet 2024c and breaking.c Stephane Popinet 2024a

```
    /**
# intialised third order stokes wave with periodic boundary conditions

based on bar.c and breaking.c
the parameters for the wave are from
  Jensen, A., Sveen, J. K., Grue, J., Richon, J. B., & Gray, C. (2001).
  Accelerations in water waves by extended particle image velocimetry.
  Experiments in Fluids, 30(5), 500-510. https://doi.org/10.1007/s003480000229
*/

#include "grid/multigrid1D.h"
#include "layered/hydro.h"
#include "layered/nh.h"
#include "layered/remap.h"

#include "layered/check_eta.h"
#include "layered/perfs.h"
#include "output_pvd.h"


int set_n_threads = 2; //set to 0 to use all available threads
double Tend = 10;      //the end time of the simulation
#define nl_ 10  //the default number of layers if none are given as command line arguments
double rmin = 0.5;  //changed later in the script to make square cells at the surface.
int LEVEL = 7;       //the grid resolution in x direction Nx = 2**LEVEL
double rho = 997;
double k_ = 7.9596;    //the wavenumber
double a = 0.0205;     //the amplitude of the wave
double ak = 7.9596*0.0205;
double h_ = 0.6;       //water depth
double g_ = 9.81;

int n_waves = 1;       //number of waves to fit the domain to
#define Lx 2*pi/k_*n_waves  // the length of the simulation domain

char results_folder[40]; //the location to save the results
char vts_folder[50]; //the locaton to save the vtu files
char guage_name[50];

#include "test/stokes.h" //third order stokes wave


/**
We use Stokes third order wave from src/test/stokes.h to initialise the surface elevation
and the velocity field*/

event init (i = 0)
{
  geometric_beta(rmin, true); //set the layer thickness smaller nearer the surface
  foreach() {
    zb[] = -h_;
    foreach_layer(){
      //set the thickess of the layer based on the surface of the wave
      h[] = (max(- zb[], 0.)+wave(x, z))*beta[point.l];
    }
  }
```

```
  /*set the velocity field of the wave*/
  foreach(){
    double z = zb[];
    foreach_layer(){
      z += h[]/2;
      u.x[] = u_x(x,z);
      w[] = u_y(x,z);
      z += h[]/2;
    }
  }
}

int main(int argc, char *argv[])
  {
  nl=nl_;
  for(int j=0;j<argc;j++){
    if (strcmp(argv[j], "-L") == 0)
    {
      LEVEL = atoi(argv[j + 1]);
    }
    if (strcmp(argv[j], "-nl") == 0)
    {
      nl = atoi(argv[j + 1]);
    }
    if (strcmp(argv[j], "-n") == 0)
    {
      n_waves = atoi(argv[j + 1]);
    }
  }

  sprintf(results_folder, "results/LEVEL%d_layers%d", LEVEL, nl);
  sprintf(vts_folder, "%s/vts", results_folder);

  char remove_old_results[100];
  sprintf(remove_old_results, "rm -r %s", results_folder);
  if (system(remove_old_results)==0){
    printf("removed old results in:%s\n", results_folder);
  }

  char make_results_folder[100];
  sprintf(make_results_folder, "mkdir -p %s", vts_folder);
  if (system(make_results_folder)==0){
    printf("made results folder:%s\n", results_folder);
  }
  sprintf(guage_name, "%s/X_0", results_folder);

  //copy the script to the results folder for later inspection if needed
  char copy_script[100];
  sprintf(copy_script, "cp multilayer.c %s/multilayer.c", results_folder);
  if (system(copy_script)==0){
    printf("copied script to results folder\n");
  }

  origin (-Lx/2., h_);
  periodic(right);
  N = 1<<LEVEL;
  rmin = sqrt(Lx*nl/N/h_);
  L0 = Lx;
  G = g_;
  breaking = 1.0;
  CFL_H = .1;
  TOLERANCE = 10e-5;
  theta_H = 0.51;
  nu = 1e-6;
  printf("nu:%e\n", nu);
  #if _OPENMP
  int num_omp = omp_get_max_threads();
  fprintf(stderr, "max number of openmp threads:%d\n", num_omp);
  if (set_n_threads){  //set number of omp threads
    omp_set_num_threads(set_n_threads);
```

```
  }
  fprintf(stderr, "set openmp threads:%d\n", omp_get_max_threads());
#endif

run();
}

event save_velocity(t += Tend; t<=Tend)
{
  char filename[200];
  sprintf(filename, "%s/velocities_nx%d_nl%d_timestep_%d.csv", results_folder, 1<<
    LEVEL, nl, i);
  fprintf(stderr, "saving results to:%s\n", filename);
  FILE *fp = fopen(filename, "w"); //if at the first timestep overwrite the
    previous file, can later add run parameters here
  fprintf(fp, "\"Time\",\"layer\",\"Points:0\",\"Points:1\",\"Points:2\",\"u.x
    \",\"u.z\",\"eta\"\n");
  foreach(serial){
    double z = zb[];
    foreach_layer(){
      z += h[]/2;
      fprintf(fp, "%f, %d, %f, %f, %f, %f, %f, %f\n", t, point.l, x, 0., z, u.x[],
     w[], eta[]);
      z += h[]/2;
    }
  }
  fclose(fp);
}

event save_energy(i++)
{
  char filename[200];
  sprintf(filename, "%s/energy_nx%d_nl%d.csv", results_folder, 1<<LEVEL, nl);
  static FILE * fp = fopen(filename, "w");
  double gpe = 0.;
  double ke = 0.;

  foreach(reduction(+:ke) reduction(+:gpe)){
    double z = zb[]*0;/**

based on bar.c and breaking.c
the parameters for the wave are from
  Jensen, A., Sveen, J. K., Grue, J., Richon, J. B., & Gray, C. (2001).
  Accelerations in water waves by extended particle image velocimetry.
  Experiments in Fluids, 30(5), 500-510. https://doi.org/10.1007/s003480000229
*/

#include "grid/multigrid1D.h"
#include "layered/hydro.h"
#include "layered/nh.h"
#include "layered/remap.h"
#include "layered/check_eta.h"
#include "layered/perfs.h"
#include "output_pvd.h"

int set_n_threads = 8; //set number of threads manually
int LEVEL = 11;      //the grid resolution in x direction Nx = 2**LEVEL

double Tend = 50;    //the end time of the simulation
// double Lx = 24.6; //The length of the simulation domain
double Lx = 24.6;
#define nl_ 10  //the default number of layers if none are given as command line
    arguments
double rmin = 0.5;  //rmin the relative height of the top layer compared to
                    //a regular distribution. the rest fo the layers follow a
    geometric distribution.

double h_ = 0.6;    //water depth

// piston file
```

```
191 int run_number = 1; //default run number if none is given in the command line
    piston files in "piston_files/%run_number/fil3.dat";
192 int file_samplerate = 100; //the samplerate of the piston speed file
193 #define piston_timesteps 10000//the max number of timesteps in the piston file
194 int piston_counter;
195 double piston_ux[piston_timesteps];
196 double neumann_pressure[piston_timesteps];
197 //piston parameters
198
199 char results_folder[40]; //the location to save the results
