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

Implicit Methods for Real-Time Simulation of Interactive Waves

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

for the degree

MASTER OF SCIENCE
in
APPLIED MATHEMATICS

by

Akash Mittal

Delft, The Netherlands
July 23, 2014

Delft University of Technology

MSc THESIS APPLIED MATHEMATICS

Implicit Methods for Real-Time Simulation of Interactive Waves

Akash Mittal

Delft University of Technology

July 23, 2014 Delft, The Netherlands

Preface

This Master thesis has been written within the Erasmus Mundus Master's program *Computer Simulations for Science and Engineering (COSSE)*[1]. This included a six months internship carried out at Maritime Research Institute Netherlands (MARIN) in collaboration with Technical University Delft.

MARIN provides a unique combination of simulation, model testing, full-scale measurements and training programmes to the shipbuilding and offshore industry and governments. One of such service includes the supply of full-scale bridge simulators to the shipping companies which can not only be used to train captains, steersmen and other ship workers, but also for research and consultancy purposes.

MARIN continuously strive towards bringing innovation in their products. Currently they are working on a new wave model called the 'Interactive Waves'. The project focuses on developing a simulator in which ships and waves interact. The new wave model is the Variational Boussinesq model (VBM) as suggested by Gert Klopman. However, this new realistic model brings much more computation effort with it. The VBM mainly requires an unsteady state solver, that solves a coupled system of equations at each frame (20 fps) . The model is currently limited by the time-step constraints, which makes it difficult to simulate large and non-uniform domains in real-time. The focus of the current Master's thesis is to develop a fast and robust solver with good stability properties.

Acknowledgements

During the past two years of my studies, I had the opportunity to study for one year at the University of Erlangen-Nuremberg, Germany, as well at the Technical University Delft, The Netherlands under the COSSE program. COSSE also provided me the possibility to interact with some of the great minds in the field of Computer Science and Applied Mathematics and, develop a friendship network which will last beyond the boundaries of the program.

Foremost, I would like to thank my supervisor Professor Kees Vuik from TU Delft for providing regular feedback and support during the thesis.

I would like to thank MARIN and in particular Auke Ditzel for giving me the opportunity to do research at MARIN on this challenging project, Auke Van Der Ploeg on his expertise on the linear solvers and Anneke Sicherer-Roetman for the support on the programming aspects. Also, I would like to thank Martijn de Jong, his work helped me alot.

During the last two years of studying abroad, the support of my parents never ended and I am very thankful for many of their advices.

Akash Mittal, Delft, July 2014

[1]For more information, please visit: http://www.kth.se/en/studies/programmes/em/cosse

Contents

Chapter 1

Introduction

1.1 Background

Simulation of ocean waves can be categorized into two major groups. First one is based on the physical models whereas the other generates the ocean waves based on either geometrical shapes or oceanography spectrum. The later method group requires less computational effort, but results in the the waves which are less realistic in nature.

Currently MARIN (Maritime Research Institute Netherlands) provides ship maneuvering simulators to the shipping companies which can be used for training and research programs. Computation of the wave field in these simulators is based on the second method and utilizes wave spectra (Fourier theory). Being deterministic in time and space; they are easy to implement on distributed simulation systems. However, this model is not interactive, that is, the impact of the ship movement on the wave field is not taken into account. In order to capture the impact of the ship movement and perform realistic simulations, the waves need to be modeled physically; but in a real time, which is a requirement imposed by the ship simulator.

The physical modeling for the ocean wave simulation requires solving the non-linear Navier Stokes equations. With the current simulation techniques available, it is not possible to solve these equations in three dimensions in real time. Although it is possible to simplify the physical model to a certain extent by making some suitable approximations. This is where the so-called Boussinesq's approximation comes into picture, which can be applied to weakly non-linear and fairly long waves. The idea is to employ these approximations and develop a numerical solver which is fast enough to carry out the simulations in real time.

The essential idea in the Boussinesq approximation is the elimination of the vertical coordinate from the flow structure, while still retaining some of the influence of the vertical structure of the flow. This is possible because wavs propagate in the horizontal direction, whereas it has a different non-wave like behavior in the vertical direction. In Boussinesq-like models, an approximate vertical structure of the flow is used to eliminate the vertical space. Though, the classical Boussinesq models are limited to long waves - having wavelengths much longer than the water depth. Also, they cannot deal with varying depths.

The variational Bousinnesq model (developed by Klopman [2]) can deal with deep waters with varying depth, as the vertical structure is treated as a function of depth, surface elevation and other parameters. The big advantage over the deterministic model is that it can incorporate the influence of the ship movement, thus making it an interactive wave model. Owing to the variation in the vertical structure and its interactive nature, it is much more computational intensive, and much work has been done in order to improve the running time of the solver. Two previous master thesis by Van't Wout [3] and De Jong [1] have focused on creating a fast linear solver and increasing the efficiency by GPU programming.

MARIN intends to use this solver to simulate the cases including multiple ships. In these cases smaller grid sizes are required and this puts severe restrictions on the current numerical techniques.

1.2 Earlier Work

1.2.1 Elwin van't Wout: Fast solver and model explanation

Elwin's work comprised of improving the linear solver that is used in the real-time ship simulator. The underlying wave model is the variational Boussinesq model as suggested by Klopman [2]. Elwin starts with derivation of the Variational Boussinesq Model (VBM). Starting point of the derivation of the model equations are the Euler equations which are valid for motions of ideal fluids and are a special case of the Navier-Stokes equations (inviscid incompressible flow). In order to reduce the problem from 3 dimensions to 2 dimensions, vertical shape functions are employed. The resulting model is linearized to reduce complexity. The linearized model is described in detail in Chapter 2.

Equations are then discretized (spatially) using a finite volume method with structured Cartesian grids. Time integration is performed using an explicit Leap-Frog scheme which is described later. One of the equations results in a system of linear equations. Elwin provides a detailed description of various numerical methods available for solving linear system of equations. Based on the properties of the matrix involved, he selects the Conjugate Gradient method as the solver. This iterative method is then combined with various preconditioners, namely Diagonal scaling, modified incomplete Cholesky and repeated red black ordering. Elwin modified the original implementation of the repeated red black-preconditioner to a repeated red black preconditioner with a predefined number of levels, combined with a complete Cholesky decomposition on the maximum level. Elwin concludes that the repeated red black preconditioner requires the lowest amount of computation time, in most of the cases.

1.2.2 Martijn De Jong: Developing a CUDA solver for large sparse matrices for MARIN

The C++ RRB solver developed by Elwin was able to solve the system of linear equations within 50 ms for domains no bigger than 100,000 to 200,000 nodes. The focus of Martijn's thesis was to increase the performance of above solver by utilizing GPU architecture. By carefully selecting the storage data structures and optimal CUDA programming parameters, Martijn was able to achieve a speedup of 30x as compared to the serial code for the linear solver. Martijn reports that the new solver can solve systems that have more than 1.5 million nodes within 50ms with ease.

1.3 Research Direction

Our main target is to explore different approaches to increase the performance of the current time dependent solver, and allow the solution of larger problems in a similar runtime. Our approach will be to first analyze the proposed algorithms on smaller test problems in MATLAB, and then implement the suitable algorithms and solvers in C++ and CUDA. The following research questions and goals are addressed in the thesis to achieve the desired target:

- Analyze the stability and accuracy of various implicit time integration schemes.
- Implement generalized Krylov subspace method.
  - Analyze the impact on the performance (speed-up) and storage requirements for the generalized Krylov Subspace methods for CUDA and C++ code.
  - Study the performance of different Preconditioner combinations to solve the coupled set of equations.
- Provide a framework to study non-uniform meshes.

1.4 Organization of the Thesis

The organization of the thesis is given below:

1. Chapter 2 describes the governing equations and discretization.
2. Chapter 3 provides a brief overview of the test cases and the systems used for testing.
3. Chapter 4 and 5 discusses various time integration techniques.
4. Chapter 6 discusses the iterative linear solvers and preconditioners.
5. Chapter 7 discusses the implementation of the linear solvers for the selected time integration.
6. Chapter 8 and 9 provides the results for the various test cases in C++ and CUDA.
7. Chapter 10 discusses the analytical results for the numerical dispersion and provides a framework for the non-uniform grid two-dimensional test case.

Chapter 2

Model Equations and Discretization

2.1 The Variational Boussinesq Model (VBM)

The basic idea of the model is to minimize the pressure in the whole fluid. Starting point of the derivation of the model equation comes from the physical model of the Navier-Stokes Equations. The Euler equations are a special case of the Navier Stokes equations where viscous forces are neglected. For ocean waves, this is considered to be a valid assumption as the Reynolds number is typically large, and thus surface waves in water are a superb example of a stationary and ergodic random process. Also, rotational effects are negligible in large water waves. The Euler equation is given by:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}.\nabla)\mathbf{u} = \frac{1}{\rho}\nabla p - \mathbf{g}, \tag{2.1}$$

The fluid velocity is given by $\mathbf{u}$, ρ denotes the mass density, p the pressure and $\mathbf{g}$ the gravitational constant. Understanding physics behind the problem is important as it provides information on the numerical techniques required for the solution of the problem. Note that the inviscid and irrotational assumption is not necessarily valid near solid boundaries, where very small flow structures associated with turbulence result from the no-slip boundary condition. In the current model, this has been ignored and the assumptions are considered to be valid throughout the domain.

