Computational fluid dynamics simulations of gravity wave flows
Comparing finite volume and finite element solvers

Jørgen D. Tyvand
Master’s Thesis, Spring 2017
This master’s thesis is submitted under the master’s programme *Computational Science and Engineering*, with programme option *Mechanics*, at the Department of Mathematics, University of Oslo. The scope of the thesis is 60 credits.

The front page depicts a section of the root system of the exceptional Lie group $E_8$, projected into the plane. Lie groups were invented by the Norwegian mathematician Sophus Lie (1842–1899) to express symmetries in differential equations and today they play a central role in various parts of mathematics.
# Contents

1 Introduction ........................................ 7

2 Mathematical Overview .............................. 10
   2.1 Navier-Stokes Equations .......................... 10
      2.1.1 Incompressible flow ........................... 10
      2.1.2 Boussinesq approximation ...................... 11
      2.1.3 ALE formulation ............................... 11
   2.2 Added mass and damping ........................... 12

3 Numerical Overview .................................. 13
   3.1 Finite volume method .............................. 13
   3.2 Finite element method ............................. 13
   3.3 Numerical methods in OpenFOAM .................... 14
   3.4 Numerical methods in FEniCS ...................... 15

4 Internal Waves ........................................ 16
   4.1 Case description .................................. 16
   4.2 Approach using OpenFOAM .......................... 18
      4.2.1 OpenFOAM solver validation .................... 18
      4.2.2 Problem setup for OpenFOAM .................... 19
      4.2.3 OpenFOAM results ............................. 24
   4.3 Approach using FEniCS ............................. 27
      4.3.1 FEniCS solver validation ....................... 29
      4.3.2 Problem setup for FEniCS ....................... 31
      4.3.3 FEniCS results and comparison ................. 33
   4.4 Discussion ......................................... 37

5 Solitary surface waves ............................... 38
   5.1 Case Description ................................. 38
   5.2 Approach using OpenFOAM .......................... 38
      5.2.1 OpenFOAM solver validation .................... 39
      5.2.2 Problem setup for OpenFOAM .................... 42
      5.2.3 Results ....................................... 42
   5.3 Approach using FEniCS ............................. 44
      5.3.1 FEniCS solver validation ....................... 45
      5.3.2 Problem setup in FEniCS ....................... 50
      5.3.3 Results ....................................... 51
5.4 Comparison and discussion ........................................ 55

6 Wave/body interaction .............................................. 56
  6.1 Case Description .................................................. 56
  6.2 Approach using OpenFOAM ...................................... 56
    6.2.1 Problem Setup in OpenFOAM ................................. 56
    6.2.2 Results ...................................................... 58
  6.3 Approach using FEniCS .......................................... 60
    6.3.1 Results ...................................................... 61
  6.4 Comparison and discussion ..................................... 65

7 Conclusion and future work ....................................... 66

A Selected FEniCS code for internal wave case ...................... 68

B Selected FEniCS code for solitary wave case ..................... 69

C Selected FEniCS code for oscillating object case.................. 70

Bibliography ................................................................ 71
Abstract

OpenFOAM and FEniCS are two solver frameworks for solving computational fluid dynamics problems. The former employs the finite volume method to discretize the governing equations, while the latter employs the finite element method.

In this thesis we compare models using these two frameworks for three different wave problems. These are internal solitary waves, solitary surface waves, and the generation of waves due to an object oscillating in heave.

Before running each specific case, the solver models are validated using benchmark cases or intuitive test cases, to make sure that we can rely on the numerical results to a fairly high degree.

The internal wave case simulations are based on and are compared to published results. We find a good agreement in our results to the published results.

For the solitary wave case, we analyze conservation of mass and energy for a solitary propagating wave with analytically constant shape and velocity. Energy and mass is found to be conserved better in OpenFOAM than in FEniCS, but with a converging improvement in FEniCS with mesh refinement.

In the final case chapter, the added mass and damping coefficients for an object oscillating in heave are computed and compared with analytical results. The FEniCS simulations do unfortunately crash after some period of time, but the results are still consistent with results found in OpenFOAM. Both frameworks give results that are in a comparable range with the analytical results.
Acknowledgements

I would like to thank my supervisors, Associate Professor Mikael Mortensen and M.Sc. Tormod Landet at UiO, for their guidance, ideas and time during the writing of this thesis.

I would also like to thank my fellow students, both the classes of 2016 and 2017, for our brainstorming, discussions, banter and countless cups of coffee. You have all contributed to a great environment, both educationally and socially.

Finally I want to thank my family for their love and support, and especially my girlfriend Elise and dog Sonny for motivating me and pushing me to do my best.
Chapter 1

Introduction

As a student at the Department of Mechanics at the University of Oslo, one is mainly taught two different approaches for solving partial differential equations (PDE’s) and Computational Fluid Dynamics (CFD) problems. These are the Finite Element (FEM) and Finite Volume (FVM) methods. For the application of these methods to PDE’s and CFD problems, the two software platforms one is introduced to in courses given at UiO are OpenFOAM and FEniCS.

The motivation for this thesis is to give a comparison of the two software platforms when applied to various wave motion problems. We will look at ease of use in regards to case setup and user control over parameters, equation discretization and application of boundary conditions. A deeper understanding of the behind-the-scenes workings of OpenFOAM will not be required (and has not been needed for the writing of this thesis), and OpenFOAM has in general been treated as a black box when it comes to the details of solver algorithms and their parameters. As FEniCS is a much more hands-on framework, requiring far more explicit user-defined setup of the equations, parameters and algorithms, we will spend some time discussing the approaches made in FEniCS in relation to each specific case.

OpenFOAM is an open-source, C++ based programming toolbox for solving, among other things, CFD problems. It natively includes a wide array of optimized solver libraries, ranging from multiphase solvers, heat transfer solvers and electromagnetic solvers, to name a few. The basic structure of each case folder is the same, giving intuitive and easy user control over the case setup. The software comes with a large number of tutorials (at least one for each included solver), that give a nice understanding of each solver, as well as a folder and file structure that can be copied and adapted to other similar problems. The choice of using advanced features like adaptive time stepping, corrector step control for algorithms, discretization scheme choices etc. are easily available to the user, but there is a very steep learning curve for features beyond basic case setup. With a deeper understanding, the user can create fully customized solvers for a specific problem, including processing functions like logging of energy etc.
FEniCS is a framework for solving partial differential equation (PDE) problems using the finite element method. For the implementations in this thesis, the Python version of FEniCS has been used. The syntax and file structure is largely similar to using Python for other scientific programming. In contrast to OpenFOAM, almost all of the control in regards to discretization, runtime mechanics and solver control has to be specifically detailed by the user in the program. Basic calls like `solve` do have default functionality, but should in almost all problems be specified further based on the problem one wants to solve.

An in-depth discussion of Python programming will not be discussed in this text, but the reader is referred to the Python website (www.python.org) or the excellent book by Langtangen [8] for a more extensive introduction to programming in Python. A nice collections of demos giving a good introduction to the basics of forms and equation setup in FEniCS can be found on the FEniCS Project demo pages [4].

Intuitively one would assume that OpenFOAM will be the most robust framework out of the box. By this I mean that when applying fairly basic, yet optimized built-in solvers in OpenFOAM, while using a naive approach in FEniCS (without advanced features like stabilization, upwinding, discontinous elements or other types of elements), one will expect OpenFOAM to both run faster and more stable, with less chance of crashing, blowups of solution etc. For each of the three cases, we will therefore first look at the numerical solutions given by OpenFOAM, and compare them to published material, mathematical and physical assumptions, or analytical solutions. Following this we will then see if our approach using FEniCS can reproduce and/or match the results obtained in OpenFOAM.

We will be comparing the two solvers for three different wave problems. These are internal solitary waves due to density differences, propagating solitary surface waves, and waves propagating away from a body oscillating in heave. Comparable features are for example runtime, mesh resolution requirements and property conservation.

The general layout of this thesis is as follows:

Chapter 2 gives a short overview of the governing equations and mathematical properties relevant for the cases studied in this text. The concept of ALE and the definition of added mass and damping is also shortly touched upon in this chapter.

Chapter 3 gives a short explanation of FVM and FEM, as well as a quick look at the way PDE’s are written and discretized using these methods. Numerical discretization in OpenFOAM and FEniCS are also described shortly in this chapter.

Chapter 4 looks at the first wave simulation problem, an internal solitary wave created by density variations in the fluid, and propagation of this wave along a wave tank.
Chapter 5 discusses the application of OpenFOAM and FEniCS to the propagation of a solitary wave propagating on the water surface, both using a two phase approach (OpenFOAM), and an approach looking at the deformation of the domain due to the movement of the fluid (ALE).

Chapter 6 looks at an object oscillating in heave, creating waves propagating along the wave surface away from the object. The approach for each solver is similar to the ones in chapter 5, but we also introduce mesh dynamics in OpenFOAM, and smoothing of the mesh velocity in FEniCS.

Finally in chapter 7 we will give a short conclusion and discussion of further work.
Chapter 2

Mathematical Overview

2.1 Navier-Stokes Equations

The Navier-Stokes equations [16] consist of the equation for the balance of momentum

\[ \rho \left( \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} \right) = -\nabla p + \nabla \cdot \tau + \rho g \]  

(2.1)

where \( \tau = \mu (\nabla \mathbf{u} + (\nabla \mathbf{u})^T) \) and with T denoting the transposed of \( \nabla \mathbf{u} \).

We also have an equation for the conservation of mass (the continuity equation)

\[ \frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \]  

(2.2)

where \( \mathbf{u} \) is the velocity vector field \( \left[ \frac{m}{s} \right] \), \( p \) is pressure \( \left[ \frac{kg}{ms^2} \right] \), \( \rho \) is the fluid density \( \left[ \frac{kg}{m^3} \right] \), \( \mu \) is the dynamic viscosity \( \left[ \frac{kg}{ms} \right] \) and \( g \) is the acceleration due to gravity \( \left[ \frac{m}{s^2} \right] \).

2.1.1 Incompressible flow

For an incompressible flow, where the density \( \rho \) is constant, these equations reduce to the well known incompressible Navier-Stokes equations

\[ \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} = -\frac{1}{\rho} \nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{g} \]  

(2.3)

\[ \nabla \cdot \mathbf{u} = 0 \]  

(2.4)
where \( \nu = \frac{\mu}{\rho} \) is the kinematic viscosity \( \left[ \frac{m^2}{s} \right] \).

### 2.1.2 Boussinesq approximation

For the OpenFOAM solution for internal waves, I have implemented the Boussinesq approximation for buoyancy. This approximation ignores variations in density, except for in the buoyancy term where the acceleration of gravity \( g \) appears. Using equation (2.1) and replacing the variable density with a constant density \( \rho_0 \) on the left hand side, and defining \( \rho = \rho_0 + \Delta \rho \) (where \( \Delta \rho \) is the difference in density), this can be rewritten as

\[
\rho_0 \left( \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} \right) = -\nabla p + \mu \nabla^2 \mathbf{u} + (\rho_0 + \Delta \rho) \mathbf{g} \tag{2.5}
\]

Finally defining \( \Delta \rho \mathbf{g} = (\rho - \rho_0) \mathbf{g} = -\rho_0 \beta (T - T_0) \mathbf{g} \), we end up with the temperature dependent equation

\[
\rho_0 \left( \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} \right) = -\nabla p + \mu \nabla^2 \mathbf{u} + \rho_0 \mathbf{g} - \rho_0 \beta (T - T_0) \mathbf{g} \tag{2.6}
\]

Here \( \beta \) is the coefficient for thermal expansion \( \left[ \frac{1}{\degree C} \right] \), and \( T \) and \( T_0 \) is temperature and some reference temperature \( \degree C \), respectively.