200 char vts_folder[50]; //the locaton to save the vtu files
201 //surface probes
202 double probe_positions[1404] = {1.5,10.048,10.745,11.498};
203 int n_probes = 1404;
204
205
206 void read_piston_data(){
207   char piston_file[40];
208   sprintf(piston_file, "piston_files/%d/piston_speed.dat", run_number);
209   printf("piston file:%s\n", piston_file);
210
211   int count = 0;
212   FILE *file;
213   file = fopen(piston_file, "r");
214   if(!file)
215     {
216         perror("Error opening piston position file");
217     }
218   int _running=1;
219   while(_running && count< piston_timesteps ){
220     _running = fscanf(file, "%lf", &(piston_ux[count]));
221     count++;
222   }
223   fclose(file);
224   for (int i=count;i<piston_timesteps;i++){
225     piston_ux[i] = 0; //set the remaining timesteps to 0
226   }
227 }
228
229 event setup_probe_positions(i=0){
230   double probe_x = 0.1;
231   for(int j=4;j<n_probes;j++){
232     probe_positions[j] =probe_x;
233     probe_x = probe_x+0.01;
234   }
235 }
236
237 double Wave_VeloX(double x , double y, double z, double t){
238   int t0_i = (int)floor(t*file_samplerate);
239   //linear interpolation of the piston speed f(t) = f(t0) +(f(t1)-f(t0))*(t-t0)/(
      t1-t0)
240   return piston_ux[t0_i] + (piston_ux[t0_i+1]-piston_ux[t0_i])*(t*file_samplerate-
      t0_i);
241 }
242
243
244
245 event init (i = 0)
246 {
247   #if _OPENMP
248   fprintf(stderr, "current threads:%d\n", omp_get_num_threads());
249   fprintf(stderr, "max number of openmp threads:%d\n", omp_get_max_threads());
250   #endif
251
252   geometric_beta(rmin, true); //set the layer thickness smaller nearer the surface
253   foreach() {
254     zb[] = -h_;
255     foreach_layer(){
256       h[] = (max(- zb[], 0.))*beta[point.l];
257     }
```

```
258   }
259
260   # define neumann_pressure_variable(t) (((Wave_VeloX(0, 0, 0, t+dt)-Wave_VeloX(0,
          0, 0, t))/dt))
261   u.n[left] = dirichlet(Wave_VeloX(0,0,0,t));
262   phi[left] = neumann(neumann_pressure_variable(t));
263   u.n[right] = neumann(0.);
264 }
265
266 int main(int argc, char *argv[])
267   {
268     //set LEVEL, layers and run number from command line args
269   nl = nl_;
270   for(int j=0;j<argc;j++){
271     if (strcmp(argv[j], "-L") == 0)
272         {
273             LEVEL = atoi(argv[j + 1]);
274         }
275     if (strcmp(argv[j], "-nl") == 0)
276     {
277       nl = atoi(argv[j + 1]);
278     }
279     if (strcmp(argv[j], "-r") == 0)
280     {
281       run_number = atoi(argv[j + 1]);
282     }
283   }
284   //make folders for saving the results
285   sprintf(results_folder, "results/run%d/LEVEL%d_layers%d", run_number, LEVEL, nl)
        ;
286   sprintf(vts_folder, "%s/vts", results_folder);
287
288   char remove_old_results[100];
289   sprintf(remove_old_results, "rm -r %s", results_folder);
290   if (system(remove_old_results)==0){
291     printf("removed old results in:%s\n", results_folder);
292   }
293
294   char make_results_folder[100];
295   sprintf(make_results_folder, "mkdir -p %s", vts_folder);
296   if (system(make_results_folder)==0){
297     printf("made results folder:%s\n", results_folder);
298   }
299
300   //copy the script to the results folder for later inspection if needed
301   char copy_script[100];
302   sprintf(copy_script, "cp multilayer.c %s/multilayer.c", results_folder);
303   if (system(copy_script)==0){
304     printf("copied script to results folder\n");
305   }
306
307   origin (0, h_);
308
309   N = 1<<LEVEL;
310   L0 = Lx;
311   G = 9.81;
312   breaking = 1;
313   CFL_H = .1;
314   CFL = 0.1;
315   rmin = sqrt(Lx*nl/N/h_);
316   printf("CFL=%f, CFL_H=%f\n", CFL, CFL_H);
317   TOLERANCE = 10e-8;
318   //printf("TOLERANCE:%f\n", TOLERANCE);
319   theta_H = 0.51;
320   nu = 1e-6;
321   #if _OPENMP
322   if (set_n_threads>0)
323   {
324   int num_omp = omp_get_max_threads();
325   fprintf(stderr, "max number of openmp threads:%d\n", num_omp);
```

```
  omp_set_num_threads(set_n_threads);
  fprintf(stderr, "set openmp threads:%d\n", omp_get_num_threads());
  }
  #elif _MPI
  fprintf(stderr, "npe:%d\n", npe());
  #endif
  read_piston_data();
  run();
  }
#if _OPENMP
event output_field (t <= Tend; t += 1)
{
    fprintf(stdout, "field vts output at step: %d, time: %.2f \n", i, t);
    static int j = 0;
    char name[100];
    sprintf(name, "%s/field_%.6i.vts",vts_folder, j++);
    fprintf(stdout, "written to: %s\n", name);
    FILE* fp = fopen(name, "w");
    output_vts_ascii_all_layers(fp, {eta,h,u}, N);
    fclose(fp);
    #if _OPENMP
    omp_set_num_threads(set_n_threads);
    #endif
}
#endif

 event save_energy (t += 0.01)
{
  char filename[200];
  sprintf(filename, "%s/energy.csv", results_folder);
  static FILE * fp = fopen (filename, "w");
  double ke = 0., gpe = 0.;
  foreach (reduction(+:ke) reduction(+:gpe)) {
    double zc = zb[];
    foreach_layer() {
      double norm2 = sq(w[]);
      foreach_dimension()
        norm2 += sq(u.x[]);
      ke += norm2*h[]*dv();
      gpe += (zc + h[]/2.)*h[]*dv();
      zc += h[];
    }
  }
  if (i == 0)
    fprintf (fp, "t, ke, gpe\n");
  fprintf(fp, "%f, %f, %f\n", t, 997*ke/2., 997*9.81*gpe);
}

event show_progress(i++)
{
  float progress = 0;
  progress = t /Tend;
  printf("t=%.3f, i=%d, dt=%g, run=%d, LEVEL=%d, nl=%d, ", t, i, dt, run_number,
    LEVEL, nl);
  printf("%.2f%%\n", progress*100);
}

event surface_probes(t+=0.01){
  char filename[100];
  sprintf(filename, "%s/surface_probes.csv", results_folder);
  FILE *fp = fopen(filename, "a");
    if (i==0){
      fprintf(fp, "time");
      for (int j = 0;j<n_probes;j++){
        fprintf(fp, ", %f", probe_positions[j]);
      }
      fprintf(fp, "\n");
  }
  fprintf(fp, "%f", t);
  for (int probe_n=0;probe_n<n_probes;probe_n++){
```

```
395
396     fprintf(fp, ", %f", interpolate(eta, probe_positions[probe_n],-h_));
397   }
398   fprintf(fp, "\n");
399   fclose(fp);
400 }
401
402     foreach_layer(){
403       double norm2 = sq(w[]);
404       foreach_dimension()
405         norm2 += sq(u.x[]);
406       ke += norm2*h[]*dv();
407       gpe += (z + h[]/2)*h[]*dv();
408       z += h[];
409     }
410   }
411   if (i == 0)
412     fprintf (fp, "t, ke, gpe\n");
413   fprintf(fp, "%f, %e, %e\n", t, rho*ke/2., rho*g_*gpe);
414 }
```