Equation 2.1 is converted to the instationary Bernoulli equation (fluid being assumed irrotational) , and then the pressure is integrated over the whole domain. The basic idea of the variational Boussinesq model is to minimize the total pressure p. For irrotational flows, velocity can be represented by a velcoity potential (ϕ). Another parameter ζ (water level) is also introduced while performing integration. The problem thus becomes finding these functions (ϕ and ζ). The vertical structure of the flow is often known. The velocity potential can be written as a series expansion in predefined vertical shape functions, thus reducing the 3D model to a 2D model.

2.1.1 Hamiltonian Description

Being an irrotational and incompressible flow, the system of equations has a Hamiltionian structure, where the force acting on the system is the gravitational and the pressure force. The

Hamiltionian $H(\zeta, \varphi)$ is given by the sum of kinetic and potential Energy. In order to obtain the minimal value of total pressure, the Hamiltionian function is minimized with respect to variables ζ and φ [3]. This results in :

$$\frac{\partial \zeta}{\partial t} = \nabla H_{\varphi}(\varphi, \zeta) \tag{2.2a}$$

$$\frac{\partial \varphi}{\partial t} = \nabla H_{\zeta}(\varphi, \zeta) \tag{2.2b}$$

where ∇H_ζ refers to the partial derivative of H with respect to ζ. The structure of the above equations will play an important role in the discussion of temporal integration which is discussed in Chapter 4.

2.1.2 Linearized Model Description

The variational Boussinesq model constructed based on the above simplifications is non-linear in nature. In order to simplify these equations and thus reducing the computational effort, the Hamiltonian equations given by Equation (2.2) is linearized. Detailed derivation from the Hamiltonian to linear set of equations is described in [3].

The final set of equations after performing the linearization are given by:

$$\frac{\partial \zeta}{\partial t} + \nabla.(\zeta \mathbf{U} + h\nabla\varphi - h\mathcal{D}_0\nabla\psi) = 0, \tag{2.3a}$$

$$\frac{\partial \varphi}{\partial t} + \mathbf{U}.\nabla\varphi + g\zeta = P_s, \tag{2.3b}$$

$$\mathcal{M}_0\psi + \nabla.(h\mathcal{D}_0\nabla\varphi - \mathcal{N}_0\nabla\psi) = 0, \tag{2.3c}$$

Notes:

1. Equations 2.3 are solved for three basic variables: water level ζ, surface velocity potential (2-D) φ and vertical structure ψ. Splitting of the velocity potential into the surface potential and the vertical shape function is given by:

$$\phi(x, y, z, t) = \varphi(x, y, t) + f(z)\psi(x, y, t) \tag{2.4}$$

2. Shape function f is chosen among either a parabolic or a cosine-hyperbolic shape. The model parameters (functionals $\mathcal{D}, \mathcal{M}$ and $\mathcal{N}$) are computed using the shape function f.

3. The water depth $h = h(x, y, t)$ is relative to the reference level $z = 0$. The bottom of a basin, river or sea is thus at level $z = -h$.

4. The total velocity can be split into the average current velocity and the velocity due to the wave front. $\mathbf{U} = \mathbf{U}(x, y, t)$ is the time average horizontal velocity of the current and is used as an external input in the model.

5. Equations 2.3 represent the motion of waves. The impact of a moving ship is not seen directly. In order to model a ship, a pressure pulse on the water surface is defined. In the pressure functional given by Equation (2.3b), P_s represented the source term with $P_s := -\frac{p_s}{\rho}$. The draft of a ship's hull is the vertical distance between the waterline and

the bottom of the hull. Given the draft of a ship, p_s is computed as the hydrostatic pressure at the given depth. Let the draft be given as d_s, then $p_s = g d_s \alpha(x, y)$ with $\alpha(x, y)$ a shape function with one in the middle of the ship and zero on the horizontal boundary of the ship. Alternatively, the shape of the ship's hull can be imported from an external surface description file.

2.2 Spatial Discretization

Both Elwin and Martijn have assumed the domain to be rectangular, and divided the domain in a rectilinear Cartesian grid. The dimensions of the domain are $L_x \times L_y$. It is divided into $N_x \times N_y$ grid points. The outermost nodes represent the physical boundary. The mesh spacing in each direction is given as $\Delta x = \frac{L_x}{N_x - 1}$ and $\Delta y = \frac{L_y}{N_y - 1}$. An example is given in ??

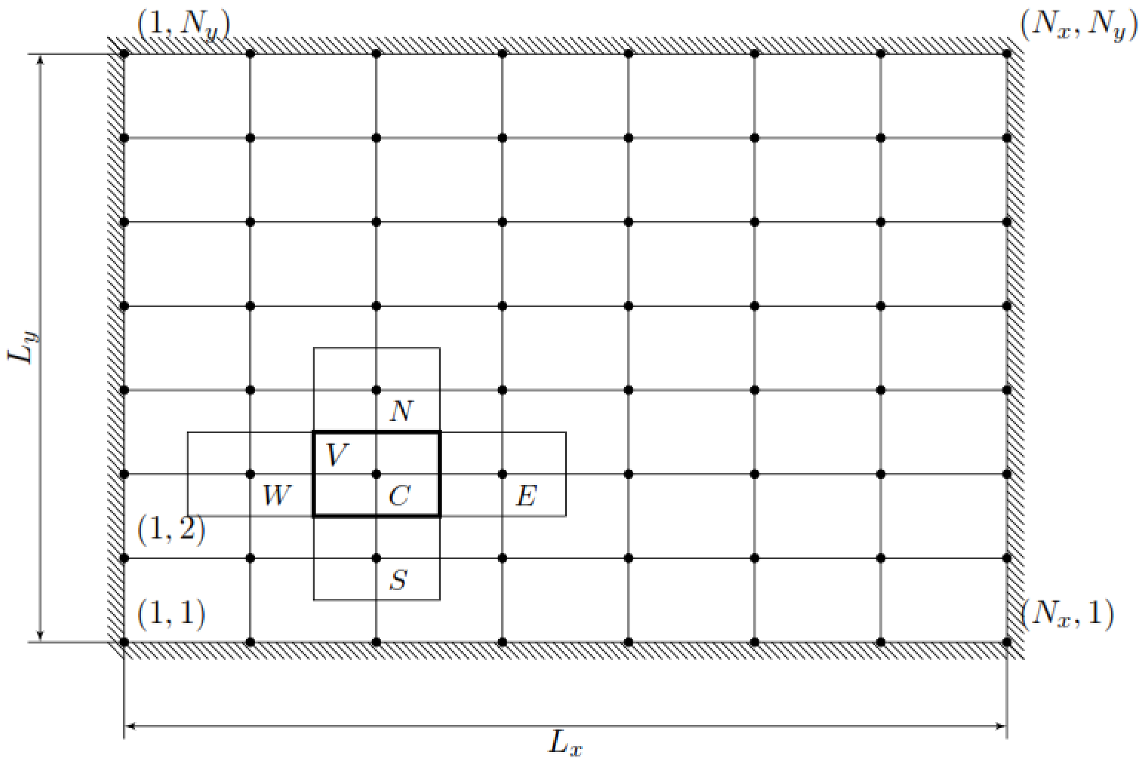


Figure 2.1: Physical domain and corresponding Cartesian grid

The model Equations 2.3 are discretized using a finite volume method. The variables are evaluated at the grid points and the finite volumes are rectangles of size $\Delta x \times \Delta y$ centered around the grid point. The derivatives are approximated with centered differences yielding a five-point stencil. For the grid point located at C (= center) the surrounding control volume V and its four nearest neighbors (N = north,E = east, S = south, W = west) are indicated in Section 2.2. On node (i, j), the discretized versions of the variables ζ, φ and ψ are given by ζ_{ij}, φ_{ij} and ψ_{ij}. In order to put the variables in matrix format, one dimensional ordering of the variables is defined which gives the vector $\vec{\zeta}, \vec{\varphi}$ and $\vec{\psi}$. The spatial discretization can be written as :

$$\frac{\mathrm{d}}{\mathrm{dt}}\begin{bmatrix}\vec{\zeta}\\ \vec{\varphi}\\ 0\end{bmatrix}+\begin{bmatrix}S_{\zeta\zeta} & S_{\zeta\varphi} & S_{\zeta\psi}\\ S_{\varphi\zeta} & S_{\varphi\varphi} & S_{\varphi\psi}\\ S_{\psi\zeta} & S_{\psi\varphi} & S_{\psi\psi}\end{bmatrix}\begin{bmatrix}\vec{\zeta}\\ \vec{\varphi}\\ \vec{\psi}\end{bmatrix}=\begin{bmatrix}0\\ P_s\\ 0\end{bmatrix} \tag{2.5}$$