The equation for heat flow is also implemented

\[
\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \alpha \nabla^2 T \tag{2.7}
\]

Here \( \alpha \) is the thermal diffusivity \( \left[ \frac{m^2}{s} \right] \).

### 2.1.3 ALE formulation

The two main field specifications for fluid flow are the Lagrangian and Eulerian field specifications. In the Lagrangian specification, individual fluid particles are followed through space and time. In the Eulerian specification however, one looks at the fluid flow over time through a fixed point. A well used analogy for the different specifications is being in a boat floating down a river, vs. being on a pier and observing the flow of the river past the pier.

The Arbitrary Lagrangian Eulerian formulation is a hybrid of the two specifications. We define a mesh velocity \( \mathbf{w} \) that we can incorporate into the Navier-Stokes
equations, giving them the form of equations (2.8) - (2.9). As we can see, the convective term vanishes in those locations where the mesh velocity and the fluid velocity are the same, like at walls with a set zero velocity for both fluid and mesh, or on moving surfaces where the mesh follows the fluid movement. Here the mesh follows the Lagrangian specification, but inside the fluid domain it has a smoothly and arbitrarily changing specification, hence the name. For a more extensive description of ALE applied to the Navier-Stokes equations, see for example [2].

\[
\frac{\partial \mathbf{u}}{\partial t} + ((\mathbf{u} - \mathbf{w}) \cdot \nabla) \mathbf{u} = -\frac{1}{\rho} \nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{g} \tag{2.8}
\]

\[
\nabla \cdot \mathbf{u} = 0 \tag{2.9}
\]

2.2 Added mass and damping

The following section is a short description of added mass and damping for bodies moving near or on the free surface. For a more in-depth look we refer to the book by Newman [11].

When a body moves on or near the free surface, forces are generated as a result of the body motion. These forces can be expressed as in equation (2.10)

\[
F_i = Re \left\{ \sum_{j=1}^{6} \xi_j e^{i\omega t} f_{ij} \right\} \tag{2.10}
\]

Here \( \xi_j \) is the displacement of the body, \( \omega \) is the angular velocity, and \( f_{ij} \) can be expressed as in (2.11)

\[
f_{ij} = \omega^2 A_{ij} - \omega B_{ij} \tag{2.11}
\]

\( A_{ij} \) and \( B_{ij} \) are referred to as the added mass and damping coefficients, proportional to the acceleration and velocity of the body, respectively.

Added mass is the result of the object accelerating through the surrounding fluid and thus displacing this fluid to make room for the object. This is modeled as an added fluid volume moving along with the object.

Damping is the result of waves being generated by the motion of the object, and these wave radiate outwards, transporting energy away from the object. The damping coefficient is also proportional to the amplitude of the generated waves.
Chapter 3

Numerical Overview

3.1 Finite volume method

The following short overview of the finite volume method is largely taken from the book by Versteeg and Malalasekera [15].

Using the diffusion equation for a given property \( \phi \) as an example, we can write it as

\[
\nabla \cdot (\Gamma \nabla \phi) + S = 0
\]

(3.1)

where \( \Gamma \) is the diffusion coefficient and \( S \) is a source term.

The equation is integrated over a control volume, and we apply Gauss’ Theorem to the laplacian term

\[
\int_{CV} \nabla \cdot (\Gamma \nabla \phi) \, dV + \int_{CV} S \, dV = \int_{A} (\Gamma \nabla \phi) \cdot \mathbf{n} \, dA + \int_{CV} S \, dV = 0
\]

(3.2)

where \( A \) is the area of the control volume surface, and \( \mathbf{n} \) is the normal vector orthogonal on the volume surface.

3.2 Finite element method

Here we will give a short overview of the Finite Element Method. I have based this overview on the lecture notes by Logg and Mardal [9].

The general goal of the Finite Element Method is to derive a weak (or variational) formulation of the given problem. One does this by multiplying the governing equations with a test function \( v \) and integration over the domain. Using the Laplacian term in the Navier-Stokes equations 2.1 as an example, this term is now given as
CHAPTER 3. NUMERICAL OVERVIEW

\[ - \int_{\Omega} \nabla^2 uv \, d\Omega \quad (3.3) \]

One desires in general to use integration by parts on the Laplace term, giving us the following decomposition in the domain and on the boundary

\[ - \int_{\Omega} \nabla^2 uv \, d\Omega = \int_{\Omega} \nabla u \cdot \nabla v \, d\Omega - \int_{\partial \Omega} \frac{\partial u}{\partial n} v \, ds \quad (3.4) \]

Here \( \partial \Omega \) is the boundary of the domain \( \Omega \).

This is done to introduce the boundary terms for application of boundary conditions, symmetry of the matrix in the domain integral term, and the fact that the second derivative of linear elements will be zero. Simplification can be done if one has given Dirichlet or Neumann boundary conditions for the specified problem that make the boundary term vanish on certain boundaries.

### 3.3 Numerical methods in OpenFOAM

To give a basic example of the equation solution structure in OpenFOAM, we look at the icoFoam.C file in the icoFoam solver library, one of the simplest incompressible solvers. The solving of the equation

\[ \frac{\partial u}{\partial t} + \nabla \cdot \phi u - \nabla \cdot \nu \nabla u = -\nabla p \quad (3.5) \]

is represented by the code in listing 3.1

```c
fvVectorMatrix UEqn
(
    fvm::ddt(U)
    + fvm::div(phi, U)
    - fvm::laplacian(nu, U)
);
solve (UEqn == -fvc::grad(p));
```

Here we use a high-level language approach to the discretization. The term \( \text{ddt} \) implies temporal derivation, while the \( \text{fvm} \) and \( \text{fvc} \) terms are implicit and explicit operations. The explicit parts can be solve directly, while the implicit parts must be combined and solved for in a matrix object.
3.4 Numerical methods in FEniCS

A coupled representation (solving the full equation set in one operation) of the incompressible Navier-Stokes equations (and considering a problem where all boundary integrals vanish due to boundary conditions) can be written as a form in FEniCS as in listing 3.2

Listing 3.2: FEniCS form example

\[
F = \text{inner}(u-u_0,v)\,dx + \text{inner}(\dot{\text{grad}}(u)\cdot u_0,v)\,dx - (1./\rho)\,\text{inner}(\text{div}(v),p)\,dx \\
+ \nu\,\text{inner}(\text{grad}(u),\text{grad}(v))\,dx - \text{inner}(g,v)\,dx - \text{inner}(\text{div}(u),q)\,dx
\]

Here \( v \) and \( q \) are the test functions in the velocity and pressure spaces, respectively, and we have employed integration by parts on the pressure and Laplace terms for purposes of symmetry and linearity.
Chapter 4

Internal Waves

4.1 Case description

As opposed to the more commonly studied surface waves, internal waves are, as the name suggests, waves that are generated and propagate within the fluid itself. Variations in density due to variations in temperature and/or salinity can create a density stratification in a fluid, and this results in an acceleration due to the buoyant restoring force. The study of internal waves has implications in several fields relevant to industry, like installations with floating risers or ones moored to the seabed in coastal areas with stratified water regions, as well as submerged structures like floating tunnels.

For the purpose of comparison with published results, I have chosen to implement one of the cases in the paper by Deborah Wood and John Grue [17]. This paper is in turn based on experiments done in the hydrodynamics lab at the University of Oslo, covered extensively in the papers by Grue et. al. [6, 7].

The original experimental setup consists of a wave tank of length 18.4 m, filled with two layers of water with different densities. The bottom layer of brine has height \( h_1 = 0.62 \) m and a density of \( \rho_0 = 1022 \) kg/m\(^3\). The top layer has a linearly stratified density ranging from \( \rho_0 \) to \( \rho_1 = 999 \) kg/m\(^3\) and is of height \( h_2 = 0.15 \) m.

In [17] they have simulated the internal wave generation and propagation by solving for temperature instead of density. By using a temperature difference of 69.25°C between the brine (\( \rho_0 \)) and the fresh water (\( \rho_1 \)) as stated in [17], we can use the Boussinesq approximation given in section 2.1.2 to solve for temperature. The given temperature difference might seem unphysical when considering what temperatures one would expect to have in a wave tank (or even more so, in an actual body of water), but we have to keep in mind that the physical density variations in the tank are based on salinity and not temperature. The range of temperatures is taken from 5°C to 72.95°C, as this is the approximate temperatures for the given densities in the brine and fresh water.

In the experiments a gate is lowered at \( x = 0.5 \) m, and a set volume of fresh
4.1. CASE DESCRIPTION

Figure 4.1: Profile of experimental setup for internal solitary wave case (taken from [17]). The left hand side denotes points of change in density, while the density notations for \( \rho_0 \) and \( \rho_1 \) denote the areas of these density values. The angular dotted areas denote areas of stratified density.

Water of density \( \rho_1 \) and depth \( d \) is then added to the left hand side of the gate. The experiment starts when the gate is raised, and a depression wave is created due to the variations in density. In the numerical simulations, the gate is not actually implemented, but is assumed removed instantaneously at the initialization of the simulation.

For the purposes of this text, the case used for comparison is case C in [17], with a fresh water volume of 77.5 l (\( d = 0.31 \) m) added on the left hand site. Figure 4.1 shows the profile of the experimental setup (taken from [17]).

For the experimental runs, the wave velocity was captured with a camera placed at 15.31 m from the start of the wave tank. Probing functions in OpenFOAM and FEniCS have been implemented to capture the velocity at the corresponding vertical line. Scaled velocity profiles for 4 different instances have been plotted in [17]. A time 0 is defined as the moment of maximum amplitude at the position of the camera, the 3 others instances are 5 and 1 seconds before and 3 seconds after this "time 0". In [17] the maximum amplitude is taken to correspond with the vertical velocity in the wave being zero. I have computed the norm between the vertical velocity and a zero vector (for timesteps within the range of the passing of the wave), and chosen the time 0 to be when this norm is the smallest among the timesteps probed. This also corresponds well with the time of maximum horizontal velocity across the probe line.

For visualization and comparison of the results I have scaled my results with the wave speed \( c \), using the ratio of \( \frac{c}{c_0} = 1.6 \) found in [17]. \( c_0 \) is the linear long wave speed, that can be computed using the equation

\[
\frac{N_0 h_2}{c_0} \cot \frac{N_0 h_2}{c_0} + \frac{h_2}{h_1} = 0
\]  

(4.1)
One can see [7] for a thorough discussion on this equation and its parameters. As stated in [17], $N_0 h_2 = 1.711$ for $h_1 = 0.62$ m and $h_2 = 0.15$ m. I have chosen to use this value and the above equation in my computations, without further examination on my part as to their derivation or accuracy.