A.3.2 Navier-Stokes Solver Initialised Wave

This script is based on stokes.c Stephane Popinet 2025b and stokes-ns.c Stephane Popinet 2024b from basilisk.fr.

```
1  /**
2  # intialised Stokes wave
3
4  We solve the two-phase Navier--Stokes equations using a
5  momentum-conserving transport of each phase. Gravity is taken into
6  account using the "reduced gravity approach". */
7  #include "adapt_wavelet_leave_interface.h"
8  #include "navier-stokes/centered.h"
9  #include "two-phase.h"
10 #include "navier-stokes/conserving.h"
11 #include "reduced.h"
12 #include "output_vtu_foreach.h"
13 #include "heights.h"
14 int vtucount = 0;
15
16 /**
17 The primary parameters are the wave steepness $ak$ and the wavenumber
18 */
19 double T0 = 10;
20 double ak = 0.16314515;
21 int LEVEL = 3;
22 int max_LEVEL = 8;
23 int padding =  2;
24 double Lx = 0.7895; // the wavelength of the wave
25 int n_waves = 1; // the number of wavelengths to fit in the domain
26
27 int set_n_threads = 2;
28 vector h[]; //scalar field of the distance from the surface, using heights.h
29 char results_folder[40]; //the location to save the results
30 char vtu_folder[50]; //the locaton to save the vtu files
31 char energy_file[60];
32
33 int n_probes = 1;
34 double probe_positions[] = {0.00001};
35 /**
36 The error on the components of the velocity field used for adaptive
37 refinement. */
38
39 double uemax = 0.1;
40
41 /**
42 The wave number, fluid depth and acceleration of gravity are set to
```

```
43 these values.**/
44
45 double k_   = 7.9583;
46 double h_   = 0.6;
47 double g_   =  9.81;
48
49 /**
50 The program takes optional argument LEVEL
51 */
52
53 int main (int argc, char * argv[])
54 {
55   for(int j=0;j<argc;j++){
56     if (strcmp(argv[j], "-L") == 0)
57     {
58       max_LEVEL = atoi(argv[j + 1]);
59       LEVEL = max_LEVEL - 4;
60     }
61     if (strcmp(argv[j], "-n") == 0)
62     {
63       n_waves = atoi(argv[j + 1]);
64     }
65   }
66   sprintf(results_folder, "results/LEVEL%d_nwaves%d", max_LEVEL, n_waves);
67   sprintf(vtu_folder, "%s/vtu", results_folder);
68   sprintf(energy_file, "%s/energy.txt", results_folder);
69
70   char remove_old_results[100];
71   sprintf(remove_old_results, "rm -r %s", results_folder);
72   if (system(remove_old_results)==0){
73     printf("removed old results in:%s\n", results_folder);
74   }
75
76   char make_results_folder[100];
77   sprintf(make_results_folder, "mkdir -p %s", vtu_folder);
78   if (system(make_results_folder)==0){
79     printf("made results folder:%s\n", results_folder);
80   }
81
82   L0 = Lx*n_waves;
83   origin (-L0/2, -h_);
84   periodic (right);
85
86   rho1 = 997;
87   rho2 = 1.204;
88   mu1 = 8.9e-4;
89   mu2 = 17.4e-6;
90   G.y = -g_;
91   DT = Lx/(1<<(max_LEVEL-1));
92   printf("DT=%.4f\n", DT);
93   N = 1 << LEVEL;
94   CFL = 0.1;
95 #if _OPENMP
96   int num_omp = omp_get_max_threads();
97   fprintf(stderr, "max number of openmp threads:%d\n", num_omp);
98   if (set_n_threads){  //set number of omp threads
99     omp_set_num_threads(set_n_threads);
100  }
101  fprintf(stderr, "set openmp threads:%d\n", omp_get_max_threads());
102  run();
103 }
104
105
106 #include "test/stokes.h"
107
108 event init (i = 0)
109 {
110   u.n[top] = neumann(0);
111   p[top]    = dirichlet(0.);
112   u.t[top]  = dirichlet(0.);
```

```
113
114   fraction (f, wave(x, y)); //set the water depth
115   foreach()
116     foreach_dimension()
117       u.x[] = u_x(x,y) * f[];
118
119   while (adapt_wavelet_leave_interface({u.x, u.y, p},{f},(double[]){uemax,uemax,
        uemax, uemax},max_LEVEL, LEVEL,padding).nf){  //for adapting more around the
        piston interface
120     printf("refining\n");
121     fraction (f, wave(x, y)); //set the water level on the refined mesh
122     foreach()
123       foreach_dimension()
124         u.x[] = u_x(x,y) * f[];
125   }
126 }
127
128 event logfile (i++)
129 {
130   FILE *fp = fopen(energy_file, "a");
131     if (i==0){
132       fprintf(fp, "t, ke, gpe\n");
133   }
134   double ke = 0., gpe = 0.;
135   foreach (reduction(+:ke) reduction(+:gpe)) {
136     double norm2 = 0.;
137     foreach_dimension()
138       norm2 += sq(u.x[]);
139     ke += norm2*f[]*dv();
140     gpe += y*f[]*dv();
141   }
142   fprintf(fp, "%g, %e, %e\n", t, rho1*ke/2., rho1*g_*gpe);
143   fclose(fp);
144 }
145
146 //save unordered mesh
147 event vtu(t+=.1, last){
148   char filename[100];
149   sprintf(filename, "%s/TIME-%05.0f", vtu_folder, (t*100));
150   output_vtu((scalar *) {f,p}, (vector *) {u}, filename);
151 }
152
153 event adapt (i++) {
154   adapt_wavelet_leave_interface({u.x, u.y, p},{f},(double[]){uemax, uemax, uemax,
        uemax}, max_LEVEL, LEVEL,padding);
155 }
156
157 event surface_probes(t+=0.01, t<T0){
158   char filename[100];
159   sprintf(filename, "%s/surface_probes.csv", results_folder);
160   FILE *fp = fopen(filename, "a");
161     if (i==0){
162       fprintf(fp, "time");
163       for (int j = 0;j<n_probes;j++){
164         fprintf(fp, ", %f", probe_positions[j]);
165       }
166       fprintf(fp, "\n");
167   }
168   fprintf(fp, "%f", t);
169   heights(f, h);
170   double min_height; //vertical distance from the center of the cell to the air
        water interface
171   double surface_elevation=0;
172   for (int probe_n=0;probe_n<n_probes;probe_n++){
173     min_height=1;
174     foreach(serial){
175       if((fabs(x-probe_positions[probe_n])<Delta/2.0+1e-9)&&(fabs(y)<0.1)){
176         if(h.y[] != nodata){
177           if (fabs(min_height)>fabs(height(h.y[])*Delta)){
178             min_height=height(h.y[])*Delta;
```

```
            surface_elevation = y + height(h.y[])*Delta;
          }
        }
      }
    }
    fprintf(fp, ", %f", surface_elevation);
  }
  fprintf(fp, "\n");
  fclose(fp);
}
```