The matrix S's are given by five point stencils as follows:

$$S_{\zeta\zeta}:\begin{bmatrix}0 & \frac{1}{2\Delta y}\overline{V_N} & 0\\ -\frac{1}{2\Delta x}\overline{U_W} & \frac{1}{2\Delta y}\overline{V_N}-\frac{1}{2\Delta y}\overline{V_S}-\frac{1}{2\Delta x}\overline{U_W}+\frac{1}{2\Delta x}\overline{U_E} & \frac{1}{2\Delta x}\overline{U_E}\\ 0 & -\frac{1}{2\Delta y}\overline{V_S} & 0\end{bmatrix} \tag{2.6}$$

where U and V denotes the current velocity in x and y direction respectively.

$$S_{\zeta\varphi}:\begin{bmatrix}0 & -\frac{1}{\Delta y^2}\overline{h_N} & 0\\ -\frac{1}{\Delta x^2}\overline{h_W} & \frac{1}{\Delta y^2}\overline{h_N}+\frac{1}{\Delta x^2}\overline{h_W}+\frac{1}{\Delta x^2}\overline{h_E}+\frac{1}{\Delta y^2}\overline{h_S} & -\frac{1}{\Delta x^2}\overline{h_E}\\ 0 & -\frac{1}{\Delta y^2}\overline{h_S} & 0\end{bmatrix} \tag{2.7}$$

$$S_{\zeta\psi}:\begin{bmatrix}0 & -\frac{1}{\Delta y^2}\overline{h_N D_N} & 0\\ -\frac{1}{\Delta x^2}\overline{h_W D_W} & \frac{1}{\Delta y^2}(\overline{h_N D_N}+\overline{h_S D_S})+\frac{1}{\Delta x^2}(\overline{h_W D_W}+\frac{1}{\Delta x^2}\overline{h_E D_E}) & -\frac{1}{\Delta x^2}\overline{h_E D_E}\\ 0 & -\frac{1}{\Delta y^2}\overline{h_S D_S} & 0\end{bmatrix} \tag{2.8}$$

$$S_{\varphi\zeta}=g,\quad S_{\varphi\psi}=0,\quad S_{\psi\zeta}=0 \tag{2.9}$$

$$S_{\varphi\varphi}:\begin{bmatrix}0 & \frac{1}{2\Delta y}\overline{V_N} & 0\\ -\frac{1}{2\Delta x}\overline{U_W} & -(\frac{1}{2\Delta y}\overline{V_N}-\frac{1}{2\Delta y}\overline{V_S}-\frac{1}{2\Delta x}\overline{U_W}+\frac{1}{2\Delta x}\overline{U_E}) & \frac{1}{2\Delta x}\overline{U_E}\\ 0 & -\frac{1}{2\Delta y}\overline{V_S} & 0\end{bmatrix} \tag{2.10}$$

$$S_{\psi\varphi}:\Delta x\Delta y\begin{bmatrix}0 & \frac{1}{\Delta y^2}\overline{h_N D_N} & 0\\ \frac{1}{\Delta x^2}\overline{h_W D_W} & -(\frac{1}{\Delta y^2}\overline{h_N D_N}+\frac{1}{\Delta x^2}\overline{h_W D_W}+\frac{1}{\Delta x^2}\overline{h_E D_E}+\frac{1}{\Delta y^2}\overline{h_S D_S}) & \frac{1}{\Delta x^2}\overline{h_E D_E}\\ 0 & \frac{1}{\Delta y^2}\overline{h_S D_S} & 0\end{bmatrix} \tag{2.11}$$

$$S_{\psi\psi}:\Delta x\Delta y\begin{bmatrix}0 & -\frac{1}{\Delta y^2}\overline{\mathcal{N}_N} & 0\\ -\frac{1}{\Delta x^2}\overline{\mathcal{N}_W} & \frac{1}{\Delta y^2}\overline{\mathcal{N}_N}+\frac{1}{\Delta x^2}\overline{\mathcal{N}_W}+\frac{1}{\Delta x^2}\overline{\mathcal{N}_E}+\frac{1}{\Delta y^2}\overline{\mathcal{N}_S}+\mathcal{M} & -\frac{1}{\Delta x^2}\overline{\mathcal{N}_E}\\ 0 & -\frac{1}{\Delta y^2}\overline{\mathcal{N}_S} & 0\end{bmatrix} \tag{2.12}$$

The system can be written as :

$$\dot{\mathbf{q}}=L\mathbf{q}+f, \tag{2.13a}$$

$$S\vec{\psi}=\mathbf{b} \tag{2.13b}$$

with $\mathbf{q} = \begin{bmatrix} \vec{\zeta} \\ \vec{\varphi} \end{bmatrix}$ and $\dot{\mathbf{q}}$ its time derivative. The matrix $L = -\begin{bmatrix} S_{\zeta\zeta} & S_{\zeta\varphi} \\ S_{\varphi\zeta} & S_{\varphi\varphi} \end{bmatrix}$ is the spatial discretization matrix and $\mathbf{f} = -\begin{bmatrix} S_{\zeta\psi}\vec{\psi} \\ S_{\varphi\psi}\vec{\psi} \end{bmatrix}$.

Elwin and Martijn focused on solving the system of equations represented by Equation (2.13b), whereas the Equation (2.13a) was solved using explicit time integration schemes described in the next Chapter.

Chapter 3

Test Problems

3.1 One Dimensional Test Problem

To assess the stability and accuracy of various implicit time integration procedures, we shall consider a simplified problem in one dimension given below as the test problem.

$$\frac{\partial \zeta}{\partial t} + \nabla.(\zeta \mathbf{U} + h \nabla \varphi) = 0, \tag{3.1a}$$

$$\frac{\partial \varphi}{\partial t} + \mathbf{U}.\nabla \varphi + g\zeta = 0, \tag{3.1b}$$

Here, we have neglected the impact of the vertical structure ψ and pressure pulse P_s. This is now a coupled initial boundary value problem, where both equations are hyperbolic in nature and also represents an Hamiltonian Set. The boundary conditions for both ζ and φ are taken as periodic.

Assuming no spatial variation of mean current velocity U and depth h, the above equations can be written as:

$$\frac{\partial \zeta}{\partial t} + \mathbf{U}\frac{\partial \zeta}{\partial x} + h\frac{\partial^2 \varphi}{\partial x^2} = 0, \tag{3.2a}$$

$$\frac{\partial \varphi}{\partial t} + \mathbf{U}\frac{\partial \varphi}{\partial x} + g\zeta = 0, \tag{3.2b}$$

Implicit time integration procedures will require solving a system of linear equations. We will use MATLAB to solve the system after performing spatial discretization and applying the boundary conditions. The idea here is to assess the stability and accuracy of various methods and not the performance. THe solution from various time integration procedures will be benchmarked with MATLAB's inbuilt ODE integrators.

3.2 Two Dimensional Test problem- Assessment with MATLAB and C++

In order to analyze the stability and accuracy of the complete system and implement the required solver in C++ and CUDA, it is required to study the two dimensional case which results in a Pentadiagonal matrix instead of a Tridiagonal matrix.

In this test problem, a simplified problem which addresses issues related to the coupling of vertical structure along with wave height and surface potential in two dimensions will be considered. First the system will be tested with MATLAB to understand the stability of the required scheme and then will be implemented in C++ and CUDA. The test problem is used to build the coupled RRB solver for the implicit set of equations first in C++ and then in CUDA. The solution obtained from C++ and CUDA can then be benchmarked against the results from MATLAB.

The equations considered are:

$$\frac{\partial \zeta}{\partial t} + \mathbf{U}_x \frac{\partial \zeta}{\partial x} + \mathbf{U}_y \frac{\partial \zeta}{\partial y} + h(\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2}) - h\mathcal{D}(\frac{\partial^2 \psi}{\partial x^2} - \frac{\partial^2 \psi}{\partial y^2}) = 0, \tag{3.3a}$$

$$\frac{\partial \varphi}{\partial t} + \mathbf{U}_x \frac{\partial \varphi}{\partial x} + \mathbf{U}_y \frac{\partial \varphi}{\partial y} + g\zeta = 0, \tag{3.3b}$$

$$\mathcal{M}\psi + hD(\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2}) - \mathcal{N}(\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2}) = 0, \tag{3.3c}$$

The algorithm required to solve the above set of equations is described in detail in Section 5.4.2.

3.3 Realistic problems IJssel, Plymouth

Two realistic problems are obtained from MARIN's database and previous simulations by Martijn in [1]. Testing will be carried out with respect to the validated models in [1]. A very small time step (much lower than the allowable limit) will be used to first carry out simulations with the current model which use Explicit Leap Frog time integration. Results from implicit time integration will then be compared with the results from explicit time integration.

3.3.1 The Gelderse IJssel

The Gelderse IJssel, a small river, is a branch from the Rhine in the Dutch provinces Gelderland and Overijssel. The river flows from Westervoort and discharges in the IJsselmeer. In Figure 6.1 a (small) part of the river is shown. From this part several test problems are extracted. This is done by extracting small regions out of the the displayed region . For the discretization an equidistant 2 m by 2 m grid is used.