4.2 Approach using OpenFOAM

The buoyancy Boussinesq approximation is implemented in a native solver framework in OpenFOAM called buoyantBoussinesqPimpleFoam. This solver is a transient solver for buoyant, incompressible flows, with specified equations for velocity, temperature and pressure that mirror the equations in section 2.1.2.

4.2.1 OpenFOAM solver validation

To validate both the OpenFOAM and FEniCS solvers, we implement a demo case given in [1]. The setup is a rectangular box with a 1:8 width-to-height ratio, filled with air and insulated at the top and bottom, and with prescribed temperatures at the side walls. We have an initial temperature of 0 in the interior, and the prescribed wall temperatures are given as $+T$ on the left hand side and $-T$ on the right.

To compare with the results in the paper, we probe the velocity at a given point (0.181,7.37) defined in [1], and compute the average horizontal velocity $\bar{u}$ over a certain time period after the motion is stabilized. This is then compared with contributed results from other solvers given in [1]. Probing in OpenFOAM is easily defined in the controlDict file, by specifying the field one wants to probe, in this case velocity, as well as defining the point or points one wants to probe at. Python scripts for plotting the results from the probe files have been written by me, and an example of this can be found in the appendices.

The equations in [1] are scaled (see section 4.3), and OpenFOAM solves (without having to do extensive editing to the solver) for unscaled equations. We therefore need to compute the value of $\Delta T$ needed to fulfill the criteria of $Ra = 3.4 \times 10^5$ required in the paper. Using values for thermal diffusivity and expansion at 0°C (found for example in [3]), we get that $\Delta T = 0.00233637^\circ C$. As the velocity scale $U = \sqrt{g\beta W \Delta T}$ and the time scale $\tau = W/U$, we find that the real-time duration for a non scaled simulation will be much larger than for a scaled simulation when $\Delta T$ is small. In fact, for this case, $\tau \approx 110$s, so our duration time for OpenFOAM will be 110 times longer than for the FEniCS solver. Conversely, our time step can be 110 times larger in OpenFOAM to compute a solution with the same temporal precision. After some trial and error I ended up with a (non-scaled) duration of 5 hours (18000 seconds) to get a fairly stable velocity oscillation at the probe point. This equals roughly 160 scaled time units, which is the value that will be used in the FEniCS solver validation. The distribution of temperature in the tank at the final time is shown in figure 4.2.
4.2. APPROACH USING OPENFOAM

Figure 4.2: Distribution of temperature $T$ for OpenFOAM benchmark at end time. The temperature is set to $\pm 0.00117^\circ C$ at the left and right walls, with isolated top and bottom walls.

Plotting the horizontal velocity over time (scaled with $U$ and $\tau$ for comparison with results and FEniCS) as shown in figure 4.3, we see that the velocity stabilizes in oscillations around a somewhat constant value. We see that the signal oscillates in packets, while one gets the impression from [1] that one should achieve a constant "smooth" oscillating motion. Whether this is a resolution issue, or if in fact the packet wave motion is correct will not be further speculated on here. An even longer runtime might be a fix to this though.

Finally, computing the average horizontal velocity at the probe point over the last 40 scaled time units, we get a value of $0.057275$, which is a very good result compared to the results in [1].

4.2.2 Problem setup for OpenFOAM

Meshing of a rectangular domain in OpenFOAM is pretty straightforward, once one learns the basics of the `blockMeshDict` file and how patches (rectangles making up the mesh) are denoted. OpenFOAM meshes are always 3-dimensional, but we can easily make a domain for quasi-2D computation by only having one block in the depth direction.

The mesh is graded towards the top for this case, meaning that the height of each volume is smaller at the top than at the bottom. This is to capture more information at the top without increasing the total number of cells, and is also a natural choice to make as most of the wave generation and motion happens in the top half of the wave tank.

The initial distribution of the temperature field was created using an add-on called `funkySetFields` (part of swak4Foam (https://openfoamwiki.net/index.php/Contrib/swak4Foam) [14]), as manual field manipulation in native OpenFOAM is tedious.
CHAPTER 4. INTERNAL WAVES

Figure 4.3: Scaled horizontal velocity $u/U$ over scaled time $t/\tau$ at probe point $(0.181, 7.37)$ for OpenFOAM benchmark case. $U = \sqrt{g\beta W \Delta T}$ and $\tau = W/U$ where $W$ is the characteristic width of the domain.

an earlier iteration of this case (done as a final presentation in the course MEK4470 - Computational Fluid Mechanics), the fields were created using a home-made Python script. This script had to be specifically written for each mesh resolution, to take into account the grading and size of the mesh. The number of cells for the stratification had to be counted manually, and the fact that the grading gives rise to two different cell counts for the two stratified areas on each side of the gate did not make matters easier.

The switch to funkySetFields solves this problem, as the setting of fields is based on functions of coordinates, so the grading and resolution does not need to be taken into account by the user. The only tricky part is to find the linear equation for the stratified areas, so as to get the correct stratification from the lower to higher temperature. But this is again not dependent on the grading or resolution. Figure 4.4 shows the initial temperature field in the wave tank at the left hand (gate side) of the tank.

The problem has been simulated on a mesh of size 368x150 for the comparison with [17], using a grading of 0.3 (The top cell being 0.3 times the height of the bottom cell). The case has been run using the built in linearUpwind discretization scheme, which is a second order scheme.

To capture the velocity profile at $x = 15.31$ m we use the included probing function in OpenFOAM, and write the velocity to file every 0.1 seconds. The time
4.2. APPROACH USING OPENFOAM

step has been set to 0.01 s, with the adjustable time step functionality turned off. The runtime is set to 25 seconds after some trial and error, to ensure that the wave passes the probe point as well as having results for plotting 3 seconds after time "0".

Two important decisions that had to be made for the simulations was what boundary condition to use for the velocity at the top, as well as whether one should include viscosity in the model. I have done simulations with both a no-slip condition on all walls (both vertical and horizontal velocity set to 0), as well as a slip condition on the top (vertical velocity set to 0). [6] and [7] employ the slip condition (they refer to it as a rigid wall condition), but there is no clear choice made in Wood/Grue.

Wood/Grue has also omitted viscosity from their simulations, but we cannot use a zero-valued viscosity in OpenFOAM (although one can set it to a very low number, say $10^{-35}$ m$^2$/s). Also, FEniCS uses parameters scaled with viscosity, so that model should employ a given viscosity.

I have done runs in OpenFOAM with both a close to zero value and the normal value of $10^{-6}$ m$^2$/s. Figures 4.5-4.8 shows the velocity profiles for the linearUpwind solver using slip and no-slip with low and "normal" viscosity. As we can see, the results are fairly similar for the four cases, but I have chosen to use the slip condition (as per [6] and [7]) and "normal" viscosity for the comparisons with [17] and FEniCS.
CHAPTER 4. INTERNAL WAVES

Figure 4.5: Comparison of horizontal velocity $u$ scaled by wave speed $c$ over height $y$ scaled by top layer height $h_2$ for the four boundary and viscosity choices in OpenFOAM at time $-5s$.

Figure 4.6: Comparison of horizontal velocity $u$ scaled by wave speed $c$ over height $y$ scaled by top layer height $h_2$ for the four boundary and viscosity choices in OpenFOAM at time $-1s$. 
4.2. APPROACH USING OPENFOAM

Figure 4.7: Comparison of horizontal velocity $u$ scaled by wave speed $c$ over height $y$ scaled by top layer height $h_2$ for the four boundary and viscosity choices in OpenFOAM at time 0s.

Figure 4.8: Comparison of horizontal velocity $u$ scaled by wave speed $c$ over height $y$ scaled by top layer height $h_2$ for the four boundary and viscosity choices in OpenFOAM at time 3s.
4.2.3 OpenFOAM results

Figures 4.9-4.10 shows a comparison between the results found in [17] for case C and the computed values from OpenFoam for the four different time steps. From the plots we can see that the numerical results from OpenFOAM correspond well with the published results.

(a) Horizontal velocity $u$ scaled with wave speed $c$ over height $y$ scaled by top layer height $h_2$ at $x = 15.31$ m for time -5s in Wood/Grue [17]

(b) Horizontal velocity $u$ scaled with wave speed $c$ over height $y$ scaled by top layer height $h_2$ at $x = 15.31$ m for time 0s -5s for OpenFOAM solution

Figure 4.9: Comparison of scaled velocity profiles for Wood/Grue and OpenFOAM results at -5s
4.2. APPROACH USING OPENFOAM

(a) Horizontal velocity $u$ scaled with wave speed $c$ over height $y$ scaled by top layer height $h_2$ at $x = 15.31$ m for time -1s in Wood/Grue [17] (left axis label not included in paper)

(b) Horizontal velocity $u$ scaled with wave speed $c$ over height $y$ scaled by top layer height $h_2$ at $x = 15.31$ m for time -1s for OpenFOAM solution

Figure 4.10: Comparison of scaled velocity profiles for Wood/Grue and OpenFOAM results at -1s
(a) Horizontal velocity $u$ scaled with wave speed $c$ over height $y$ scaled by top layer height $h_2$ at $x = 15.31$ m for time 0s in Wood/Grue [17]

(b) Horizontal velocity $u$ scaled with wave speed $c$ over height $y$ scaled by top layer height $h_2$ at $x = 15.31$ m for time 0s for OpenFOAM solution

Figure 4.11: Comparison of scaled velocity profiles for Wood/Grue and OpenFOAM results at 0s
4.3. APPROACH USING FEniCS

(a) Horizontal velocity $u$ scaled with wave speed $c$ over height $y$ scaled by top layer height $h_2$ at $x = 15.31$ m for time 3s in Wood/Grue [17] (left axis label not included in paper)

(b) Horizontal velocity $u$ scaled with wave speed $c$ over height $y$ scaled by top layer height $h_2$ at $x = 15.31$ m for time 0s 3 for OpenFOAM solution

Figure 4.12: Comparison of scaled velocity profiles for Wood/Grue and OpenFOAM results at 3s

4.3 Approach using FEniCS

Selected code for the FEniCS implementation of this case can be found in Appendix A.
For the implementation in FEniCS, we will use the scaled equation set taken from [1]. The equation set is given as

\[
\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} = -\nabla P + \frac{Pr}{Ra} \nabla^2 \mathbf{u} + j \theta \tag{4.2}
\]

\[
\nabla \cdot \mathbf{u} = 0 \tag{4.3}
\]

\[
\frac{\partial \theta}{\partial t} + (\mathbf{u} \cdot \nabla) \theta = \frac{1}{\sqrt{RaPr}} \nabla^2 \theta \tag{4.4}
\]

Here $Ra$ is the Rayleigh number

\[
Ra = \frac{g \beta \Delta TW^3}{\nu \alpha} \tag{4.5}
\]

and $Pr$ is the Prandtl number

\[
Pr = \frac{\nu}{\alpha} \tag{4.6}
\]

where $P$ is the deviation from hydrostatic pressure, $\alpha$ is thermal diffusivity, $\beta$ is thermal expansion, $\Delta T$ is the difference between wall temperatures, $W$ is a characteristic length $[m]$, $\nu$ is kinematic viscosity and $j$ is the unit vector in the $y$-direction. These equations have been scaled by using characteristic length $W$, buoyancy velocity scale $U = \sqrt{g \beta W \Delta T}$, time scale $\tau = W/U$ and pressure scale $P = \rho U^2$.