A.3.3 Multilayer Piston

This is my script for the implementation of a piston in the multilayer solver of basilisk. I made it by further developing my script for the initialised wave based on bar.c Stephane Popinet 2024c and breaking.c Stephane Popinet 2024a.

```
/**
based on bar.c and breaking.c
the parameters for the wave are from
  Jensen, A., Sveen, J. K., Grue, J., Richon, J. B., & Gray, C. (2001).
  Accelerations in water waves by extended particle image velocimetry.
  Experiments in Fluids, 30(5), 500-510. https://doi.org/10.1007/s003480000229
*/

#include "grid/multigrid1D.h"
#include "layered/hydro.h"
#include "layered/nh.h"
#include "layered/remap.h"
#include "layered/check_eta.h"
#include "layered/perfs.h"
#include "output_pvd.h"

int set_n_threads = 8; //set number of threads manually
int LEVEL = 11;       //the grid resolution in x direction Nx = 2**LEVEL

double Tend = 50;     //the end time of the simulation
// double Lx = 24.6; //The length of the simulation domain
double Lx = 24.6;
#define nl_ 10  //the default number of layers if none are given as command line
    arguments
double rmin = 0.5;  //rmin the relative height of the top layer compared to
                    //a regular distribution. the rest fo the layers follow a
    geometric distribution.

double h_ = 0.6;     //water depth

// piston file
int run_number = 1; //default run number if none is given in the command line
    piston files in "piston_files/%run_number/fil3.dat";
int file_samplerate = 100; //the samplerate of the piston speed file
#define piston_timesteps 10000//the max number of timesteps in the piston file
int piston_counter;
double piston_ux[piston_timesteps];
double neumann_pressure[piston_timesteps];
//piston parameters

char results_folder[40]; //the location to save the results
char vts_folder[50]; //the locaton to save the vtu files
//surface probes
double probe_positions[1404] = {1.5,10.048,10.745,11.498};
int n_probes = 1404;


void read_piston_data(){
  char piston_file[40];
  sprintf(piston_file, "piston_files/%d/piston_speed.dat", run_number);
  printf("piston file:%s\n", piston_file);
```

```
49
50    int count = 0;
51    FILE *file;
52    file = fopen(piston_file, "r");
53    if(!file)
54      {
55          perror("Error opening piston position file");
56      }
57    int _running=1;
58    while(_running && count< piston_timesteps ){
59      _running = fscanf(file, "%lf", &(piston_ux[count]));
60      count++;
61    }
62    fclose(file);
63    for (int i=count;i<piston_timesteps;i++){
64      piston_ux[i] = 0; //set the remaining timesteps to 0
65    }
66  }
67
68  event setup_probe_positions(i=0){
69    double probe_x = 0.1;
70    for(int j=4;j<n_probes;j++){
71      probe_positions[j] =probe_x;
72      probe_x = probe_x+0.01;
73    }
74  }
75
76  double Wave_VeloX(double x , double y, double z, double t){
77    int t0_i = (int)floor(t*file_samplerate);
78    //linear interpolation of the piston speed f(t) = f(t0) +(f(t1)-f(t0))*(t-t0)/(
        t1-t0)
79    return piston_ux[t0_i] + (piston_ux[t0_i+1]-piston_ux[t0_i])*(t*file_samplerate-
        t0_i);
80  }
81
82
83
84  event init (i = 0)
85  {
86    #if _OPENMP
87    fprintf(stderr, "current threads:%d\n", omp_get_num_threads());
88    fprintf(stderr, "max number of openmp threads:%d\n", omp_get_max_threads());
89    #endif
90
91    geometric_beta(rmin, true); //set the layer thickness smaller nearer the surface
92    foreach() {
93      zb[] = -h_;
94      foreach_layer(){
95        h[] = (max(- zb[], 0.))*beta[point.l];
96      }
97    }
98
99    # define neumann_pressure_variable(t) (((Wave_VeloX(0, 0, 0, t+dt)-Wave_VeloX(0,
         0, 0, t))/dt))
100   u.n[left] = dirichlet(Wave_VeloX(0,0,0,t));
101   phi[left] = neumann(neumann_pressure_variable(t));
102   u.n[right] = neumann(0.);
103 }
104
105 int main(int argc, char *argv[])
106   {
107     //set LEVEL, layers and run number from command line args
108   nl = nl_;
109   for(int j=0;j<argc;j++){
110     if (strcmp(argv[j], "-L") == 0)
111         {
112             LEVEL = atoi(argv[j + 1]);
113         }
114     if (strcmp(argv[j], "-nl") == 0)
115     {
```

```
      nl = atoi(argv[j + 1]);
    }
    if (strcmp(argv[j], "-r") == 0)
    {
      run_number = atoi(argv[j + 1]);
    }
  }
  //make folders for saving the results
  sprintf(results_folder, "results/run%d/LEVEL%d_layers%d", run_number, LEVEL, nl)
    ;
  sprintf(vts_folder, "%s/vts", results_folder);

  char remove_old_results[100];
  sprintf(remove_old_results, "rm -r %s", results_folder);
  if (system(remove_old_results)==0){
    printf("removed old results in:%s\n", results_folder);
  }

  char make_results_folder[100];
  sprintf(make_results_folder, "mkdir -p %s", vts_folder);
  if (system(make_results_folder)==0){
    printf("made results folder:%s\n", results_folder);
  }

  //copy the script to the results folder for later inspection if needed
  char copy_script[100];
  sprintf(copy_script, "cp multilayer.c %s/multilayer.c", results_folder);
  if (system(copy_script)==0){
    printf("copied script to results folder\n");
  }

  origin (0, h_);

  N = 1<<LEVEL;
  L0 = Lx;
  G = 9.81;
  breaking = 1;
  CFL_H = .1;
  CFL = 0.1;
  rmin = sqrt(Lx*nl/N/h_);
  printf("CFL=%f, CFL_H=%f\n", CFL, CFL_H);
  TOLERANCE = 10e-8;
  //printf("TOLERANCE:%f\n", TOLERANCE);
  theta_H = 0.51;
  nu = 1e-6;
  #if _OPENMP
  if (set_n_threads>0)
  {
  int num_omp = omp_get_max_threads();
  fprintf(stderr, "max number of openmp threads:%d\n", num_omp);
  omp_set_num_threads(set_n_threads);
  fprintf(stderr, "set openmp threads:%d\n", omp_get_num_threads());
  }
  #elif _MPI
  fprintf(stderr, "npe:%d\n", npe());
  #endif
  read_piston_data();
  run();
  }
#if _OPENMP
event output_field (t <= Tend; t += 1)
{
    fprintf(stdout, "field vts output at step: %d, time: %.2f \n", i, t);
    static int j = 0;
    char name[100];
    sprintf(name, "%s/field_%.6i.vts",vts_folder, j++);
    fprintf(stdout, "written to: %s\n", name);
    FILE* fp = fopen(name, "w");
    output_vts_ascii_all_layers(fp, {eta,h,u}, N);
    fclose(fp);
```

```
    #if _OPENMP
    omp_set_num_threads(set_n_threads);
    #endif
}
#endif

 event save_energy (t += 0.01)
{
  char filename[200];
  sprintf(filename, "%s/energy.csv", results_folder);
  static FILE * fp = fopen (filename, "w");
  double ke = 0., gpe = 0.;
  foreach (reduction(+:ke) reduction(+:gpe)) {
    double zc = zb[];
    foreach_layer() {
      double norm2 = sq(w[]);
      foreach_dimension()
        norm2 += sq(u.x[]);
      ke += norm2*h[]*dv();
      gpe += (zc + h[]/2.)*h[]*dv();
      zc += h[];
    }
  }
  if (i == 0)
    fprintf (fp, "t, ke, gpe\n");
  fprintf(fp, "%f, %f, %f\n", t, 997*ke/2., 997*9.81*gpe);
}

event show_progress(i++)
{
  float progress = 0;
  progress = t /Tend;
  printf("t=%.3f, i=%d, dt=%g, run=%d, LEVEL=%d, nl=%d, ", t, i, dt, run_number,
    LEVEL, nl);
  printf("%.2f%%\n", progress*100);
}

event surface_probes(t+=0.01){
  char filename[100];
  sprintf(filename, "%s/surface_probes.csv", results_folder);
  FILE *fp = fopen(filename, "a");
    if (i==0){
      fprintf(fp, "time");
      for (int j = 0;j<n_probes;j++){
        fprintf(fp, ", %f", probe_positions[j]);
      }
      fprintf(fp, "\n");
  }
  fprintf(fp, "%f", t);
  for (int probe_n=0;probe_n<n_probes;probe_n++){

    fprintf(fp, ", %f", interpolate(eta, probe_positions[probe_n],-h_));
  }
  fprintf(fp, "\n");
  fclose(fp);
}
```