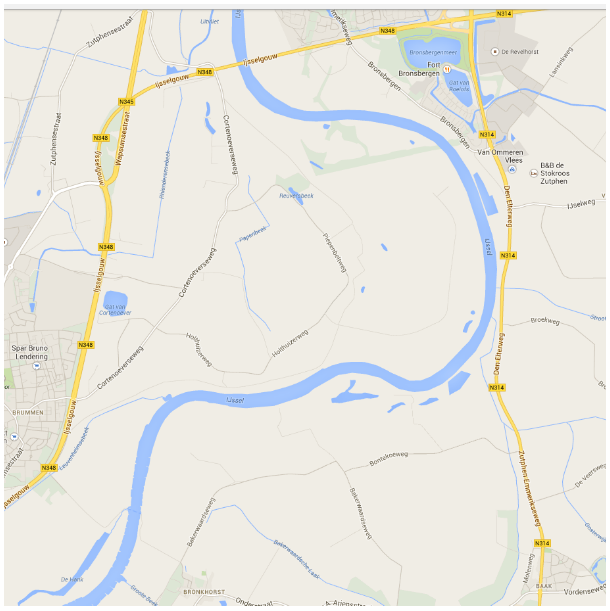


Figure 3.1: The Gelderse IJssel (Google Maps)

3.3.2 The Plymouth Sound

Plymouth Sound is a bay located at Plymouth, a town in the South shore region of England, United Kingdom. The Plymouth Breakwater is a dam in the centre of the Plymouth Sound which protects anchored ships in the Northern part of the Plymouth Sound against south-western storms. From this region also test problems are extracted, see Figure 6.2. For the discretization an equidistant 5 m by 5 m grid is used.

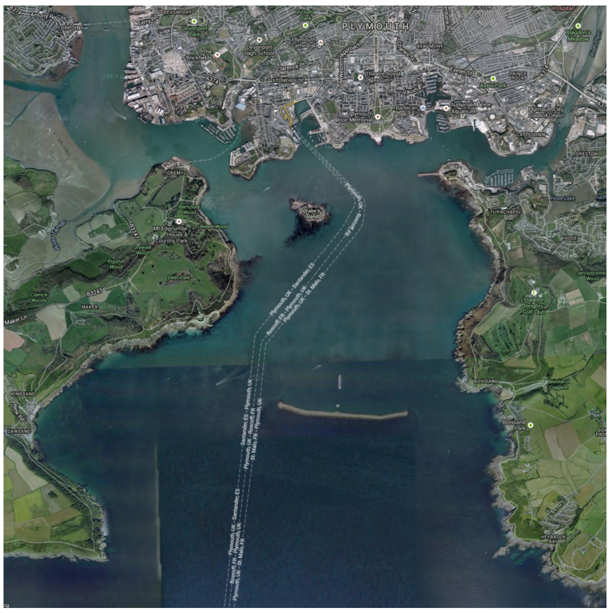


Figure 3.2: The Plymouth Sound (Google Maps)

3.4 Testing System

Below is an overview of the machine that has been used in our simulations. This system can be considered having one of the best available GPU cards along with CPU configuration. Though it is possible to obtain GPU cards with higher specifications, their cost is much higher.

Brand / Type	Dell Precision Workstation T3600
Owner / System no.	MARIN LIN0245
CPU	Intel Xeon E5-1620 @ 3.6 GHz
No. of cores	4 (8 Threads)
Cache	256 kB L1 / 1 MB L2 / 10 MB L3
Memory	15.5 GB RAM DDR2 @ 1066 MHz
Motherboard	Dell Custom
Operating System	Scientific Linux Release 6.5
System kernel	2.6.32 - 431 .17.1.ei1.x86-64
CUDA release	6.0
Nvidia Driver version	331.62
GCC version	4.4.7
GPU	(CUDA) GeForce GTX 780
Memory	3072 MB
No. of cores	12 SM 192 (cores/SM) = 2304 cores
GPU Clock Rate	941 MHz
Compute capability	3.5

Chapter 4

Temporal Discretization

The system of equations to be advanced in time are given in Equation (2.13a). Expanding the equation we get:

$$\dot{\vec{\zeta}} = -S_{\zeta\zeta}\vec{\zeta} - S_{\zeta\varphi}\vec{\varphi} - S_{\zeta\psi}\vec{\psi} \tag{4.1a}$$

$$\dot{\vec{\varphi}} = -S_{\varphi\zeta}\vec{\zeta} - S_{\varphi\varphi}\vec{\varphi} - S_{\varphi\psi}\vec{\psi} + P_s \tag{4.1b}$$

The equation for φ can be further simplified taking into account the structure of the stencils.

$$\dot{\vec{\varphi}} = -S_{\varphi\varphi}\vec{\varphi} - g\vec{\zeta} + P_s \tag{4.1c}$$

where g is the gravitational constant. Above equations in the form of Equation (2.13a) are solved using the explicit Leap-Frog scheme in [1] and [3]. The method is described in 4.1.

The equation for variable ψ, which does not contain the time derivative, but still depends on the other variables is given by

$$S_{\psi\psi}\vec{\psi} = -S_{\psi\varphi}\vec{\varphi} \tag{4.1d}$$

4.1 Explicit Time Stepping :Leap-Frog Scheme

In explicit schemes, the fluxes and the sources are computed at the n^{th} (previous) time level and their contribution is added to the current value of the variable. In the work by Elwin and Martijn, the Leap-Frog method has been used to integrate equation 2.13a. The Leapfrog method is one of the Symplectic Integration techniques designed for the numerical solution of Hamilton's equations given by Equation (2.2).

The method described in [3] depends on two previous time steps (a so-called multistep) method. The exact solution to Equation (2.13a) is approximated at the time intervals $t_n = n\Delta t$, $n = 0, 1, 2, \ldots$ with $\Delta t > 0$ being the time step size. The numerical approximations are denoted by $\mathbf{q}^n \approx \mathbf{q}(t_n)$.

To keep the derivation short, we will first focus on the fixed constant step size $\Delta t := t_{n+1} - t_n$. A Taylor series expansion gives:

$$\mathbf{q}^{n+1} = \mathbf{q}^n + \Delta t\dot{\mathbf{q}}^n + \frac{1}{2}\Delta t^2\ddot{\mathbf{q}}^n + \frac{1}{6}\Delta t^3\dddot{\mathbf{q}}^n + \mathcal{O}(\Delta t^4), \tag{4.2a}$$

$$\mathbf{q}^{n-1} = \mathbf{q}^n - \Delta t\dot{\mathbf{q}}^n + \frac{1}{2}\Delta t^2\ddot{\mathbf{q}}^n - \frac{1}{6}\Delta t^3\dddot{\mathbf{q}}^n + \mathcal{O}(\Delta t^4) \tag{4.2b}$$

Subtracting the second equation from first, we obtain:

$$\mathbf{q}^{n+1} - \mathbf{q}^{n-1} = 2\Delta t\dot{\mathbf{q}}^n + \mathcal{O}(\Delta t^3), \tag{4.3}$$

which reduces to

$$\dot{\mathbf{q}}^n = \frac{\mathbf{q}^{n+1} - \mathbf{q}^{n-1}}{2\Delta t} + \mathcal{O}(\Delta t^2), \tag{4.4}$$

The Explicit Leap-Frog method is second order accurate in time, and is only conditionally stable. The main advantage of this method is that, it is simple and fast, and is easily parallelizable.

The disadvantage is that the time step is limited by the numerical stability requirements. For the equations represented by Equation (2.13a), the stability condition requires that the time step is shorter than the crossing time of the grid cells by the faster wave:

$$\Delta t < \frac{\Delta x_i}{c_i^{max}} \tag{4.5}$$

for all grid cells and all directions $i = x, y$ and c represents the velocity of the wave. This is the famous Courant-Friedrich-Levy (CFL) condition valid for explicit time integration of arbitrary hyperbolic PDE's. In the current implementation, with the uniform-grid, Δx is constant for each cell, and the velocity of the surface gravity wave is used to determine the time step such that it satisfies the CFL condition. A safety margin is added to ensure stability without needing to check the stability criteria at each time step.

For example, given the equation $\dot{y} = \lambda y$ (λ arises as an eigenvalue of a local Jacobian and could be complex), the Leap-Frog method is stable only for $|\lambda\Delta t \leq 1|$. In order to approximate the interaction between the ship and the waves, a grid size of the order of half a meter or smaller is required. This limits the time step to 0.01 seconds maximum, while the maneuvering simulator at MARIN often can use time steps as large as 0.1 seconds.

The intention to use lower mesh sizes to capture details of wave interaction with ships at finer levels will require a further reduction in the time step, hence increasing the computation time. This provides the motivation to explore time integration methods which are more stable, hence allowing us to use larger time steps, and still giving good accuracy.