The non-dimensional temperature $\theta$ is given as

\[
\theta = \frac{T - T_r}{T_h - T_c} \tag{4.7}
\]

where $T_h$ and $T_c$ are the max and min temperatures (the values at top and bottom), and $T_r$ is some reference temperature.

One way to implement the equations (4.2) - (4.4) in FEniCS, is to use a mixed function space to solve the entire equation set in a coupled solver algorithm. This is a simple and robust approach, but at the expense of computation time due to solving for several unknowns at the same time.

A faster but more complex approach is to use a splitting scheme. The one implemented in FEniCS for this case is a mixture between a Chorin scheme [5] and a Crank-Nicholson scheme [10]. The scheme consists of four steps given below, written out in finite element method form.
4.3. APPROACH USING FENICS

First one computes a tentative velocity $\mathbf{u}^*$, using the solutions for the two previous timesteps, $\mathbf{u}^0$ and $\mathbf{u}^{-1}$ (both initialized as zero throughout the domain)

\[
\frac{1}{2\Delta t} \int_{\Omega} \left( 3\mathbf{u}^* - 4\mathbf{u}^0 + \mathbf{u}^{-1} \right) \mathbf{v} \, d\Omega + \int_{\Omega} \left( \(2\mathbf{u}^0 - \mathbf{u}^{-1}\) \cdot \nabla \mathbf{u}^* \right) \mathbf{v} \, d\Omega \\
+ \frac{P_r}{Ra} \int_{\Omega} \nabla \mathbf{u}^* : \nabla \mathbf{v} \, d\Omega - \frac{P_r}{Ra} \int_{\partial \Omega} \nabla \mathbf{u}^* : \nabla \mathbf{v} \, ds - \int_{\Omega} \left( (2\theta^0 - \theta^{-1}) \mathbf{j} \right) \mathbf{v} \, d\Omega = 0
\]  

(4.8)

Here the integral over the domain boundary $\partial \Omega$ is the result of the integration by parts as described in section 3.2, and the fact that we use a slip Dirichlet boundary condition at the top. This results in the test function $\mathbf{v}$ only being defined as zero-valued in the vertical direction at this boundary, so this term does not fully vanish there.

Next we update the pressure using $\mathbf{u}^*$, and further use the solution $p^1$ to solve for the velocity $\mathbf{u}^1$

\[
\int_{\Omega} \nabla p^1 : \nabla q \, d\Omega = -\frac{3}{2\Delta t} \int_{\Omega} (\nabla \cdot \mathbf{u}^*) \mathbf{v} \, d\Omega \\
\int_{\Omega} \mathbf{u}^1 \mathbf{v} \, d\Omega = \int_{\Omega} \mathbf{u}^* \mathbf{v} \, d\Omega - \frac{2\Delta t}{3} \int_{\Omega} \nabla p^1 \mathbf{v} \, d\Omega
\]  

(4.9)

(4.10)

Finally we solve for $\theta^1$

\[
\frac{1}{2\Delta t} \int_{\Omega} \left( 3\theta^1 - 4\theta^0 + \theta^{-1} \right) \mathbf{\tau} \, d\Omega + \int_{\Omega} \mathbf{u}^1 \cdot \nabla \theta^1 \mathbf{\tau} \, d\Omega \\
+ \frac{1}{\sqrt{RaPr}} \int_{\Omega} \nabla \theta^1 : \nabla \mathbf{\tau} \, d\Omega = 0
\]  

(4.11)

4.3.1 FEniCS solver validation

The solver is validated using the same benchmark case described in section 4.2.1, and run for 160 scaled time units. Figures 4.13 and 4.14 show the end time scaled temperature distribution and the horizontal velocity at the probe point, respectively.

Comparing these figures to the ones in section 4.2.1 we see that the results are pretty similar, though with some larger oscillations in the FEniCS case. The computed average horizontal velocity for the last 40 time units is $0.054103$ for the FEniCS solver, which is also fairly close to the results in [1].
Figure 4.13: Distribution of temperature $T$ for FENiCS benchmark at end time. The temperature is set to $\pm 0.00117^\circ C$ at the left and right walls, with isolated top and bottom walls.

Figure 4.14: Scaled horizontal velocity $u/U$ over scaled time $t/\tau$ at probe point(0.181,7.37) for FEniCS benchmark case. $U = \sqrt{g\beta W \Delta T}$ and $\tau = W/U$ where $W$ is the characteristic width of the domain.
4.3. APPROACH USING FENICS

4.3.2 Problem setup for FEniCS

The computational domain for the FEniCS solution was created using gmsh (gmsh.info), as we cannot create graded structured meshes in native FEniCS. This can be done using the Progression syntax in gmsh, which creates points along a line using a geometric procession (each line segment is $x$ times larger than the previous). One must find the right progression scale to get the wanted grading, but this is not too much work. The created mesh has an identical grading as the mesh used in OpenFOAM, although the nature of finite elements gives us two elements per corresponding "volume block" in OpenFOAM, as shown in figure 4.15. The mesh is also scaled to have a height of 1, which is taken as the characteristic length $W$ in the scaled equation set (4.2) - (4.4).

Early on in the process of simulating this problem using FEniCS, I tried using an unstructured mesh with a grading towards the top. These simulations almost always crashed after some time, due to blow-ups around the area where the initial wave motions start (see figure 4.16). This is most likely due to remnants of the velocity that never really die out, and this behavior seems to be easier to subdue.
CHAPTER 4. INTERNAL WAVES

The initial temperature distribution is fairly straightforward to initialize in FEniCS, one only needs to find the linear stratification equation as we did for the OpenFOAM case (and to account for the scaling of the domain). The temperature is then scaled to the variable $\theta$ used in the equations, where $T_h$ and $T_c$ are the upper and lower temperatures, respectively. The reference temperature $T_r$ is taken as the total temperature distribution in the tank divided by volume.

One problem I encountered in the simulations was unboundedness in the solution for temperature. By this I mean that the solution to $\Theta$ tended to attain values outside the maximum and minimum of the initial range of values. This is an unphysical behavior that is clearly a numerical issue, and will result in a blowup of the solution as this error grows. To counter this, all values outside of the initial range was replaced by the max or min value to keep the values inside the proper range.

The velocity probes are generated using the Probes function from fenicstools written by Mikael Mortensen and Miroslav Kuchta (https://github.com/mikaem/fenicstools), probing every 0.1 seconds, and dumped to file at the end of each simulation.

As the results for the OpenFOAM solutions are in good agreement with [17], I have chosen to do the comparison between OpenFOAM and FEniCS using waves run in a shorter tank of length 5 m, with probes at $x = 3$ m. This is done to reduce the computation time for FEniCS. For the comparisons I did extra simulations in OpenFOAM using a mesh of size 100x150 (same resolution as above), as well as one of size 200x300. The FEniCS simulations are done with these mesh resolutions as well, with a time step of $0.01/\tau$ to have the same temporal resolution as in the
4.3. APPROACH USING FENICS

OpenFOAM case.

For all three wave problems, all the simulations were run in serial. For the internal wave case running the program in parallel should not be a big problem in theory. But I encountered a problem on my laptop where the simulation never progressed beyond the call to assign the solution after the first time step to the previous time step when trying to run the case in parallel. This first case was simulated using functions from FEniCS VectorFunctionSpace objects for the velocity, while the two cases below were run using velocity functions created as vectors of two "regular" FunctionSpace functions. The assigning of functions is then done using special functionAssigner objects. I did not want to rewrite my whole program after learning of this way to do things, and there is no guarantee that it would solve the issue of freezing in parallel.

Running the program on larger computers at UiO (for example the Abacus computer) also had it's issues. Version differences in FEniCS created problems with syntax, as well as other crashes that did not justify spending time to attempt to run the program on parallel on these computers. Running the program in serial on these computers did not give a great increase in runtime efficiency over my laptop, as they are used by many people at UiO, often running programs that greatly reduce the capacity of these computers. Having programs that create large amounts of visualization files is also a problem on these computers, as the program will crash if you are out of storage space on your user profile. Therefore I opted in the end to do all simulations on my own laptop, to have full control over storage space and to avoid issues with version incompatibility.

4.3.3 FEniCS results and comparison

A comparison of the velocity profiles for the two mesh sizes for the short tank OpenFOAM solutions and FEniCS are shown in figures 4.17 - 4.20. As one can see, there is good agreement between the solutions in most of the fluid domain, but some variations in the wave itself. The solution for FEniCS using the fine mesh shows a higher velocity in the wave at time -5s and lower velocity at time 3s. This indicates a broadening of the wave that does not occur for the OpenFOAM cases or the coarse FEniCS case. If this is a physical property, due to some false diffusion introduced by the numerics or a mesh property (no mesh convergence) is unclear to me, and the fact that it only occurs for the one case is also a bit of a mystery.

One other strange aspect is the jagged solutions for the coarse OpenFOAM solution, that seems to clear away with higher mesh resolution. Both these aspects where first found with the probes as x = 4 m, and they persisted when moving the probe to x = 3 m. It it therefore reasonable to assume that they are not due to reflections or other wall effects.
Figure 4.17: Comparison of horizontal velocity $u$ scaled with wave speed $c$ over height $y$ scaled by top layer height $h_2$. Taken at position $x = 3$ m at time -5s for both mesh resolutions in OpenFOAM and FEniCS.

Figure 4.18: Comparison of horizontal velocity $u$ scaled with wave speed $c$ over height $y$ scaled by top layer height $h_2$. Taken at position $x = 3$ m at time -1s for both mesh resolutions in OpenFOAM and FEniCS.
4.3. APPROACH USING FENICS

Figure 4.19: Comparison of horizontal velocity $u$ scaled with wave speed $c$ over height $y$ scaled by top layer height $h_2$. Taken at position $x = 3$ m at time $0s$ for both mesh resolutions in OpenFOAM and FEniCS.

Figure 4.20: Comparison of horizontal velocity $u$ scaled with wave speed $c$ over height $y$ scaled by top layer height $h_2$. Taken at position $x = 3$ m at time $3s$ for both mesh resolutions in OpenFOAM and FEniCS.
Another feature that I observed when simulating in FEniCS was a vortex or breaking motion of the wave in the generation phase. This is something that was found to occur in [6, 7, 17], so one would expect to find this in the solutions. They do seem to appear at a lower resolution in FEniCS than in OpenFOAM, which implies that there is heavy smoothing present in the OpenFOAM solution algorithm. One does however start to see the presence of this breaking in the finer of the two OpenFOAM solutions, and one would expect it to be even more prominent with an even higher resolution. For computational time purposes I have not done finer simulations than the ones presented here. Figures 4.21 - 4.24 show this behavior for the four simulations.