A.3.4 Navier Stokes Moving Piston

I made this script by expanding on the script for the initialised wave in the Navier Stokes solver and by using the methods for a moving piston from wave_wall.c Hooft 2018.

```
    /*
 * based on http://basilisk.fr/sandbox/Antoonvh/wave_wall.c
 * can read piston position data from file. Set up for file with 100hz sample rate
 */
#include <sys/stat.h>
#include <stdio.h>
```

```
#include <string.h>
#include "utils.h"
#include "adapt_wavelet_leave_interface_two_levels.h"
#include "heights.h"
#include "output.h"
#include "navier-stokes/centered.h"
#include "two-phase.h"
#include "navier-stokes/conserving.h"
#include "reduced.h"
#include "profiling.h"
#include "output_vtu_foreach.h"

int set_n_threads = 8; //0 to use all available threads for OPENMP
int LEVEL = 4;
int max_LEVEL = 11; //Default level if none is given as command line argument
int padding = 2;
int EXTRA_PISTON_LEVEL = 0; //extra refinement around the piston to make it leak
    less

#define _h 0.6//water depth
double l = 24.6; //the size of the domain, preferable if l=(water_depth*2**LEVEL)/
    n where n is an integer
double g_ = 9.81;
double domain_height = 1.0; //the height of the simulation domain
double femax = 0.2;
double uemax = 0.2;
double pemax = 0.2;
double Tend = 50.;

double probe_positions[144];
int n_probes = 144;
int n_extra_probe = 4;
double extra_probe_positions[]={1.50, 10.04, 10.75, 11.50};

vector h[]; //scalar field of the distance from the surface, using heights.h
char results_folder[40]; //the location to save the results
char vtu_folder[50]; //the locaton to save the vtu files

//piston file
int run_number = 1; //default run number if none is given in the command line
    piston files in "piston_files/%run_number/fil3.dat";
char piston_file[40];
char piston_speed_file[40];
int file_samplerate = 100; //the samplerate of the piston position file
#define piston_timesteps 10000//the number of timesteps in the piston file
int piston_counter;
double piston_positions[piston_timesteps];
double piston_ux[piston_timesteps];
//piston parameters
#define piston_back_wall_offset .1 //the distance from the left of the domain to
    the front of the piston
double piston_height = 0.2;   //height of the piston above the still water level
double piston_bottom_clearance = 0.00; //piston distance above bottom
#define piston_w .2 //width of the piston
#define PISTON Piston_Pos_x(x, y, z, t)-x
scalar pstn[];

void read_piston_data(){
  int count = 0;
  FILE *file;
  file = fopen(piston_file, "r");
  if(!file)
    {
        perror("Error opening piston position file");
    }
  int _running=1;
  while(_running && count< piston_timesteps ){
    _running = fscanf(file, "%lf", &(piston_positions[count]));
    count++;
  }
```

```
73    fclose(file);
74
75    for (int i=count;i<piston_timesteps;i++){
76      piston_positions[i] = 0;
77    }
78    count = 0;
79    file = fopen(piston_speed_file, "r");
80    if(!file)
81      {
82          perror("Error opening piston speed file");
83      }
84    _running=1;
85    while(_running && count< piston_timesteps ){
86      _running = fscanf(file, "%lf", &(piston_ux[count]));
87      count++;
88    }
89    fclose(file);
90
91    for (int i=count;i<piston_timesteps;i++){
92      piston_ux[i] = 0;
93    }
94  }
95
96  double Piston_Velo_x(double x , double y, double z, double t){
97    int t0_i = (int)floor(t*file_samplerate);
98    //linear interpolation of the piston speed f(t) = f(t0) +(f(t1)-f(t0))*(t-t0)/(
        t1-t0)
99    return piston_ux[t0_i] + (piston_ux[t0_i+1]-piston_ux[t0_i])*(t*file_samplerate-
        t0_i);
100 }
101
102 double Piston_Pos_x(double x, double y, double z, double t){
103   int t0_i = (int)floor(t*file_samplerate);
104   return piston_positions[t0_i] + (piston_positions[t0_i+1]-piston_positions[t0_i
        ])*(t*file_samplerate-t0_i);
105 }
106
107 event setup_probe_positions(i=0){
108   int j = 0;
109   for(j=0;j<n_extra_probe;j++){
110     probe_positions[j] = extra_probe_positions[j];
111   }
112   probe_positions[j] = 0.1;
113   j++;
114   for(j=j;j<n_probes;j++){
115     probe_positions[j] = probe_positions[j-1]+0.1;
116   }
117 }
118
119 void mask_domain(){
120   //mask away the top of the domain
121   mask(y > domain_height - _h ? top : none);
122   u.n[top]  = neumann(0.);
123   p[top]    = dirichlet(0.);
124   pf[top]   = dirichlet(0.);
125 }
126
127 int main(int argc, char *argv[]) {
128   //set max_LEVEL, extra piston refinement
129   //and run number from command line args
130   for(int j=0;j<argc;j++){
131     if (strcmp(argv[j], "-L") == 0)
132         {
133             max_LEVEL = atoi(argv[j + 1]);
134         }
135     if (strcmp(argv[j], "-P") == 0)
136         {
137             EXTRA_PISTON_LEVEL = atoi(argv[j + 1]);
138         }
139     if (strcmp(argv[j], "-r") == 0)
```

```
        {
            run_number = atoi(argv[j + 1]);
        }
  }

  //make folders for saving the results
  sprintf(results_folder, "results/run%d/LEVEL%d_%d", run_number, max_LEVEL,
    EXTRA_PISTON_LEVEL);
  sprintf(vtu_folder, "%s/vtu", results_folder);

  char remove_old_results[100];
  sprintf(remove_old_results, "rm -r %s", results_folder);
  if (system(remove_old_results)==0){
    printf("removed old results in:%s\n", results_folder);
  }

  char make_results_folder[100];