4.2 Implicit Time Stepping

4.2.1 Fully Implicit Scheme

Stability of the time integration can be significantly improved by employing fully implicit time integration techniques. The most common and simplest of the fully implicit scheme is the Backward Euler scheme. For the set of equations given by Equation (2.13a), we get:

$$\frac{q_{n+1} - q_n}{dt} = (Lq_{n+1} - S_{\zeta\psi}\vec{\psi}_{n+1}) \tag{4.6a}$$

where q $= \begin{bmatrix} \vec{\zeta} \\ \vec{\varphi} \end{bmatrix}$. The method is first order accurate in time [1] and is unconditionally stable.

While the stability of the implicit time discretization is a great improvement over the explicit schemes, one has to solve the implicit equations for the unknowns q_{n+1} and $\vec{\psi}_{n+1}$, which requires solving a system of equations (Differential Algebraic Equations) represented by :

$$(I - L)q_{n+1} = q_n - S_{\zeta\psi}\vec{\psi}_{n+1} \tag{4.7a}$$

$$S_{\psi\psi}\vec{\psi}_{n+1} = -S_{\psi\varphi}\vec{\varphi}_{n+1} \tag{4.7b}$$

As the method is first order accurate in time, and the spatial discretization was second order accurate, overall accuracy of method is only first order. Such methods are usually good to achieve steady-state results, but not very accurate. The disadvantages of using the fully implicit scheme includes:

- Moving from explicit to implicit schemes incurs extra computational efforts as it requires solving the linear system of equations given by Equation (4.7a). For the system represented above, the iterative linear solvers are used, which are described in more detail in Chapter 5.
- First order implicit methods simply under-relax more to maintain the stability of the iterative solution. It is this increased damping, with the increase in time-step size, which induces more inaccuracies in transient behavior.
- Spatial discretization plays an important role in the stability of the numerical solutions of unsteady state hyperbolic equations. Generally for the Euler equations, the central difference scheme is more accurate than the first order upwind schemes. Also stability in the case of the central differencing scheme is not an issue as the diffusive forces are not active. In the case flux limiters are used for spatial discretization which have the capability of capturing shocks, discontinuities or sharp changes in the solution domain, the implicit scheme results in the formation of non-linear system of equations, which then requires more computational effort (computation of Jacobi).

4.2.2 β - Implicit Scheme

Semi-implicit methods try to combine the stability of the implicit methods with the efficiency of the explicit methods. The system of equations is discretized using a one parameter (β) implicit scheme to advance in time:

$$q_{n+1} = q_n + \Delta t(\beta(Lq_{n+1} - S_{\zeta\psi}\vec{\psi}_{n+1}) + (1-\beta)(Lq_n - S_{\zeta\psi}\vec{\psi}_n)) \tag{4.8a}$$

The parameter β can vary between 0 and 1. For $\beta = 1$, we get the Backward-Euler method (fully implicit) and the so-called Trapezoidal scheme for $\beta = 0.5$

For $\beta = 0.5$, we can rewrite the equations as:

$$q_{n+1} = q_n + 0.5\Delta t(\dot{q_{n+1}} + \dot{q_n}) \tag{4.9a}$$

$$= q_n + \Delta t\dot{q_n} + 0.5\Delta t(\dot{q_{n+1}} - \dot{q_n}) \tag{4.9b}$$

$$= q_n + \Delta t\dot{q_n} + \frac{1}{2}\Delta t^2\frac{d\dot{q}}{dt} + O(\Delta t^3) \tag{4.9c}$$

which gives the trapezoidal method of second order temporal accuracy.

[1] The can be derived from the Taylor series expansion

4.2.3 Stability

The Scalar Test Equation

To understand the stability of time integration methods, we consider the scalar, complex test equation :

$$w^{'}(t) = \lambda w(t) \tag{4.10}$$

where $\lambda \in C$.

Application of the time integration scheme (Explicit or Implicit) gives :

$$w_{n+1} = R(\Delta t\lambda)w_n, \tag{4.11}$$

$R(\Delta t\lambda)$ is called the Stability Function. Let $z = \Delta t\lambda$. For the explicit schemes of order s, $R(z)$ is a polynomial of degree $\leq s$. For implicit methods it is a rational function with degree of both numerator and denominator $\leq s$. The stability region is defined in terms of $R(z)$ as:

$$S = z \in C : |R(z)| \leq 1 \tag{4.12}$$

The scheme that has the property that S contains entire left half plane $C^{-} = x \in C : Re(z) \leq 0$ is called A-Stable. A scheme is said to be Strongly A-Stable if it is A-stable with $|R(\infty)| < 1$, and it is said to be L-stable if in addition $|R(\infty)| = 0$.

For the semi-implicit scheme with parameter β, the stability function is $R(z) = \dfrac{1 + (1-\beta)z}{1 - \beta z}$. The implicit trapezoidal rule given by $\beta = 0.5$ is A-stable, whereas the fully implicit Backward Euler method is L-stable with $\beta = 1$.

Stability for Linear Systems

Let the linear system of equations (m equations, m unknowns) be given as:

$$w^{'}(t) = Aw(t) + g(t) \tag{4.13}$$

with $A \in \mathcal{R}^{mxm}$. Application of the semi-implicit scheme with parameter β gives:

$$w_{n+1} = R(\Delta tA)w_n + (I - \beta\Delta tA)^{-1}\Delta t g_{n+\beta} \tag{4.14}$$

where

$$R(\Delta tA) = (I - \beta\Delta tA)^{-1})(I + (1-\beta)\Delta tA) \tag{4.15}$$

and $g_{n+\beta} = (1-\beta)g(t_n) + \beta g(t_{n+1})$. Starting from initial solution of w_0, we obtain:

$$w_n = R(\Delta tA)^n w_0 + \Delta t \sum_{i=0}^{n-1} R(\Delta tA)^{n-i-1}(I - \beta\Delta tA)^{-1} g_{i+\beta} \tag{4.16}$$

If we perturb the initial solution $\hat{w}_0$, we get the formula for the perturbed solution at n^{th} time step as:

$$\hat{w}_n - w_n = R(\Delta tA)^n(\hat{w}_0 - w_0) \tag{4.17}$$

Hence, the powers $R(\Delta tA)^n$ determine the growth of the initial errors.

Let λ_j with $1 \leq j \leq m$ denote the eigenvalues of the matrix A, and let A be diagonizable such that $A = U\Lambda U^{-1}$ where $\Lambda = diag(\lambda_j)$ [7]. Let K be the condition number of U. Then for $\Delta t\lambda_j \in S, 1 \leq j \leq m \implies ||R(\Delta tA)^n|| \leq K \; \forall n \geq 1$ where S represents the stability region described above in the scalar test equation case.

Stability of the semi-implicit methods requires a moderate bound for these powers. In case the powers are not bounded, $\Delta t\lambda_j$ does not belong to the Stability Region S, resulting in an unstable method.

4.2.4 Backward Differentiation Formula

Another approach to achieve second order temporal accuracy is by using information from multiple previous time steps. This gives rise to a two-parameter three-level time integration scheme:

$$q_{n+1} = q_n + \Delta t \left[\alpha \frac{q_n - q_{n-1}}{\Delta t} - \alpha \dot{q_n} + \beta \dot{q_{n+1}} + (1-\beta)\dot{q_n} \right] \tag{4.18}$$

where $\dot{(}q_n$ represents the derivative of q at the n^{th} time interval.

This scheme is three-level whenever the parameter $\alpha \neq 0$. When $\alpha = 1/3$ and $\beta = 2/3$, we obtain the second order Backward Differentiation Formula (BDF2) for constant time step Δt.

The analysis of the accuracy for a multi-step method is presented below. Let us assume that q_n was computed from q_{n-1} with a second order temporal accuracy. This implies

$$\frac{q_n - q_{n-1}}{\Delta t} = \dot{q_n} - (\Delta t/2)\frac{d\dot{q}}{dt} + O(\Delta t^2) \tag{4.19}$$

Substituting this into Equation (4.18)

$$q_{n+1} = q_n + \Delta t \left[\dot{q_n} - \alpha \frac{\Delta t}{2}\frac{d\dot{q}}{dt} + \beta \Delta t \frac{d\dot{q}}{dt} + O(\Delta t^2) \right] \tag{4.20a}$$

$$= q_n + \Delta t \dot{q_n} + \Delta t^2 (\beta - \frac{\alpha}{2})\frac{d\dot{q}}{dt} + O(\Delta t^3) \tag{4.20b}$$

The method is second order temporal accurate when $2\beta - \alpha = 1$. At the start of simulation, one could use trapezoidal scheme for second order accuracy, or more stable backward Euler method (as the value at $n - 1^{th}$ time step is not available). The BDF2 method has better stability properties than trapezoidal rule, and is regularly used for stiff problems. It can also be viewed as an implicit counterpart of the explicit Leap-Frog scheme.

4.2.5 Semi-Implicit schemes

Any implicit method requires solving a linear system of equations. In [1], a linear solver has been developed for the case when the system of equations is represented by a pentadiagonal matrix. From Equation (4.7a), the linear system of equations formed for the variables q_{n+1} is block pentadiagonal instead of pentadiagonal as $q = \begin{bmatrix} \vec{\zeta} \\ \vec{\varphi} \end{bmatrix}$, and each $\vec{\zeta}$ and $\vec{\varphi}$ are represented by their own five point stencils.