![Figure 4.21: Visualization of breaking of wave in the coarse OpenFOAM simulation](image1)

![Figure 4.22: Visualization of breaking of wave in the fine OpenFOAM simulation](image2)

![Figure 4.23: Visualization of breaking of wave in the coarse FEniCS simulation](image3)
Finally, comparing runtimes for the lowest resolution cases, the runtime in OpenFOAM is 252 seconds, while the FEniCS simulation takes 3809 seconds. This is a pretty large difference, even when remembering that the number of degrees of freedom (computational nodes) is larger when using FEM compared to FVM. Taking pressure as an example, one has only one DoF per volume block in OpenFOAM, while for the extreme case of only one volume block as the computational domain, the two triangular elements would have a total of four DoF’s for the pressure when using P1 elements. There are of course other things that can drive up the runtime, and for this case the assembly of the cells and the solution of the equations are the most time consuming processes. These are features that one could most likely optimize without too much work and some deeper understanding of the assembly and solver processes in FEniCS.

4.4 Discussion

We have found that by using a fairly straight forward setup in OpenFOAM (once setFunkyFields was introduced to the setup process), we can generate internal waves that show a similar velocity profile and wave shape as the ones found and discussed in [6, 7, 17]. The FEniCS simulations also give fairly good results, and they also capture more of the chaotic motion around the area of wave generation. The broadening of the wave for the higher resolution is a case of concern though. Looking at higher resolution cases in OpenFOAM would be of interest, but for comparison with FEniCS the computational time of FEniCS running in serial for any finer meshes than the one discussed here is too long. Being able to run in parallel in FEniCS would also enable us to run simulations on the full length of the wave tank, which could give us a comparison with [17] at the actual point of probing that they have plotted for in their paper.
Chapter 5

Solitary surface waves

5.1 Case Description

For the purpose of this text, we define a solitary surface wave as a single wave propagating with constant velocity and shape along the surface of a body of water. As both shape and velocity are (in theory) conserved for all times during the propagation, the conservation of mass and energy when solving with OpenFOAM and FEniCS are of interest.

To reduce the size of the computational domain we implement periodic boundaries, to enable the wave to leave the right hand side of the computational domain and enter the left hand side unchanged. Our domain will be 40m long, to ensure that the velocities in the water far from the wave crest are virtually zero. We will compute 5 cycles of periodicity, e.g. a propagation of 200 m, and see how well energy and mass is conserved.

The data for the solitary wave is taken from files created by Geir Pedersen at UiO. These files have been generated using fully non-linear potential flow theory, to generate velocity field data for the fluid domain below a solitary wave of a given amplitude. The files used for this thesis are the field data for horizontal and vertical velocity $u$ and $v$, as well as a file called $uvsurf$ that contains velocity and coordinate data for the surface of the wave. The data in the files are scaled using a mean water depth $h = 1$ m and a velocity scale $U = \sqrt{gh}$. All data has been reverted to dimensional variables before computations in both OpenFOAM and FEniCS. The case files we have looked at are for a wave of amplitude 0.1 m. Wroniszewski et. al. [18] have implemented a similar case using the same files by Pedersen, also using OpenFOAM as one of the solvers they look at. They do however model without viscosity, so we will not do comparisons with their results here.

5.2 Approach using OpenFOAM

To model the propagation of the solitary wave using OpenFOAM, I have looked at the problem as a multiphase (water/air) problem, using the native interFoam solver.
interFoam uses a phase function called alpha.water that defines the distribution of
air and water in the domain. Water is given the value 1 and air the value 0, and the
interface/surfaces thus takes values between 0 and 1. This phase fraction is solved
for, along with momentum and other fluid properties, to capture the movement of
the two fluids as well as of the interface.

The generation of the initial condition for the phase function alphaWater has been
done using an OpenFOAM application called initSoliton, written by Tormod Landet at UiO. The basic generation of the initial file structures was also taken from
files supplied by Landet, but have been modified to conform to this case (periodic
boundary, domain dimensions, run time etc.)

5.2.1 OpenFOAM solver validation

An easy test case for the OpenFOAM and FEniCS solvers is one where the initial
wave surface has the same shape as the actual case we are looking at below, but with
an initial velocity of zero throughout the domain. This results in the generation of
two waves propagating in separate directions, with the same shape and horizontal
velocity magnitude (with opposite signs). This wave should then meet and reform
at the boundary, and again when returning to the center, without loss of form or
velocity.

We simulate the wave splitting problem in OpenFOAM using resolutions of 200,
400, 800 and 1200 cells in the x direction, with 200, 400 and 600 cells in the y
direction. For the 800 resolution case we omit the finest y resolution, and the 1200
resolution is only done using 200 cells in the y direction. The time stepping is set
to be adjustable, so no temporal refinement is considered, as OpenFOAM finds a
reasonable time step automatically.

In figures 5.1 - 5.2 we plot the kinetic, potential and total energy, as well as the
total mass over one circuit of propagation (40m). These have been computed for
the whole domain, by defining \( E_{kinetic} = 0.5 \int_{\Omega} \rho u^2 \, d\Omega \) and \( E_{potential} = -\int_{\Omega} \rho g y \, d\Omega \).

We look at the deviation from the initial potential energy, and define the total en-
ergy as the sum of kinetic and potential energy.

We see that the total mass is constant for all cases, though with a minutely varying
total mass value between them. The evolution of kinetic energy is as one would
expect, with an increase from the start point, a return to zero at the boundary,
and a similar evolution on the way back to the center. A plausible reason for the
energy not reaching zero at the boundary is that we compute the kinetic energy for
the whole system, so there will be some contribution from the movement of the air
phase. The energy is a function of the density \( \rho \), which has a factor of 1000 to 1 in
favor of water, so the impact of the energy of the water will dominate the general
result.
The potential energy plot shows the change in potential energy from the start, and behaves (as expected) as the inverse of the kinetic energy.

The most interesting plot however is the evolution of the total energy. We see that the resolution of 400x600 gives a virtually unchanged total energy (for the whole system), while the general trend is an increase in the 200 y-resolution cases and a decrease in the 400-resolution cases. The 200x600 case does however follow the decreasing trend. There seems to be no clear system to the increase or decrease, as one can see that the case for 800x200 has a higher total energy than the 1200x200 case. As the kinetic energy ranges from 0 to 120 Joule, a change in total value of 10 to 15 (in magnitude) is not that high of an error all in all. But one can gather that one needs a high y-resolution and a fairly high x-resolution to get a good preservation of energy. For example one would expect a resolution of 800x600 to give even better results than for 400x600. But for computational concerns I have not done finer resolutions than the ones presented above.

Figure 5.1: Evolution of kinetic (left) and potential (right) energy over time for the OpenFOAM validation simulations. Thin solid lines are 200 y-resolution simulations, dash-dotted are 400 y-resolution, and the crossed (seen as thick solid lines) are 600 y-resolution simulations
5.2. APPROACH USING OPENFOAM

Figure 5.2: Evolution of total energy (left) and total mass (right) over time for the OpenFOAM validation simulations. Thin solid lines are 200 y-resolution simulations, dash-dotted are 400 y-resolution, and the crossed (seen as thick solid lines) are 600 y-resolution simulation.

We can also plot the horizontal velocity and alpha.water function over the starting position $x = 0$ m at the beginning and end of the simulations, to see if there are any large changes to these values. Figure 5.3 shows the initial and final horizontal velocities across the line, and figure 5.4 shows the initial and final distribution of the alpha.water function across the line for the 400x600 case.

Figure 5.3: Initial and final horizontal velocity $u$ over depth $y$ for 400x600 mesh validation of OpenFOAM solver

Figure 5.4: Initial and final alpha.water distribution over depth $y$ for 400x600 mesh validation of OpenFOAM solver
CHAPTER 5. SOLITARY SURFACE WAVES

Figure 5.5: Final alpha.water distribution over depth $y$ for 200x200 simulation in OpenFOAM

From the plots we can see that the horizontal velocity below the wave returns to a fairly zero-valued distribution below the wave, but that we have velocities in the air that surpass that in the water in magnitude by a fairly large amount. The alpha.water function seems to be pretty much the same at the start as at the beginning, showing that the interface stays fairly much within the same amount of cells, without dissipating over more cells during the duration of the simulation. A tendency of dissipation of alpha.water is more evident in a plot for the 200x200 actual case simulation as in figure 5.5, showing a fair amount of dissipation of the alpha.water function below the surface.

5.2.2 Problem setup for OpenFOAM

The setup for the actual case simulations are pretty much the same as for the validation, but we also include the velocity in the water by including the velocity field files created by Pedersen.

The velocity files are read and interpolated onto the computational domain, and simulations are run for 5 times the computed wave speed $c$ (also found by Pedersen). The simulations are done using the same resolutions as for the validation.

5.2.3 Results

Figures 5.6 - 5.7 show the kinetic, potential and total energy as well as total mass for the simulated cases. From the plots we see that the lowest (200) y-resolution cases (dash-dotted lines) have a decreasing tendency in both kinetic, potential and total energy from the very beginning of the simulations. The 400 y-resolution cases (dashed lines) have a better evolution for the first half of the simulation period, but the two highest x-resolution cases (400x400 and 800x400) drop off from this point. The 200x400 case however has a somewhat surprisingly steady energy distribution throughout the simulation.

The 600 y-resolution cases (solid lines) both show a steady energy distribution,
with 400x600 being the most steady as in the validation cases. Why the 200x600 case performs better here than for the validation could have to do with the fact that we include the full velocity field, but it is hard to gain any real insight into this from the plots.

Figure 5.6: Evolution of kinetic (left) and potential (right) energy over time for the OpenFOAM simulations for the solitary wave problem. Thin solid lines are 200 y-resolution simulations, dash-dotted are 400 y-resolution, and the crossed (seen as thick solid lines) are 600 y-resolution simulations.

Figure 5.7: Evolution of total energy (left) and total mass (right) over time for the OpenFOAM simulations for the solitary wave problem. Thin solid lines are 200 y-resolution simulations, dash-dotted are 400 y-resolution, and the crossed (seen as thick solid lines) are 600 y-resolution simulations.

We can also here plot the initial and final horizontal velocity and alpha.water distribution across the centerline, to see if there is a pronounced change in these values.
As we can see, there is a slight increase in the velocity directly under the wave surface, and a fairly large negative velocity in the air phase. The alpha.water function remains fairly consistent throughout the simulations.

5.3 Approach using FEniCS

Selected code for the FEniCS implementation of this case can be found in Appendix B.

For the FEniCS approach, we are only looking at the movement of the wave itself, and the wave surface is taken as the top boundary of the computational domain, which is deformed by the changes in the velocity field.