  sprintf(make_results_folder, "mkdir -p %s", vtu_folder);
  if (system(make_results_folder)==0){
    printf("made results folder:%s\n", results_folder);
  }

  //copy the script to the results folder for later inspection if needed
  char copy_script[100];
  sprintf(copy_script, "cp moving_piston.c %s/moving_piston.c", results_folder);
  if (system(copy_script)==0){
    printf("copied script to results folder\n");
  }

  sprintf(piston_file, "piston_files/%d/piston_position.dat", run_number);
  sprintf(piston_speed_file, "piston_files/%d/piston_speed.dat", run_number);

  printf("%s\n", piston_file);
  read_piston_data();

  L0 = l;
  rho1 = 997;
  rho2 = 1.204;
  mu1 = 8.9e-4;
  mu2 = 17.4e-6;
  G.y = - g_;
  N = 1 << LEVEL;
  CFL = 0.1;
  DT = l/(1<<(max_LEVEL));
  printf("DT=%.5f\n", DT);
  u.n[bottom] = dirichlet(0.);
  u.t[bottom] = dirichlet(0.);
  u.n[left] = dirichlet(0.);
  u.n[right] = neumann(0.);
  u.n[top] = neumann(0.);
  pstn.refine = pstn.prolongation = fraction_refine;

#if _OPENMP
  int num_omp = omp_get_max_threads();
  fprintf(stderr, "max number of openmp threads:%d\n", num_omp);
  if (set_n_threads){  //set number of omp threads
    omp_set_num_threads(set_n_threads);
  }
  fprintf(stderr, "set openmp threads:%d\n", omp_get_max_threads());
#endif
  run();
}

event init (i = 0) {
  origin(-piston_back_wall_offset, -_h);
  mask_domain();
  fraction (f, - y); //set the water depth _h
  fraction (pstn, PISTON); //set the piston fraction
  while (adapt_wavelet_leave_interface({u.x, u.y, p},{f, pstn},(double[]){uemax,
    uemax,femax,pemax, femax}, max_LEVEL+EXTRA_PISTON_LEVEL, LEVEL,padding, (int
```

```
    []){max_LEVEL, max_LEVEL+EXTRA_PISTON_LEVEL}).nf){  //for adapting more around
     the piston interface
208     fraction (f, - y); //set the water level on the refined mesh
209     fraction (pstn, PISTON); //set the piston fraction on the refined mesh
210   }
211   unrefine ((x < -0.1)&&(level>6));
212   foreach(){
213     pf[] = 0;
214     p[] = pf[];
215   }
216 }
217
218 /*
219 The grid is adapted to keep max refinement at the air water interface.
220 And to minimise the error in the velocity field. Extra grid refinement around
221 the piston is possible.
222  */
223 event adapt (i++){
224   adapt_wavelet_leave_interface({u.x, u.y, p},{f, pstn},(double[]){uemax, uemax,
      pemax, femax, pemax}, max_LEVEL+EXTRA_PISTON_LEVEL, LEVEL,padding, (int[]){
      max_LEVEL, max_LEVEL+EXTRA_PISTON_LEVEL});
225   unrefine ((x>(Piston_Pos_x(x,y,z,t)+0.05))&&(level>=max_LEVEL));
226   unrefine ((x < Piston_Pos_x(x,y,z,t)-piston_w*0.6)); //unrefine the area to the
      left of the piston
227   unrefine ((y<-0.4)&&(x>(Piston_Pos_x(x,y,z,t)+0.02))); //unrefine the bottom
228   unrefine ((y>0.1));
229   fraction(pstn, PISTON);
230 }
231
232
233 /**
234 The moving piston is implemented via Stephane's trick. Which results in a leaking
     piston.
235 */
236 event piston (i++, first) {
237   fraction (pstn, PISTON);
238   foreach(){
239     u.y[] = u.y[]*(1 - pstn[]);
240     u.x[] = pstn[]*Piston_Velo_x(x, y ,z, t) + u.x[]*(1 - pstn[]);
241   }
242   u.n[left]=dirichlet(Piston_Velo_x(x,y,z,t));
243 }
244
245 event surface_probes(t+=0.01){
246   char filename[100];
247   sprintf(filename, "%s/surface_probes.csv", results_folder);
248   FILE *fp = fopen(filename, "a");
249     if (i==0){
250       fprintf(fp, "time");
251       for (int j = 0;j<n_probes;j++){
252         fprintf(fp, ", %f", probe_positions[j]);
253       }
254       fprintf(fp, "\n");
255   }
256   fprintf(fp, "%f", t);
257   heights(f, h);
258   double min_height; //vertical distance from the center of the cell to the air
      water interface
259   double surface_elevation=0;
260   for (int probe_n=0;probe_n<n_probes;probe_n++){
261     min_height=1;
262     foreach(serial){
263       if((fabs(x-probe_positions[probe_n])<Delta/2.0)&&(fabs(y)<0.1)){
264         if(h.y[] != nodata){
265           if (fabs(min_height)>fabs(height(h.y[])*Delta)){
266             min_height=height(h.y[])*Delta;
267             surface_elevation = y + height(h.y[])*Delta;
268           }
269         }
270       }
```

```
    }
    fprintf(fp, ", %f", surface_elevation);
  }
  fprintf(fp, "\n");
  fclose(fp);
}

//save unordered mesh
event vtu(t+=1, last){
  printf("Saving vtu file\n");
  char filename[100];
  sprintf(filename, "%s/TIME-%05.0f", vtu_folder, (t*100));
  output_vtu((scalar *) {f,p,pstn}, (vector *) {u}, filename);
}

event save_energy(t+=0.01)
{
  printf("saving energy\n");
  char filename[200];
  sprintf(filename, "%s/energy.csv", results_folder);
  static FILE * fp = fopen(filename, "w");
  double ke = 0., gpe = 0., volume=0.;
  foreach (reduction(+:ke) reduction(+:gpe) reduction(+:volume)) {
    double norm2 = 0.;
    if (x>Piston_Pos_x(x,y,z,t)){
      foreach_dimension()
        norm2 += sq(u.x[]);
      ke += norm2*f[]*dv();
      gpe += (y+_h/2)*f[]*dv();
      volume += f[]*dv();
    }
  }
  if (i == 0)
    fprintf (fp, "t, ke, gpe, f\n");
  fprintf(fp, "%f, %f, %f, %f\n", t, rho1*ke/2., rho1*g_*gpe, volume);
}

event stop (t = Tend);

event show_progress(i++)
{
  printf("t=%02.3f, i=%04d, dt=%.3g, run:%d, LEVEL=%d, Extra_piston_level=%d\n", t
    , i, dt,run_number, max_LEVEL, EXTRA_PISTON_LEVEL);
}


event profiling (i += 20) {
  static FILE * fp = fopen ("profiling", "w");
  trace_print (fp, 1);
}
```