As previously mentioned, it is our intention to make use of the linear solver in [1] with minimum modifications possible. For this reason, we split Equation (4.7a) into Equation (4.1a) and Equation (4.1b). Then we derive the corresponding equations for various implicit time integration procedures.

In the first approach, we implicitly advance variable $\vec{\zeta}$, and use the previous time step values of variables $\vec{\varphi}$ and $\vec{\psi}$. The equations are given below:

$$\vec{\zeta}_{n+1} = \vec{\zeta}_n + \Delta t(-S_{\zeta\zeta}\vec{\zeta}_{n+1} - S_{\zeta\varphi}\vec{\varphi}_n - S_{\zeta\psi}\vec{\psi}_n) \tag{4.21a}$$

$$(\frac{I}{\Delta t} + S_{\zeta\zeta})\vec{\zeta}_{n+1} = \frac{1}{\Delta t}\vec{\zeta}_n - (S_{\zeta\varphi}\vec{\varphi}_n + S_{\zeta\psi}\vec{\psi}_n) \tag{4.21b}$$

Please note that variables φ and ψ here are treated explicitly.

After advancing $\vec{\zeta}$, equation for $\vec{\varphi}$ is advanced implicitly.

$$\vec{\varphi}_{n+1} = \vec{\varphi}_n + \Delta t(-S_{\varphi\zeta}\vec{\zeta}_{n+1} - S_{\varphi\varphi}\vec{\varphi}_{n+1} - S_{\varphi\psi}\vec{\psi}_n) \tag{4.22a}$$

$$(\frac{I}{\Delta t} + S_{\varphi\varphi})\vec{\varphi}_{n+1} = \frac{1}{\Delta t}\vec{\varphi}_n - (S_{\varphi\zeta}\vec{\zeta}_{n+1} + S_{\varphi\psi}\vec{\psi}_n) \tag{4.22b}$$

Now, only variable ψ here is treated explicitly. As the value of ζ at $n+1^{th}$ time step is available, we are able to use it. After obtaining the value of both ζ and φ at the new time step, the linear system of equations for $\vec{\psi}$ is solved.

As it is a combination of implicit and explicit scheme, stability of the method is not guaranteed.

Another possibility here is to use a predictor-corrector kind of scheme, described below:

- Advance $\vec{\varphi}$ using an explicit time integration scheme.
- Based on the new value of $\vec{\varphi}$, compute $\vec{\psi}$ by solving linear system of equations.
- Implicitly advance $\vec{\zeta}$ where the values of $\vec{\varphi}$ and $\vec{\psi}$ on the right-hand side of Equation (4.21) are substituted by above computed values.
- Perform implicit correction of $\vec{\varphi}$ and compute $\vec{\psi}$.

This will definitely demand more number computational effort, but will be more stable than Equation (4.21). Similar derivations can be done for higher order methods.

4.3 Symplectic integration

The Symplectic integrators are designed for the numerical solution of Hamiltonian equations given by Equation (2.2). Symplectic integrators ensures time-reversibility and preservation of the symplectic (Ergodic) nature of the equation.

The so-called symplectic Euler method (first order) can be constructed as follows:

$$\zeta_{n+1} = \zeta_n + \Delta t \nabla H_{\varphi}(\zeta_{n+1}, \varphi_n) \tag{4.23a}$$

$$\varphi_{n+1} = \varphi_n + \Delta t \nabla H_{\zeta}(\zeta_{n+1}, \varphi_n) \tag{4.23b}$$

The methods are implicit for general Hamiltonian systems. However if $H(\zeta, \varphi)$ is separable as $H(\zeta, \varphi) = T(\zeta) + U(\varphi)$, it turns out to be explicit.

The Stormer-Verlet schemes are Symplectic methods of order 2. They are composed of the two symplectic Euler methods with step size $\frac{\Delta t}{2}$.

$$\zeta_{n+1/2} = \zeta_n + \frac{\Delta t}{2} \nabla H_\varphi(\zeta_{n+1/2}, \varphi_n) \tag{4.24a}$$

$$\varphi_{n+1} = \varphi_n + \frac{\Delta t}{2} (\nabla H_\zeta(\zeta_{n+1/2}, \varphi_n) + \nabla H_\zeta(\zeta_{n+1/2}, \varphi_{n+1}) \tag{4.24b}$$

$$\zeta_{n+1} = \zeta_{n+1/2} + \frac{\Delta t}{2} \nabla H_\varphi(\zeta_{n+1/2}, \varphi_{n+1}) \tag{4.24c}$$

Implicit mid-point rule:

For a fully implicit method, ζ and φ are combined in one equation represented by variable $\vec{q} = \begin{bmatrix} \vec{\zeta} \\ \vec{\varphi} \end{bmatrix}$. Then the implicit mid-point rule is given by:

$$q_{n+1} = q_n + \Delta t \nabla H(\frac{q_{n+1} + q_n}{2}) \tag{4.25}$$

In the next Chapter, we implement the above time integration schemes for various test cases and analyze their accuracy and stability behaviors.

Chapter 5

Analysis of Time Integration Schemes

5.1 Stability and Accuracy of various time integration schemes

To assess the stability of various implicit time integration procedures, we have considered a simplified problem in one dimension given below.

$$\frac{\partial \zeta}{\partial t} + \mathbf{U}\frac{\partial \zeta}{\partial x} + h\frac{\partial^2 \varphi}{\partial x^2} = 0, \tag{5.1a}$$

$$\frac{\partial \varphi}{\partial t} + \mathbf{U}\frac{\partial \varphi}{\partial x} + g\zeta = 0, \tag{5.1b}$$

Here $\mathbf{U}$ represents the mean current velocity and h represents the depth of the ocean. Both $\mathbf{U}$ and h have been assumed to be uniform in x direction.

Periodic boundary conditions have been assumed for both ζ and φ, and the initial conditions are given as:

- $\zeta(0, x) = \cos(\frac{4\pi x}{L})$
- $\varphi(0, x) = 1$

The central difference scheme has been used for the spatial discretization. It results in following equations:

$$\frac{\partial \vec{\zeta}}{\partial t} + S_{\zeta\zeta}\vec{\zeta} + S_{\zeta\varphi}\vec{\varphi} = 0, \tag{5.2a}$$

$$\frac{\partial \vec{\varphi}}{\partial t} + S_{\varphi\zeta}\vec{\zeta} + S_{\varphi\varphi}\vec{\varphi} = 0, \tag{5.2b}$$

where the discretization matrices are given by three point stencils as given below.

$$
\begin{aligned}
S_{\zeta\zeta} &: \begin{bmatrix} \frac{-U}{2\Delta x} & 0 & \frac{U}{2\Delta x} \end{bmatrix} \\
S_{\zeta\varphi} &: \begin{bmatrix} \frac{h}{\Delta x^2} & \frac{-2h}{\Delta x^2} & \frac{h}{\Delta x^2} \end{bmatrix} \\
S_{\varphi\zeta} &: \begin{bmatrix} 0 & g & 0 \end{bmatrix} \\
S_{\varphi\varphi} &: \begin{bmatrix} \frac{-U}{2\Delta x} & 0 & \frac{U}{2\Delta x} \end{bmatrix}
\end{aligned}
$$

The matrices $S_{\zeta\zeta}$ and $S_{\varphi\varphi}$ are skew- symmetric, matrix $S_{\zeta\varphi}$ is symmetric, whereas $S_{\varphi\zeta}$ is only a diagonal matrix. For the sample problem, following values have been used:

- $x - length$ = L =100 m
- Δx = 0.5 m
- U = 1.0 m/sec
- h = 50 m
- $time - max$ = 100 sec

eFor the combined system of equations represented by Equation (5.2), the maximum eigenvalue obtained is :$\lambda_{max} = 2\sqrt{gh}$. Time step is limited by the CFL condition for the explicit time integration schemes. It is given as:

$$\Delta t \leq \frac{\Delta x}{\lambda_{max}} \leq \frac{\Delta x}{2\sqrt{gh}} = 0.0113 \text{ sec} = \Delta t_{cfl} \tag{5.3}$$

Various time integration schemes as discussed in Chapter 4 have been implemented. In case of Fully Implicit Schemes (Fully Implicit Backward Euler, Fully Implicit Trapezoidal etc.), Equation (5.2) is solved simultaneously. In case of Semi-Implicit schemes, an iterative procedure is followed to solve Equation (5.2).

5.1.1 Benchmark Solution using MATLAB integrators

In order to get the benchmark solution, MATLAB ODE integrators are used. The system of ODE's given by Equation (5.2) are integrated by ODE45 and ODE15s functions. ODE45 is an explicit Runge-Kutta method with adaptive time stepping. The time step is modified based on the error estimate provided. ODE15s is a higher order implicit method. For the benchmark, a relative error estimate of $1e^{-6}$ is used.