To solve for the motion of the wave, we could have looked at potential flow theory, solving the Laplace equation $\nabla^2 \phi = 0$, and applying appropriate boundary conditions at the bottom and surface. But in keeping with the methods for solving the other cases in this paper, I have chosen to solve the Navier-Stokes equations, using the ALE formulation as in equations (2.8) - (2.9):
\[
\frac{\partial \mathbf{u}}{\partial t} + ((\mathbf{u} - \mathbf{w}) \cdot \nabla) \mathbf{u} = -\frac{\nabla p}{\rho} + \nu \nabla^2 \mathbf{u} + \mathbf{g} \quad \text{in } \Omega \\
\nabla \cdot \mathbf{u} = 0 \quad \text{in } \Omega \\
\mathbf{u} \cdot \mathbf{n} = 0 \quad \text{on } \partial \Omega_B \subset \partial \Omega \\
\sigma \cdot \mathbf{n} = 0 \quad \text{on } \partial \Omega_S \subset \partial \Omega
\]

(5.1)

where

\[
\sigma = -p + \nabla^2 \mathbf{u} \quad \text{(5.2)}
\]

is the stress tensor, \( \mathbf{n} \) is the outward-facing normal vector, and \( \partial \Omega_B \) and \( \partial \Omega_S \) are the bottom and surface boundaries, respectively. We have implemented a slip condition on the bottom, meaning that the fluid can move freely along the bottom wall in the horizontal direction. The condition on the surface implies that the total stress on the surface is zero. We also impose periodic boundary conditions for the side walls.

If we use the finite element method to create a discretization scheme for solving this equation set, we get

\[
\frac{1}{\Delta t} \int_{\Omega} (\mathbf{u} - \mathbf{u}_0)v \, d\Omega + \int_{\Omega} ((\mathbf{u} - \mathbf{w}) \cdot \nabla) \mathbf{u} v \, d\Omega = -\frac{1}{\rho} \int_{\Omega} p(\nabla \cdot v) \, d\Omega + \int_{\partial \Omega_B} p(v \cdot \mathbf{n}) \, ds \\
+ \int_{\Omega} \nabla \mathbf{u} : \nabla v \, d\Omega - \int_{\partial \Omega_B} (\nabla \mathbf{u} \cdot \mathbf{n}) v \, ds \\
+ \int_{\Omega} g v \, d\Omega
\]

(5.3)

\[
\int_{\Omega} (\nabla \cdot \mathbf{u}) v \, d\Omega = 0
\]

(5.4)

Here we have used integration by parts for the pressure and Laplace terms, and the fact that the boundary terms arising from these two operations add up to an equation for the total stress on the surface. As we have set a condition for this stress on the top surface, these two terms vanish at this boundary (as the sum of the two terms equal zero). We still have to implement the boundary integral for the bottom boundary though, as the test function \( v \) is only zero in the vertical direction (due to the unidirectional Dirichlet boundary condition).

### 5.3.1 FEniCS solver validation

To validate the FEniCS solver, we test it on the same zero-velocity case as for the OpenFOAM solver.
To generate the initial wave-shaped mesh, we create a RectangleMesh of a desired size. This mesh has an ordered and easily accessible structure, suited for manipulation and deformation in a consistent way. The surface elevation is read from file (again computed by Geir Pedersen), and is interpolated to the coordinates of the mesh nodes. The mesh is then deformed using these values, moving each node at a given x-coordinate by an appropriate fraction based on y-location (1 at the top, 0 at the bottom). This fraction function ensures that the distribution of nodes at any given x-location is consistent. So whether the height at that location is 1 m (like far away from the crest), or 1.1 m (like at the crest), the fraction of the top node is 1 and the bottom node is 0, with an equal distribution of values between 0 and 1 for all other nodes in that vertical slice. An example of an initial mesh is shown in figure 5.10.

![Initial wave formed mesh after mesh deformation for solitary wave case in FEniCS](image)

Figure 5.10: Initial wave formed mesh after mesh deformation for solitary wave case in FEniCS

As we do not have an interface, the resolution in the y-direction is not of the same importance for the FEniCS solver. What does seem to be an issue though is to not have too "flat" elements, e.g. to stretch the elements too much in the x-direction. This will happen with a too high y-resolution in relation to the x-resolution, so trial and error has been used to find that element blocks that are fairly square (when looking at a pair of triangular elements) give the most stable running of the simulations. These are achieved by using 40-to-1 and 20-to-1 x-y ratios, giving "standing" and "lying" rectangular element blocks as shown in figures 5.11 and 5.12. The resolutions used in the validation and simulations are 200x10, 400x10, 400x20, 800x20 and 800x40.
One problem that arose during the early stages of simulating this case was a numerical instability in the computed velocity at the surface. This instability might also be appearing inside the wave as well, but as we use the surface velocity to move the mesh, it is imperative that the velocity is a smooth solution.

To remedy this sawtooth-looking instability, I implemented a version of a Savitzky Golay Filter [13] I found online at [12]. Newer versions of Scipy have this filter built in, but my version is an older one so I had to implement this scratch-built version. I edited it to account for periodic boundaries using padding from the opposite side of the signal, as the original version just pads the signal at each side using values from that side. This filter is used on the computed u and v velocities at the surface twice at each time step, which is then used to compute the mesh displacement. Figure 5.13 shows an extreme close-up of the vertical velocity far away from the wave for the proper case we discuss in section 5.3.2. This is after only one time step, and we clearly see the sawtooth instability of the solution.

To move the mesh after each time step, we interpolate the velocity solution to a P1
function (after smoothing), and extract the vertical velocity at the top. The mesh velocity is then taken to be the product of this vertical velocity with $\Delta t$ (the time step length) and the fraction function described above. This gives a mesh velocity function equaling the vertical velocity at the top and with 0 velocity at the bottom, so in fact the mesh only moves in a vertical direction, but with a total displacement giving the visual illusion of a wave moving in the horizontal direction.

In regards to the issue of parallelization for this case, the mesh movement above would create a problem when using more processors, as the domain is divided among the processors in a random way. This would imply that nodes in the same x-slice could reside on different processors, and the communication between these to move nodes on one processor based on the surface velocity of a surface node on another processor could be a problem. I have therefore decided to once again run the simulations in serial.

One other issue that arises from moving the mesh is that the only temporal solution that resides on the current mesh is $u^0$. By this I mean that if one wanted to use a higher order temporal discretization (as we have done in the internal wave case), we get the issue of solution $u^{-1}$ having been computed on a different mesh than $u^0$, a mesh that no longer exists after moving the mesh before solving for $u^0$. This could probably be fixed by mapping functionality and other advanced methods, but has not been pursued in this text. I am not sure whether or not functions on different meshes can be used in the same form, so creating a temporary mesh at each iteration to hold the solution $u^{-1}$ could be one way to implement higher order
5.3. APPROACH USING FENICS

temporal discretization.

As a result of only being able to use the previous solution in the discretization scheme, I have implemented the case using mixed function spaces and a coupled solver. This solves the whole equation set in one step, with a solution containing both velocity and pressure. This is more robust when it comes to finding the correct pressure, but at the cost of computational time.

The evolution of the kinetic, potential and total energy as well as total mass for the FEniCS validation is shown in figures 5.14 - 5.15. We see a fairly similar evolution of the kinetic and potential energy as for the OpenFOAM validation. The total energy and mass is fairly well preserved for the higher resolution cases, though the coarse 200x10 case seems to be too coarse to retain energy and mass at a satisfactory level. As the potential energy calculated is only a function of the computational domain (and constant density), the discrepancy between this and the computed kinetic energy (the change in total energy) is due to the reduction of total mass.

Interestingly, though the variation in mass is discernible from the plot (as opposed to the exactly retained mass in OpenFOAM), the value only fluctuates between 40768 and 40755 kg for the higher resolution cases. This is in fact only a change in mass of 0.032%, so FEniCS retains the total mass well for the validation case, though not perfectly as is the case with OpenFOAM. The retention of mass actually seems better for the lowest y-resolution in each pair of x-resolution simulations.

Figure 5.14: Evolution of kinetic (left) and potential (right) energy over time for the FEniCS validation simulations
CHAPTER 5. SOLITARY SURFACE WAVES

Figure 5.15: Evolution of total energy (left) and total mass (right) over time for the FEniCS validation simulations

One obvious question regarding refinement is whether to use temporal or spatial refinement when trying to get better or converging results. The ratio between time step and spatial discretization size is important for stability in discretization schemes, but choosing to refine the one over the other is not always obvious. I have therefore run simulations using two temporal refinements on the 400x10 case in FEniCS, ending up with a time step one fourth of the original 0.01 s. One can roughly expect this to run at the same speed as the 800x20 case, where we increase the number of degrees of freedom for the mesh by four. Figure 5.16 shows the original time step, two temporal refinements and the spatial refinement, and it seems that temporal refinement actually worsens the rate of mass loss for this case. I have therefore only looked at spatial refinement for the cases run in this thesis, as this always improves the results for the mesh resolutions I have implemented. Temporal refinement for finer meshes than the ones used in this thesis would probably be needed at some point, but the implemented choice of time step seems to be sufficient for the cases I have run.

5.3.2 Problem setup in FEniCS

The files used to initialize the velocity field are separated into horizontal, vertical and surface velocity files. The computational coordinates of internal field files do not coincide with the coordinates for the surface, so we need to combine the data into a complete velocity field by some function manipulation in FEniCS.

Firstly we create matrices for the u and v velocity fields, as well as coordinate arrays for x and y. As the data given only includes values where the computations made by Pedersen were reliable, we need to extend the domain to ensure that the velocity is zero far away from the crest. This is done by adding columns of zero values to the ends of the velocity matrices, as well as corresponding x-coordinates to the x-array. We can then get an arbitrary, interpolated velocity field by interpolating onto a domain of our own choice (the mesh created above).
5.3. APPROACH USING FENICS

Figure 5.16: Comparison of temporal and spatial refinement of validation simulations for FEniCS solver

As mentioned above, the surface velocity is not represented in the field velocity files, so the interpolation of the fields create strange effects on or near the surface, as the values above the surface are all zero. One way to correct this could be to extrapolate the velocity below the surface for each x-coordinate and fit the zero-valued data to a curve. A more robust (and correct) method is to interpolate the surface velocity to the coordinates of the top dofs of the deformed mesh, and then manually replace all the top values with the interpolated values.

Figure 5.17 shows the initial horizontal velocity profile before and after inserting for the values for the surface velocities.

The simulations are done on the same mesh resolutions as for the validation case, with a time step of 0.01 seconds.

5.3.3 Results

Figures 5.18 - 5.19 show the evolution of the energy and mass for the FEniCS simulations. The general trend from the validation case can be seen here as well, with the coarse resolution case giving bad results, but with the other resolutions giving fairly good results (better with higher x-resolution). The total energy decreases steadily throughout the simulation at what looks to be a linear rate, but the rate of loss seems to get better with mesh refinement as well. The kinetic and potential energy seem to vary with a periodic fluctuation. Even higher resolutions than the ones used here might give even better results, but have not been implemented
Figure 5.17: Comparison of velocity field before and after insertion of interpolated surface velocity for solitary wave problem in FEnICS
5.3. APPROACH USING FENICS

for computational purposes. One interesting thing to note is that the energy is virtually equal for the 20-1 and 40-1 cases for the higher resolution cases. This implies that the x-resolution is of much higher importance for the accuracy than the y-resolution, though similar ratios as described above should still be preserved. We also see that the mass actually decreases less for the 40-1 ratio than the 20-1 ratio, but that this also seems to converge for a higher x-resolution.