A.3.5 Navier Stokes Solver Boundary Piston

This script is based on the scripts I made for the initialised wave case and is also based on Stephane Popinet 2024b.

```
/*
 * sets the left boundary condition to the speed of the piston movement
 * reads piston position data from file.
 * Thank you to Oystein Lande for how to implement the boundary conditions
 */
#include <stdint.h>
#include <sys/stat.h>
#include <stdio.h>
#include <string.h>
#include "utils.h"
#include "adapt_wavelet_leave_interface.h"
#include "heights.h"
```

```
#include "output.h"
#include "navier-stokes/centered.h"
#include "two-phase.h"
#include "navier-stokes/conserving.h"
#include "reduced.h"
#include "profiling.h"
#include "output_vtu_foreach.h"

int set_n_threads = 8; //0 to use all available threads for OPENMP
int LEVEL = 6;       //length_of domain/2**LEVEL the largest cells used
int max_LEVEL = 10; //Default level if none is given as command line argument length_of_domain/2**max_LEVEL the size of the smallest cells
int padding = 2;     //How many cells around the air water interface that should keep the highest refinement level

#define _h 0.6//water depth
double l = 24.6; //the size of the domain, preferable if l=(water_depth*2**LEVEL)/n where n is an integer
double domain_height = 1.0; //the height of the simulation domain
double g_ = 9.81;
double femax = 0.2;
double uemax = 0.2;
double pemax = .2;
double Tend = 50.;

char results_folder[40]; //the location to save the results
char vtu_folder[50]; //the locaton to save the vtu files

//surface probes
double probe_positions[1404]; //surface probes that meausre the elevation of the free surface spaced 0.1m apart
int n_probes = 1404;
int n_extra_probe = 4;
double extra_probe_positions[]={1.50, 10.04, 10.75, 11.50}; //extra surface probes at these positions(will be the first probes in the surfacs_probes file)
vector h[]; //scalar field for calculating the distance from the surface, using heights.h

//rampoff
double ramp_start = 0.1;    //where the rampoff of the piston function starts. Meters above the still water level
double ramp_distance = 0.1; //the length of the rampoff zone

//piston file
int run_number = 1; //default run number if none is given in the command line piston files in "piston_files/%run_number/piston_speed.dat";
int file_samplerate = 100; //the samplerate of the piston position file
#define piston_timesteps 10000//the number of timesteps in the piston file
double piston_ux[piston_timesteps];

//pressure calculated from the piston movement for the boundary condition
#define neumann_pressure_variable(i) ((paddle_rampoff(y)*(Wave_VeloX(0,0,0,t+dt)-Wave_VeloX(0,0,0,t))/dt - a.n[i])*fm.n[i]/alpha.n[i])

void read_piston_data(){
  char piston_file[40];

  sprintf(piston_file, "piston_files/%d/piston_speed.dat", run_number);
  //sprintf(piston_file, "piston_speed.dat");
  printf("piston file:%s\n", piston_file);

  int count = 0;
  FILE *file;
  file = fopen(piston_file, "r");
  if(!file)
    {
        perror("Error opening piston position file");
    }
  int _running=1;
  while(_running && count< piston_timesteps ){
```

```
    _running = fscanf(file, "%lf", &(piston_ux[count]));
    count++;
  }
  fclose(file);
  for (int i=count;i<piston_timesteps;i++){
    piston_ux[i] = 0;
  }
}

double paddle_rampoff(double y){
  double ramp = 1;
  if (y>ramp_start){
    ramp = (ramp_start+ramp_distance-y)/ramp_distance;
    if (y>ramp_start+ramp_distance){
      ramp = 0;
    }
  }
  return ramp;
}

double Wave_VeloX(double x , double y, double z, double t){
  int t0_i = (int)floor(t*file_samplerate);
  //linear interpolation of the piston speed f(t) = f(t0) +(f(t1)-f(t0))*(t-t0)/(
    t1-t0)
  return piston_ux[t0_i] + (piston_ux[t0_i+1]-piston_ux[t0_i])*(t*file_samplerate-
    t0_i);
}

event setup_probe_positions(i=0){
  int j = 0;
  for(j=0;j<n_extra_probe;j++){
    probe_positions[j] = extra_probe_positions[j];
  }
  probe_positions[j] = 0.1;
  j++;
  for(j=j;j<n_probes;j++){
    probe_positions[j] = probe_positions[j-1]+0.01;
  }
}

int main(int argc, char *argv[]) {

  //set max_LEVEL, run number and number of threads from command line args
  for(int j=0;j<argc;j++){
    if (strcmp(argv[j], "-L") == 0) //check for command line flag
      {
          max_LEVEL = atoi(argv[j + 1]);
      }
    if (strcmp(argv[j], "-r") == 0)
      {
          run_number = atoi(argv[j + 1]);
      }
    if (strcmp(argv[j], "--nthreads") == 0)
      {
          set_n_threads = atoi(argv[j + 1]);
      }
  }

  //make folders for saving the results
  sprintf(results_folder, "results/run%d/LEVEL%d", run_number, max_LEVEL);
  sprintf(vtu_folder, "%s/vtu", results_folder);

  char remove_old_results[100];
  sprintf(remove_old_results, "rm -r %s", results_folder);
  if (system(remove_old_results)==0){
    printf("removed old results in:%s\n", results_folder);
  }

  char make_results_folder[100];
  sprintf(make_results_folder, "mkdir -p %s", vtu_folder);
```

```
142   if (system(make_results_folder)==0){
143     printf("made results folder:%s\n", results_folder);
144   }
145
146   //copy the script to the results folder for later inspection if needed
147   char copy_script[100];
148   sprintf(copy_script, "cp piston.c %s/piston.c", results_folder);
149   if (system(copy_script)==0){
150     printf("copied script to results folder\n");
151   }
152
153   read_piston_data();
154   L0 = 1;
155   rho1 = 997;
156   rho2 = 1.204;
157   mu1 = 8.9e-4;
158   mu2 = 17.4e-6;
159   G.y = - g_;
160   N = 1 << LEVEL;
161   //TOLERANCE = 1e-4;
162   printf("TOLERANCE:%f\n", TOLERANCE);
163   DT = 1/(1<<(max_LEVEL+1));
164   CFL = 0.1;
165   printf("DT:%f\n", DT);
166
167 #if _OPENMP
168   int num_omp = omp_get_max_threads();
169   fprintf(stderr, "max number of openmp threads:%d\n", num_omp);
170   if (set_n_threads){  //set number of omp threads
171     omp_set_num_threads(set_n_threads);
172   }
173   fprintf(stderr, "set openmp threads:%d\n", omp_get_max_threads());
174 #endif
175   run();
176 }
177
178
179 void mask_domain(){
180   //mask away the top of the domain
181   mask(y > domain_height - _h ? top : none);
182 }
183
184 event init (i = 0) {
185   origin(0, -_h);
186   mask_domain();
187   // boundary conditions
188   u.n[top]  = neumann(0.);
189   p[top]    = dirichlet(0.);
190   u.t[top]  = dirichlet(0.);
191
192   u.n[left] = dirichlet(Wave_VeloX(0,0,0,t)*paddle_rampoff(y));
193   p[left]   = neumann(neumann_pressure_variable(0));
194
195   u.n[bottom] = dirichlet(0.);
196   u.t[bottom] = dirichlet(0.);
197
198   u.n[right]  = neumann(0.);
199
200   fraction (f, - y); //set the water depth _h
201   while (adapt_wavelet_leave_interface({u.x, u.y, p},{f},(double[]){uemax,uemax,
        pemax, femax},max_LEVEL, LEVEL,padding).nf){  //for adapting more around the
        piston interface
202     fraction (f, - y); //set the water level on the refined mesh
203   }
204 }
205
206 /*
207 The grid is adapted to keep max refinement at the air water interface.
208 And to minimise the error in the velocity field.
209  */
```

```
event adapt (i++){
  adapt_wavelet_leave_interface({u.x, u.y, p},{f},(double[]){uemax, uemax, pemax,
    femax}, max_LEVEL, LEVEL,padding);
  unrefine ((y>0.1)); //unrefine the air above the waves. make sure the waves are
    not so high that they get unrefined
                      //this is necessary to avoid problems when air vortexes
    reach the top boundary
}


event surface_probes(t+=0.01){
  char filename[100];
  sprintf(filename, "%s/surface_probes.csv", results_folder);
  FILE *fp = fopen(filename, "a");
    if (i==0){
      fprintf(fp, "time");
      for (int j = 0;j<n_probes;j++){
        fprintf(fp, ", %f", probe_positions[j]);
      }
      fprintf(fp, "\n");
  }
  fprintf(fp, "%f", t);
  heights(f, h);
  double min_height; //vertical distance from the center of the cell to the air
    water interface
  double surface_elevation=0;
  for (int probe_n=0;probe_n<n_probes;probe_n++){
    min_height=1;
    foreach(serial){
      if((fabs(x-probe_positions[probe_n])<Delta/2.0)&&(fabs(y)<0.1)){
        if(h.y[] != nodata){
          if (fabs(min_height)>fabs(height(h.y[])*Delta)){
            min_height=height(h.y[])*Delta;
            surface_elevation = y + height(h.y[])*Delta;
          }
        }
      }
    }
    fprintf(fp, ", %f", surface_elevation);
  }
  fprintf(fp, "\n");
  fclose(fp);
}

//save unordered mesh
event vtu(t+=1, last){
  printf("Saving vtu file\n");
  char filename[100];
  sprintf(filename, "%s/TIME-%05.0f", vtu_folder, (t*100));
  output_vtu((scalar *) {f,p}, (vector *) {u}, filename);
}

event save_energy(t+=0.01)
{
  //printf("saving energy\n");
  char filename[200];
  sprintf(filename, "%s/energy.csv", results_folder);
  static FILE * fp = fopen(filename, "w");
  double ke = 0., gpe = 0.;
  foreach (reduction(+:ke) reduction(+:gpe)) {
    double norm2 = 0.;
    foreach_dimension()
      norm2 += sq(u.x[]);
    ke += norm2*f[]*dv();
    gpe += (y+_h/2)*f[]*dv();
  }
  if (i == 0)
    fprintf (fp, "t, ke, gpe\n");
  fprintf(fp, "%f, %f, %f\n", t, rho1*ke/2., rho1*g_*gpe);
}
```

```
276
277 /*
278 simulation stopped at Tend
279 */
280 event stop (t = Tend);
281
282 event show_progress(t+=1)
283 {
284   printf("t=%02.3f, i=%04d, dt=%.3g\n", t, i, dt);
285 }
286
287 event profiling (i += 20) {
288   static FILE * fp = fopen ("profiling", "w");
289   trace_print (fp, 1);
290 }
```

A.4 3D piston Implementation

The adaptation of the piston implementation to work in 3D was relatively straightforward. Following is how I changed the velocity function to also work in 3D.

```
1
2 double Wave_VeloX(double x, double y, double z, double t){
3   int piston_number = 0;
4   piston_number = (int) floor(z/piston_width);
5   //linear interpolation of the piston speed
6   int t0_i = (int)floor(t*file_samplerate);
7   return piston_ux[piston_number][t0_i] + (piston_ux[piston_number][t0_i+1]-\\
8   piston_ux[piston_number][t0_i])*(t*file_samplerate-t0_i);
9 }
```

This was the only major change to the piston implementation needed to make it work in three dimensions.