Using ODE15s, we obtain the following solution for the the variable ζ.

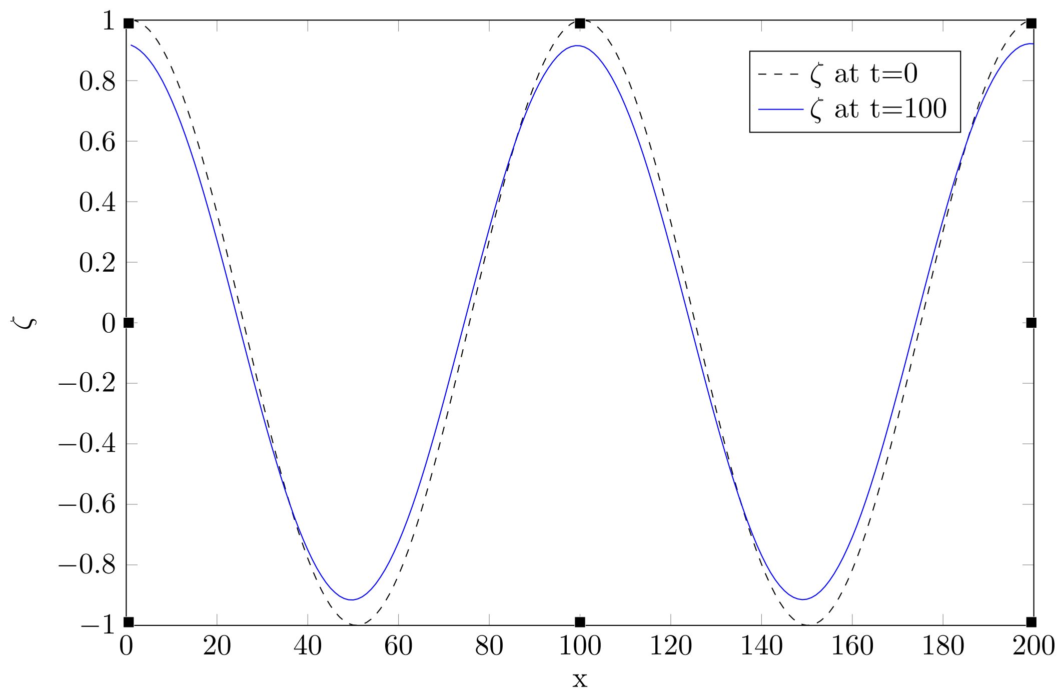


Figure 5.1: ζ variation with length at different times using ODE15s

The result depends on the Initial condition chosen. If the Initial conditions are smooth, as in Figure 5.1, the final solution is also smooth.

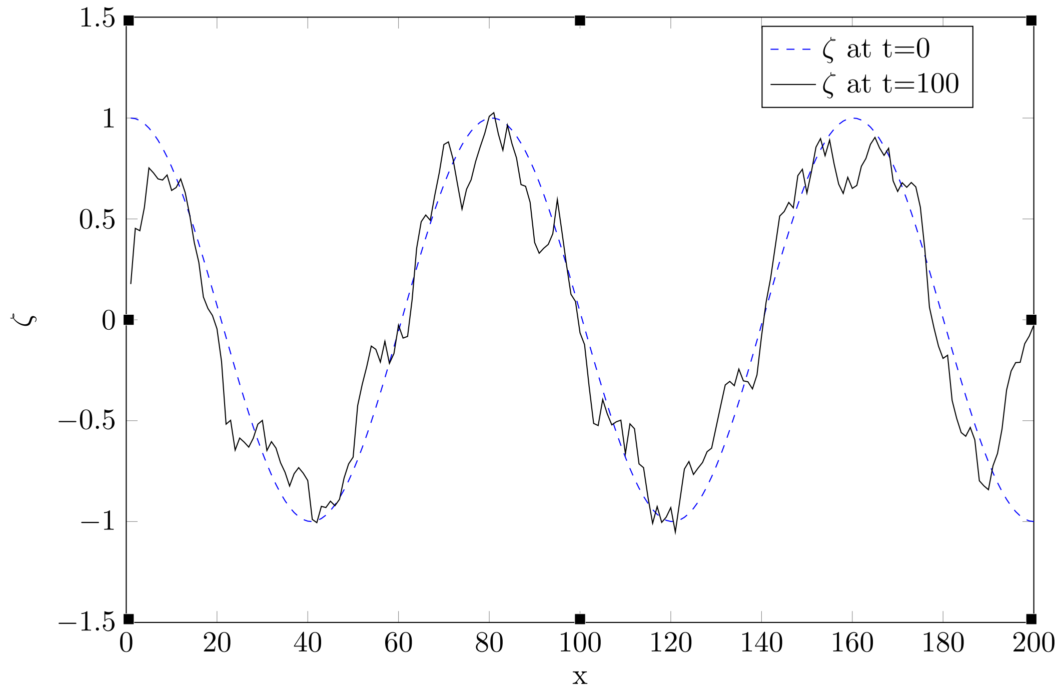


Figure 5.2: ζ variation with length at different times using ODE15s with discontinuity

In case if we introduce some discontinuity in the initial condition as shown in in Figure 5.2 at x=0, wiggles/oscillations are observed with the Central discretization. In order to remove these oscillations, high resolution spatial discretization schemes (flux-limiters or total variation diminishing) are required, which are not considered in the current study.

The comparison of the ODE45 (explicit) and ODE15s (implicit) is shown in Figure 5.3.

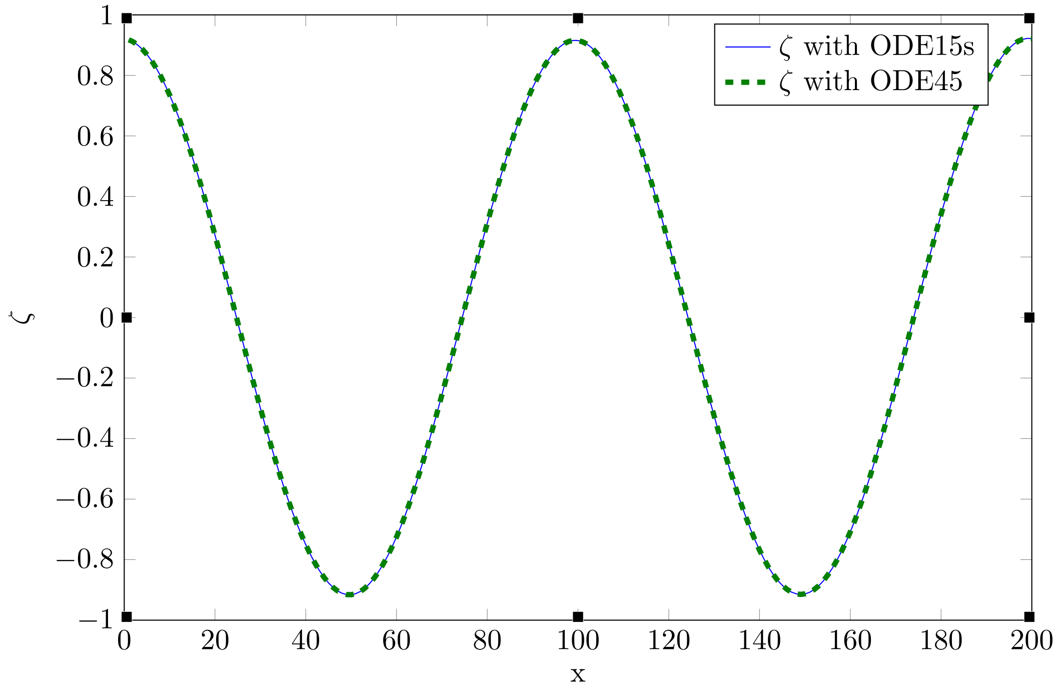


Figure 5.3: ζ variation with length at different times using ODE15s and ODE45

In order to compare the two solutions, we define the norm of the difference between the two vectors. Let x_s be the number of grid points in x direction.

$$Norm(u,v) = \sqrt{\frac{1}{x_s}\sum_{i=1}^{x_s}(u(i) - v(i))^2} \tag{5.4}$$

For the solutions given in Figure 5.3, the norm as given by Equation (5.4) is $2.0500e^{-4}$. We will now use ODE45 as the benchmark for the explicit time integration solvers given in Chapter 4, and ODE15s as the benchmark for the implicit and semi-implicit time integration methods.

5.1.2 Explicit Scheme

The Leap Frog scheme has been used as the explicit time integration scheme as described in the Chapter 4. For the first time step as the value at two previous time steps in unknown, the Forward Euler method is used. The Leap Frog scheme requires the CFL condition for the stability of the time integration. For time step upto Δt_{cfl}, the method is stable. It becomes unstable when the time step is equal or greater than Δt_{cfl}.

For various time steps, simulation was run in Matlab for 100 seconds, and the results plotted in Figure 5.4.