![Figure 5.18: Evolution of kinetic (left) and potential (right) energy over time for the solitary wave simulations in FEniCS](image)

![Figure 5.19: Evolution of total energy (left) and total mass (right) over time for the solitary wave simulations in FEniCS](image)

Another thing that I discovered for the FEniCS simulations is a sort of "shedding" of velocity at the beginning of the simulations. EXPAND Figures 5.20, 5.21 and 5.22 show the surface elevation and (smoothed) horizontal and vertical velocities at the surface at the beginning and end of the simulation of the 400x10 case.

Comparing the initial and final total mass in the tank for the highest resolution simulation, we have a change from 40768 to 40758 kg, a loss of 0.025%, which is even better than for the shorter validation case. This has probably something to do
with the fact that the actual case wave is already in motion with a set shape that does not vary as much as the validation case.

Figure 5.20: Initial (left) and final (right) surface elevation $y$ over position $x$ for 400x10 mesh for solitary wave problem in FEniCS

Figure 5.21: Initial (left) and final (right) horizontal surface velocity $u$ over position $x$ for 400x10 mesh for solitary wave problem in FEniCS
5.4 Comparison and discussion

Looking first at the difference in runtime for the validation cases, the OpenFOAM simulation for the 200x200 resolution takes 97 seconds, while the 200x10 FEniCS simulation uses 604 seconds. This is again a huge difference, that will only get larger with finer mesh resolutions.

The energy can be retained fairly well in OpenFOAM by choosing the right mesh resolution, but this seems to be sort or arbitrary when it comes to the ratio. A high y-resolution does seem to be the way to go however. The energy in FEniCS has a more resolution-consistent increase in quality, with what seems to be a convergence towards a linear decline with higher mesh resolution. Better temporal discretization schemes might be one solution, as well as implementing a splitting scheme. However, the problem of solutions residing on different meshes is the main problem faced in regards to higher order schemes, and the reason for using a coupled solver and first order time discretization.

Mass is retained perfectly in OpenFOAM, and the rate of loss is fairly low in FEniCS. This could again be improved with better discretization schemes.
Chapter 6

Wave/body interaction

6.1 Case Description

We will be looking at a body of "infinite" height, oscillating in heave with an amplitude of 0.05m (initial motion downwards), and with angular velocities $\omega$ ranging from $0.25\pi$ to $2\pi$. The computational domain will be a tank with water depth of 4m, with a layer of 1m of air above the water surface for the OpenFOAM simulations. The body has a width of 2m, and an initial submersion of 0.5m. The computational domain is 40m wide, with the oscillating body centered at $x = 0$m.

We will compare various mesh refinements, and compute the added mass and damping for the different values of $\omega$. This will then be compared with known values for added mass and damping for a body in heave.

6.2 Approach using OpenFOAM

The OpenFOAM framework for this case was created using the interDyMFoam, which is an interFoam solver adapted for mesh dynamics. No further information will be given beyond what is discussed in section 5.2, and no validation will be done on the solver.

6.2.1 Problem Setup in OpenFOAM

For the OpenFOAM simulations, we run a series of commands in the terminal window to generate the computational domain, and to define the geometry of the setup. These parameters are defined in specific files in the folder hierarchy of each OpenFOAM case. For example, the oscillating object is created by making the call `topoSet`, which removes a defined area of the original rectangular domain (defined in the blockMeshDict file). The bottom of the object is defined by another `topoSet` call to the file `topoSetDict-selectBottom`, as the patches on the bottom of the object will need specific boundary conditions for the displacement of the object. Finally the phase function `alpha.water` is set using a call `setFields` to the file `setFieldsDict`. Figure 6.1 shows the initial `alpha.water` distribution in the full computational domain.
6.2. APPROACH USING OPENFOAM

Figure 6.1: Initial field distribution of alpha.water for oscillating object problem in OpenFOAM

The movement of the object is initialized by defining a set point displacement in the \textit{pointDisplacement} file. The values we can give in the file are $\omega$ and the amplitude $a$, but the formula for the displacement ($asin(\omega t)$) is hard-coded into the deeper file structure of OpenFOAM. I found no way to change this, as I would have liked to use a displacement based on the cosine as this will give a smoother start to the simulations. Since velocity is the derivative of the displacement, a velocity of $a\omega cos(\omega t)$ means that the velocity is highest at time 0, when the object starts from rest. This gives a virtual infinite acceleration, which might create problems. OpenFOAM seems to handle this anyway, but for the FEniCS case I have chosen to use a displacement of $acos(\omega t)$ to ensure a smooth starting motion.

For the OpenFOAM simulations we have used a mesh resolution of 400 blocks in the x direction, and y resolutions of 10, 20, 40 and 80 blocks. Simulations have also been done on a 800x40 mesh to see if increased x-resolution gives any improvement in the results. The logging of the forces is done using the "forces" function object which is native in OpenFOAM. This is applied to the patches forming the bottom of the oscillating object. This function enables us to write the total forces on the specified patches to file, which can be used for further processing.

When processing, the computed pressure forces are read from the outfiles, and the added mass and damping coefficients are computed using a Python program. This is done by using the least squares function in Numpy on equation 6.1

$$F = -(A\ddot{x} + B\dot{x} + Cx) + K$$

(6.1)

Here the dots denotes temporal derivation of the displacement x, A is added mass, B is damping, C is the stiffness matrix and K is some constant. By using the displacement $x = asin(\omega t)$ we get

$$F = A\omega^2 asin(\omega t) - B\omega cos(\omega t) - Casin(\omega t) + K$$

(6.2)
For heave, the value of \( C \) is given as 
\[
C = A_{\text{bottom}} \times \rho \times g,
\]
where \( A_{\text{bottom}} \) is the area of the bottom of the object, \( \rho \) is the density of the water, and \( g \) is the constant of gravity. Defining the variables \( A^* = a(A\omega^2 - C) \) and \( B^* = -B\omega \), we can solve for these variables by solving the equation

\[
M \cdot y = T
\]  
(6.3)

Where \( M \) is a matrix consisting of three columns of \( \sin(\omega t_i) \), \( \cos(\omega t_i) \) and constant "ones", \( y \) is an array of the coefficients \( A^*, B^* \) and the constant coefficient \( K \) (linked to the "ones" column in \( M \)), and \( T \) is an array of the computed forces at each time \( t_i \). This can be solved with the least squares method using the function \text{linalg.lstsq} found in Numpy for Python.

The simulations are run for 10 seconds with a time step of 0.01 seconds. The reason for the smaller \( y \)-resolution in this case compared to the solitary wave case is the fact that the results we are looking for are based on forces on the bottom of the object, instead of the actual wave motion away from the object. A finer \( y \)-resolution like in the solitary wave case would of course give us a sharper interface and a nicer view of the wave motion if we wanted to visualize or do analysis of the waves, but for the purpose of computational speed we have used fewer blocks in the \( y \)-direction for this case.

6.2.2 Results

A good way to validate that the computed values for the coefficients are correct is to plot the computed force against the function \( F = A^*\sin(\omega t) + B^*\cos(\omega t) + K \), this being the force using the computed values from the least squares method in Python. An example of this is shown in figure 6.2. We can see that the general trend is found correctly, but there are some spikes in the computed forces, as well as the effect of having a sudden increase in the velocity. In fact, I have removed the first computed value from the force data, as this is a very high value compared to the "correct values" shown in the plot.

Figures 6.3 and 6.4 show the scaled computed values (coefficients) for the added mass and damping as a function of scaled angular velocity. This is compared to the analytical solution given in [11] for a body of width to depth ratio of 4 as in our case. As we can see, the computed values for the added mass overshoot the analytical solution slightly, but is within a reasonable range of the correct values. The values for \( \omega = 0.25\pi \) are off in general, and are in fact not visible at all in the plot for damping when using the axis values found in [11]. An interesting observation is that the result seems fairly independent of \( y \)-resolution. Refining in the \( x \)-direction also gives fairly similar results, so the computed values seem to be converged at the resolutions used in the simulations.
6.2. APPROACH USING OPENFOAM

Figure 6.2: Computed pressure force vs analytical force using least squares result coefficients. Plotted over time for 400x40 simulation of oscillating object problem in OpenFOAM

Figure 6.3: Added mass coefficients (added mass in heave $a_{22}$ over density $\rho$ times square of object width $W$) over scaled angular velocity ($\omega$ times square root of initial object depth $D$ divided by gravity $g$). Plotted over time for oscillating object problem for OpenFOAM
Figure 6.4: Damping coefficients coefficients (damping in heave $b_{22}$ over density $\rho$ times square of object width $W$ times angular velocity $\omega$) over scaled angular frequency ($\omega$ times square root of initial object depth $D$ divided by gravity $g$). Plotted over time for oscillating object problem for OpenFOAM

6.3 Approach using FEniCS

Selected code for the FEniCS implementation of this case can be found in Appendix C.

When meshing the domain for the FEniCS case, I first attempted to make a structured mesh in gmsh, to be able to employ similar techniques as in section 5.3.2. This was attempted by trying to combine three rectangles into the complete domain, one for the lower tank part, and two for each side of the oscillating object. Unfortunately, this introduced the problem of the whole top of the lower rectangle being treated as a boundary in FEniCS, so when trying to move the bottom of the object, this whole y-coordinate line also wanted to move as well.

The mesh was instead created as an unstructured mesh in gmsh, which is pretty straightforward. One only needs to define the corner points of the mesh, and the relative point density at each node. Figure 6.5 shows the mesh close to the object for the FEniCS case.

I wanted to have the mesh behave as in the solitary wave case, where the top of the computational domain moves with the wave motion created by the oscillation of the object. One major challenge faced when implementing this is the fact that the top node on each side of the object has to move as part of the surface, while all other nodes on the side move as part of the object. This means that the top and...
bottom nodes have to be able to move in opposite directions, as for example the initial downward movement of the object will push the water at each side upwards.

This was solved by creating a special boundary condition for the mesh displacement of each side of the object (and also the sides of the domain). I created an interpolation function using the velocity and position of the top and bottom nodes. The value of the mesh displacement of all nodes between the top and bottom nodes were then given a value by interpolating the y-coordinate of each node using this function.

The boundary condition for the mesh displacement on the surface was taken as the vertical velocity at the surface, and zero at the bottom. The horizontal mesh velocity was set to zero on the whole boundary. At the bottom of the oscillating object, we set the mesh and fluid velocity to be the temporal derivative of the desired point displacement ($-a\omega\sin(\omega t)$ in our case).

The solution algorithm is similar to the one used in the solitary wave case, and we solve equations similar to 5.1-5.4. The only difference is that I have chosen to apply a no-slip condition to the bottom, so the bottom boundary terms in equation 5.3 vanish. We first solve for the velocity $u$ using a coupled solver and a mixed function space as in section 5.3.2. The solution is then interpolated to a P1 space and applied to the boundary conditions for the mesh velocity. We then solve the Laplace equation $\nabla^2 w = 0$ to find the mesh velocity for the whole domain. Logging of the forces on the bottom of the object is easily done by using the assemble function in FEniCS, and applying this to the pressure solution in the y direction at the bottom of the object. This is written to a file called forces, which we use for further processing.