Bibliography

Bell, John B, Phillip Colella and Harland M Glaz (1989). 'A second-order projection method for the incompressible navier-stokes equations'. eng. In: *Journal of computational physics* 85.2, pp. 257–283. ISSN: 0021-9991.

De, S. C. (1955). 'Contributions to the theory of stokes waves'. eng. In: *Mathematical proceedings of the Cambridge Philosophical Society* 51.4, pp. 713–736. ISSN: 0305-0041.

Fenton, J.D. (1985). 'A fifth-order Stokes theory for steady waves : Fenton, J.D., 1985. J. WatWay Port coast. Ocean Engng, Am. Soc. civ. Engrs, 111(2):216–234'. eng. In: *Deep-sea research. Part B. Oceanographic literature review* 32.9, pp. 728–729. ISSN: 0198-0254.

Gimse, Martin Funderud (2024). *Lab results from 12/04/2024 and 18/09/2024*. URL: https://github.com/martingim/master/tree/main/matlab/PIV_basilisk/Lab_results (visited on 31/03/2025).

— (2025a). *The Basilisk scripts used in this thesis*. URL: https://github.com/martingim/master/tree/main/basilisk (visited on 14/05/2025).

— (2025b). *Movies and Figures of Basilisk simulations*. URL: https://github.com/martingim/master/tree/main/movies_and_figures (visited on 07/04/2025).

Hooft, Antoon van (2018). *A wave-wall collision experiment*. URL: http://basilisk.fr/sandbox/Antoonvh/wave_wall.c (visited on 04/11/2024).

Hunt, J. N. (1953). 'A NOTE ON GRAVITY WAVES OF FINITE AMPLITUDE'. eng. In: *Quarterly journal of mechanics and applied mathematics* 6.3, pp. 336–343. ISSN: 0033-5614.

Jensen, A. et al. (2001). 'Accelerations in water waves by extended particle image velocimetry'. eng. In: *Experiments in fluids* 30.5, pp. 500–510. ISSN: 0723-4864.

Kolaas, Jostein (2016). *Getting started with HydrolabPIV v1.0*. eng.

Kundu, Pijush K (2016). *Fluid mechanics*. eng. Sixth edition. Amsterdam: Elsevier/AP, pp. 355–364. ISBN: 9780124059351.

Le Mehaute, Bernard (1976). *An introduction to hydrodynamics and water waves*. eng. New York: Springer-Verlag. ISBN: 0387072322.

Levi-Civita, T. (1925). 'Détermination rigoureuse des ondes permanentes d'ampleur finie'. fre. In: *Mathematische annalen* 93.1, pp. 264–314. ISSN: 0025-5831.

Nobach, H and M Honkanen (2005). 'Two-dimensional Gaussian regression for sub-pixel displacement estimation in particle image velocimetry or particle position estimation in particle tracking velocimetry'. eng. In: *Experiments in fluids* 38.4, pp. 511–515. ISSN: 0723-4864.

Øystein Lande, Arne Bockmann (2018). *Adapt wavelet leave interface*. URL: http://basilisk.fr/sandbox/oystelan/adapt_wavelet_leave_interface.h (visited on 23/10/2024).

Perlin, Marc, Wooyoung Choi and Zhigang Tian (Jan. 2013). 'Breaking Waves in Deep and Intermediate Waters'. In: *Annual Review of Fluid Mechanics* 45, pp. 115–145. DOI: 10.1146/annurev-fluid-011212-140721.

Popinet, Stéphane (2003). 'Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries'. eng. In: *Journal of computational physics* 190.2, pp. 572–600. ISSN: 0021-9991.

— (2018). 'Numerical Models of Surface Tension'. In: *Annual Review of Fluid Mechanics* 50.Volume 50, 2018, pp. 49–75. ISSN: 1545-4479. DOI: https://doi.org/10.1146/annurev-fluid-122316-045034. URL: https://www.annualreviews.org/content/journals/10.1146/annurev-fluid-122316-045034.

— (2020). 'A vertically-Lagrangian, non-hydrostatic, multilayer model for multiscale free-surface flows'. In: *Journal of Computational Physics* 418, p. 109609. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2020.109609. URL: https://www.sciencedirect.com/science/article/pii/S0021999120303831.

Popinet, Stephane (2024a). *3D breaking Stokes wave (multilayer solver)*. URL: http://basilisk.fr/src/examples/breaking.c (visited on 29/11/2024).

— (2025a). *Basilisk Tips*. URL: http://basilisk.fr/src/Tips (visited on 31/03/2025).

— (2024b). *Breaking Stokes wave*. URL: http://basilisk.fr/src/test/stokes-ns.c (visited on 29/11/2024).

— (2025b). *Breaking Stokes wave*. URL: http://basilisk.fr/src/test/stokes.c (visited on 31/03/2025).

— (2025c). *Incompressible Navier–Stokes solver (centered formulation)*. URL: http://basilisk.fr/src/navier-stokes/centered.h (visited on 17/03/2025).

— (2025d). *Momentum-conserving advection of velocity*. URL: http://basilisk.fr/src/navier-stokes/conserving.h (visited on 31/03/2025).

— (2024c). *Sinusoidal wave propagation over a bar*. URL: http://basilisk.fr/src/test/bar.c (visited on 29/11/2024).

— (2025e). *Two-phase interfacial flows*. URL: http://basilisk.fr/src/two-phase.h (visited on 06/03/2025).

— (2025f). *Volume-Of-Fluid advection*. URL: http://basilisk.fr/src/vof.h (visited on 06/03/2025).

Raffel, Markus et al. (2018). *Particle Image Velocimetry: A Practical Guide*. eng. 3rd ed. 2018. Cham: Springer International Publishing AG, pp. 219–223. ISBN: 9783319688510.

Rayleigh, Lord (1917). 'XXXVIII. On periodic irrotational waves at the surface of deep water'. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 33.197, pp. 381–389. DOI: 10.1080/14786440508635653. eprint: https://doi.org/10.1080/14786440508635653. URL: https://doi.org/10.1080/14786440508635653.

Skjelbreia L, Hendrickson J. (Jan. 1960). In: *Coastal Engineering Proceedings* 1.7, p. 10. DOI: 10.9753/icce.v7.10. URL: https://icce-ojs-tamu.tdl.org/icce/article/view/2169.

Stokes, George Gabriel (1847). 'On the theory of oscillatory waves'. In: *Trans. Cam. Philos. Soc.* 8, pp. 441–455.

— (2009). 'Supplement to a paper on the Theory of Oscillatory Waves'. eng. In: *Mathematical and Physical Papers*. Cambridge University Press, pp. 314–326. ISBN: 9781108002622.

Struik, D.J. (1926). 'Détermination rigoureuse des ondes irrotationelles périodiques dans un canal à profondeur finie'. fre. In: *Mathematische annalen* 95.1, pp. 595–634. ISSN: 0025-5831.

WILLERT, CE and M GHARIB (1991). 'Digital particle image velocimetry'. eng. In: *Experiments in fluids* 10.4, pp. 181–193. ISSN: 0723-4864.

Zhao, Kuifeng and Philip L.-F. Liu (2022a). 'On Stokes wave solutions'. eng. In: *Proceedings - Royal Society. Mathematical, physical and engineering sciences* 478.2258. ISSN: 1364-5021.

— (Jan. 2022b). *Supplementary material from "On Stokes wave solutions"*. DOI: 10.6084/m9.figshare.c.5777469.v2. URL: https://rs.figshare.com/collections/Supplementary_material_from_On_Stokes_wave_solutions_/5777469/2.

Zhao, Kuifeng, Yufei Wang and Philip L.-F. Liu (2024). 'A guide for selecting periodic water wave theories - Le Méhauté (1976)'s graph revisited'. In: *Coastal Engineering* 188, p. 104432. ISSN: 0378-3839. DOI: https://doi.org/10.1016/j.coastaleng.2023.104432. URL: https://www.sciencedirect.com/science/article/pii/S0378383923001564.