### 6.3.1 Results

Unfortunately the simulations in FEniCS crash after a certain point for all the values of $\omega$. The velocity solutions show similar numerical instabilities as for the solitary wave case, so I had to write a filter for the velocity at the surface that is applied
to the solution before solving for \( w \). This filter is a five-point average filter, using extrapolated values at the boundaries, and demanding that the horizontal velocity at the boundaries was zero (the nodes at the walls cannot move horizontally). Figure 6.6 shows the horizontal velocity at the surface on the left hand side of the object, as well as the smoothed signal after running it ten times through the filter. This is after one timestep, and occurs even with temporal refinement (the velocity is just scaled with the reduction in the size of the timestep). The crash of the simulation occurs after some time due to the mesh velocity at the top node of the wall growing to a point where it either squishes the wall in on itself, or explodes upwards.

Plotting the velocity at a time step shortly before the crash as in 6.7 we see the general behavior that I wanted to get out of this model. There is a clear wave motion propagating away from the oscillating object, and the velocity is mainly centered in two vortex motions at the lower corners of the object.

Although the program crashes, I was able to use the logged force calculations up until the time of crashing to compare with the analytical values for added mass and damping. Figure 6.8 shows a similar plotting as we did for OpenFOAM of the simulations and the solution to the least squares problem, and we can see that there is a beginning instability at the end of the time series leading up to the crash. Keeping in mind that the point displacement was changed to \( \cos(\omega t) \), one needs to change the order of the sine and cosine in the matrix \( M \) described in section 6.2.1, as well as changing the definition of \( B^* \) to \( B^* = B\omega \).
6.3. APPROACH USING FENICS

Figure 6.7: Visualization of velocity field and wave motion before crash for oscillating object problem in FEniCS

Figure 6.8: Computed pressure force vs analytical force using least squares result coefficients. Plotted over time for simulation of oscillating object problem in FEniCS

Figures 6.9 and 6.10 show the computed added mass and damping for the FEniCS solutions. We can see that although the solver crashes, the logged values give us fairly good results up until the time of crashing. The values for the lowest case $\omega = 0.25\pi$ do show strange results in the FEniCS solutions however, similar to what we saw in OpenFOAM.
**CHAPTER 6. WAVE/BODY INTERACTION**

Figure 6.9: Added mass coefficients (added mass in heave $a_{22}$ over density $\rho$ times square of object width $W$) over scaled angular velocity ($\omega$ times square root of initial object depth $D$ divided by gravity $g$). Plotted over time for oscillating object problem for FEniCS

Figure 6.10: Damping coefficients (damping in heave $b_{22}$ over density $\rho$ times square of object width $W$ times angular velocity $\omega$) over scaled angular frequency ($\omega$ times square root of initial object depth $D$ divided by gravity $g$). Plotted over time for oscillating object problem for FEniCS
6.4 Comparison and discussion

As the FEniCS solver crashes, I have not looked at a comparison of run times for this case. We would presumably see similar differences in run time however, as the solvers are pretty much identical to the ones used in the solitary wave case.

Looking at the plots for added mass and damping, we see similar results between the OpenFOAM and FEniCS solutions. Both solvers overshoot both parameters slightly compared to the analytical solution from [11], but some margin of error must be considered for the analytical solution, as this has been imported from the book using imaging software, and the plot data exported and plotted against my solutions.
Chapter 7

Conclusion and future work

Summarizing the findings of chapters 4, 5 and 6, we can give the following general conclusions and remarks:

- OpenFOAM is a faster and generally more stable framework, as is expected when using optimized solvers compared to the self-implemented models we employ in FEniCS.

- Fairly similar results are found for the internal wave case, with the FEniCS solver capturing more of the chaotic motion at the gate location for the lowest mesh resolution.

- OpenFOAM preserves mass perfectly, whereas there is a small loss of mass in the FEniCS model for the solitary wave propagation case.

- Energy is preserved well for the higher y-resolution cases in OpenFOAM. There is loss in all the FEniCS cases, but this seems to have a trend of being reduced with higher x-resolution.

- The added mass and damping coefficients land within the same range for both the OpenFOAM and FEniCS solutions.

In general, we see that the solutions for the solitary wave case should benefit from further refinements, and a conversion to higher order discretization might also improve the results. But as noted above, the challenge of solutions residing on different meshes must be addressed. Further refinements of the internal wave case would probably emphasize the chaotic initial motions better in OpenFOAM, but as the results are as good as they are compared to [17], this is not something I would spend too much time on compared to the two other cases. In the oscillating object case we seem to have fairly converged results both in the OpenFOAM and FEniCS cases, but getting the FEniCS case to run without crashing would of course be the first goal when it comes to further work on this case.

Further refinement should be an easy fix for all cases in OpenFOAM by implementing parallelization, as this should work as long as the simulations are able to...
run in serial. In FEniCS I have as described above not been able to run the programs in parallel, and further refinements will have a very long runtime without further optimization. As we can see from the plots above the solutions seem to converge with higher x-resolution for the solitary wave, so being able to run 1600, 3200 etc x-resolution cases should give better results, but with an expected slowing rate of improvement with each refinement.
Appendix A

Selected FEniCS code for internal wave case

Listing A.1: Initial field creation for temperature field

```python
# Define initial temperature field
class InitialT(Expression):
def eval(self, value, x):
    if x[0] < (0.5/0.77) and x[1] <= -(0.31/0.77):
        value[0] = 5.0
    if x[0] <= (0.5/0.77) and x[1] > -(0.31/0.77) and x[1] <= -(0.16/0.77):
        value[0] = 145.43 + 348.81 * x[1]
    if x[0] <= (0.5/0.77) and x[1] > -(0.16/0.77):
        value[0] = 72.95
    if x[0] > (0.5/0.77) and x[1] <= 0.0:
        value[0] = 5.0
    if x[0] > (0.5/0.77) and x[1] > 0.0:
        value[0] = 5.0 + 348.81 * x[1]
def value_shape(self):
    return ()
```

Listing A.2: Forms for the equations

```python
# Tentative velocity step
F1 = (1./(2*dt))*inner((3*u-4*u0+u_1), v)*dx + inner(grad(u)*(2*u0 - u_1), v)*dx - \
    inner(j*inner(2*Theta0 - Theta_1), v)*dx + C1*inner(grad(u),grad(v))*dx - \
    C1*inner(dot(grad(u),n),v)*ds(2)
a1 = lhs(F1) L1 = rhs(F1)
# Pressure update
a2 = inner(grad(p), grad(q))*dx L2 = -(3./(2*dt))*div(u1)*q*dx
# Velocity update
a3 = inner(u, v)*dx L3 = inner(u1, v)*dx - ((2.*dt)/3.)*inner(grad(p1), v)*dx
# Temperature
F4 = (1./(2*dt))*inner(3*Theta - 4*Theta0 + Theta_1, Theta_t)*dx + \
    inner(dot(grad(Theta),ul), Theta_t)*dx + (1./C2)*inner(inner(grad(Theta),grad(Theta_t)),Theta_t)*dx
```

Listing A.3: Replacement of values outside of the initial range at each time step

```python
above = np.array(np.where(Theta1.vector() > Theta_u)) length = len(above[0,:]) for i in range(length):
    Theta1.vector()[above[0,i]] = Theta_u
below = np.array(np.where(Theta1.vector() < Theta_l)) length = len(below[0,:]) for i in range(length):
    Theta1.vector()[below[0,i]] = Theta_l
```

68
Appendix B

Selected FEniCS code for solitary wave case

Listing B.1: Finding the DoF’s at the top of the computational domain

```python
# dof_coordinates = S.dofmap().tabulate_all_coordinates(mesh)
nn = S.dim() dn = mesh.geometry().dim()
dof_coordinates.resize((nn, dn))
# Indices of the top vertices
top_dofs = np.where(dof_coordinates[:, 1] == 0)[0]
# Sort indices by x coordinate of the corresponding vertex
list(top_dofs)
top_dofs = list(top_dofs)
top_dofs.sort(key=lambda i: dof_coordinates[i, 0])
top_dofs = np.array(top_dofs, int)
xcoords = dof_coordinates[top_dofs, 0]
```

Listing B.2: Displacement of the mesh to create initial wave shaped mesh

```python
boundary = BoundaryMesh(mesh, "exterior")
for x in boundary.coordinates():
    ind = 0
    for i in range(len(eta)):
        if np.isclose(x[1], 0, rtol=1e-05, atol=1e-08) and np.isclose(x[0], xcoords[i],
            rtol=1e-05, atol=1e-08):
            ind = i
ALE.move(mesh, boundary)
```

Listing B.3: ISmoothing of surface velocities and moving of mesh

```python
origU = u_.vector()[top_dofs]
for i in range(2):
    u_.vector()[top_dofs] = savitzky_golay(u_.vector()[top_dofs], 31, 3)
origV = v_.vector()[top_dofs]
for i in range(2):
    v_.vector()[top_dofs] = savitzky_golay(v_.vector()[top_dofs], 31, 3)
top_vel = v_.vector()[top_dofs[1::2]]
for i in range(len(top_vel)):
    valDict[D_xcoords[i]] = top_vel[i]
for i in range(len(D_xcoords)):
    disp.vector()[dofDict[D_xcoords[i]]] = valDict[D_xcoords[i]] * timestep *
    Frac_v[dofDict[D_xcoords[i]]]
ALE.move(mesh, disp)
```
Appendix C

Selected FEniCS code for oscillating object case

Listing C.1: Filter for the vertical surface velocity

```python
def v_filter(x, y):
    step = x[1] - x[0]
    left_x = x[:10]
    left_pad_x = np.linspace(x[0] - 10*step, x[0] - step, 10)
    left_vals = y[:10]
    right_x = x[-10:]
    right_pad_x = np.linspace(x[-1] + step, x[-1] + 10*step, 10)
    right_vals = y[-10:]
    f_left = InterpolatedUnivariateSpline(left_x, left_vals, k=1)
    f_right = InterpolatedUnivariateSpline(right_x, right_vals, k=1)
    pad_left = f_left(left_pad_x)
    pad_right = f_right(right_pad_x)
    padded_signal = np.concatenate((pad_left, y, pad_right))
    for i in range(len(y)):
        y[i] = (padded_signal[10+i-2] + padded_signal[10+i-1] +
                padded_signal[10+i] + padded_signal[10+i+1] +
                padded_signal[10+i+2])/5.0
    return y
```

Listing C.2: Boundary condition for mesh velocity at the left hand side of the object

```python
class leftObjectMeshVel(Expression):
    def __init__(self, U, yvals, dofs):
        print("init left")
        self.U = U
        self.yvals = yvals
        self.dofs = dofs
        print(self.yvals)
    def eval(self, values, x):
        #print("Evaluating x= %f, y= %f (x[0], x[1])")
        f_left = sc.interp1d( [self.yvals[0], self.yvals[-1]],
                              [self.U.vector()[self.dofs[0]], self.U.vector()[self.dofs[-1]]])
        values[0] = 0
        values[1] = f_left(x[1])
```

Listing C.3: Assembly of forces on the bottom of the object

```python
force = assemble(up_.sub(2)*n[1]*ds(7))
```
Bibliography


[12] Scipy Cookbook: Savitzky Golay Filtering