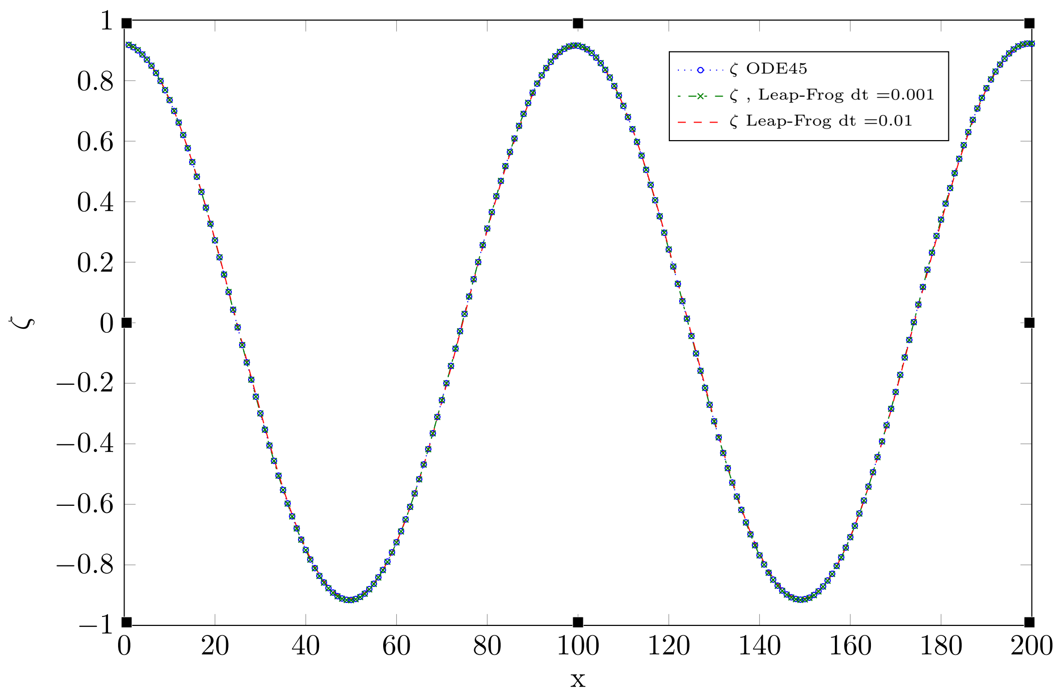


Figure 5.4: Comparison of ζ variation with length for ODE45 vs Explicit Leap Frog

It is known that the Leap-Frog method does not suffer from phase lag or numerical dispersion (an analytical behaviour of the Leap-Frog method is discussed in Chapter 11). We can see that the solution is accurate for the time steps below Δt_{CFL}. In order to compare the results, the norm defined in Equation (5.4) is used and tabulated below.

Table 5.1: Leap Frog vs ODE45, Error Norm

Time step (sec)	Error Norm
0.0113 (Δt_{cfl})	0.0370
0.01	0.0229
0.001	0.0007

5.1.3 Fully Implicit Schemes

Equation (5.2) is solved using three implicit schemes : The Backward Euler scheme, the Trapezoidal scheme and the Backward Differentiation Formulae (BDF2). It is observed that all the methods are unconditionally stable. For a linear system of equations (as in our case), the Trapezoidal method is equivalent to the implicit mid-point rule which is a Symplectic integrator. At larger time steps, the results are not very accurate and the solution is damped. The following subsections give the plots of the three schemes for varying time steps.

Fully Implicit Backward Euler Method

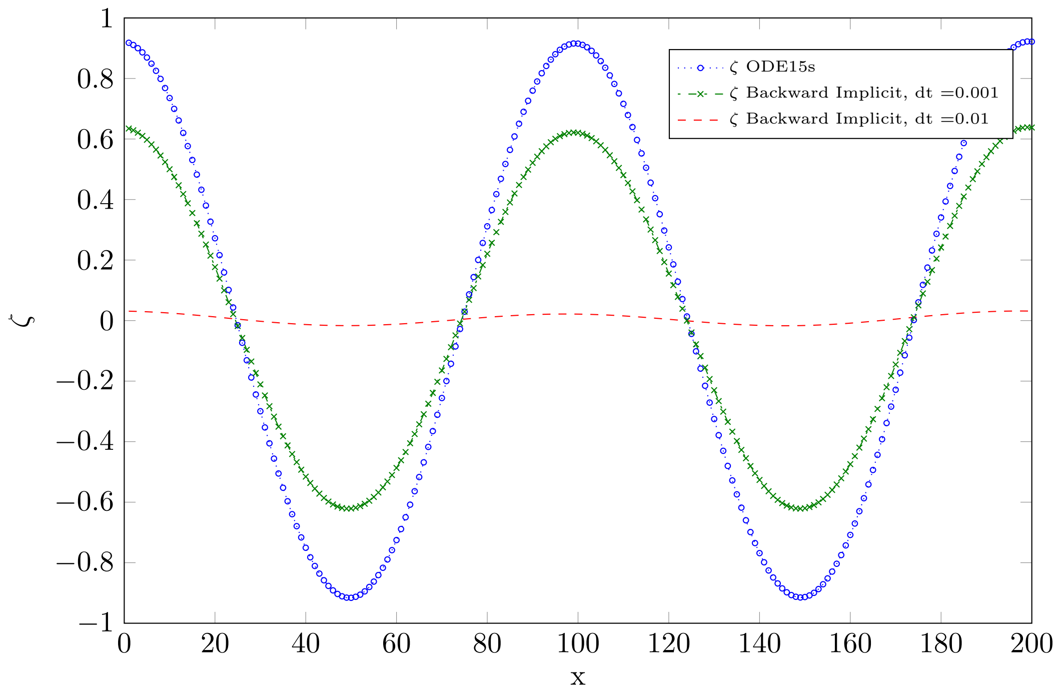


Figure 5.5: Comparison of ζ variation with length for ODE15s vs Implicit Backward Euler

Even though the Implicit Backward Euler method is unconditionally stable, it suffers from very large damping at larger time steps as observed in the Figure 5.5. As the time step is increased to 0.1 seconds, the solution dies out very quickly, which is unwanted for the equations represented by the Symplectic system where total energy should be conserved. The error norm is given in Table 5.2.

Fully Implicit Trapezoidal Method

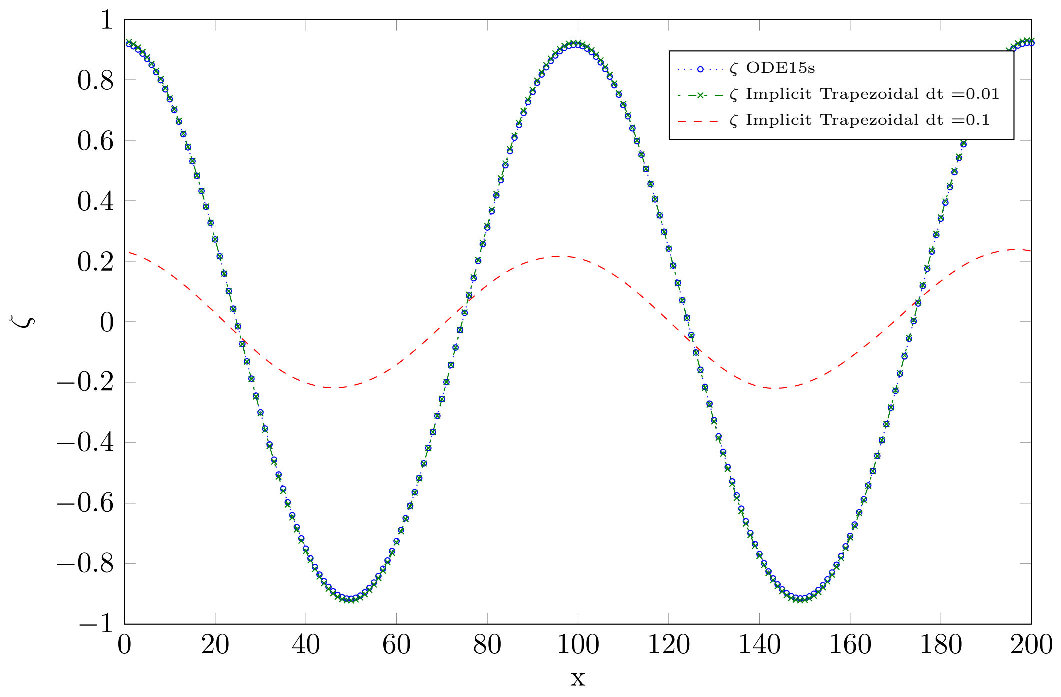


Figure 5.6: Comparison of ζ variation with length for ODE15s vs Implicit Trapezoidal Method

As expected, the Implicit Trapezoidal scheme provides better results than Backward Euler. For larger time steps (0.1 sec), the solution is not very accurate. Some phase error is observed around $x_s = 140$, which results in large error norm for $\Delta t = 0.1$ sec. Along with the phase error, some damping is also observed. As later studied in Section 5.2, this reduction in amplitude is not damping, but results from amplitude variation with change in phase. The phase error reduces as we reduce the time step (around 0.01 second).

Fully Implicit BDF2 Method

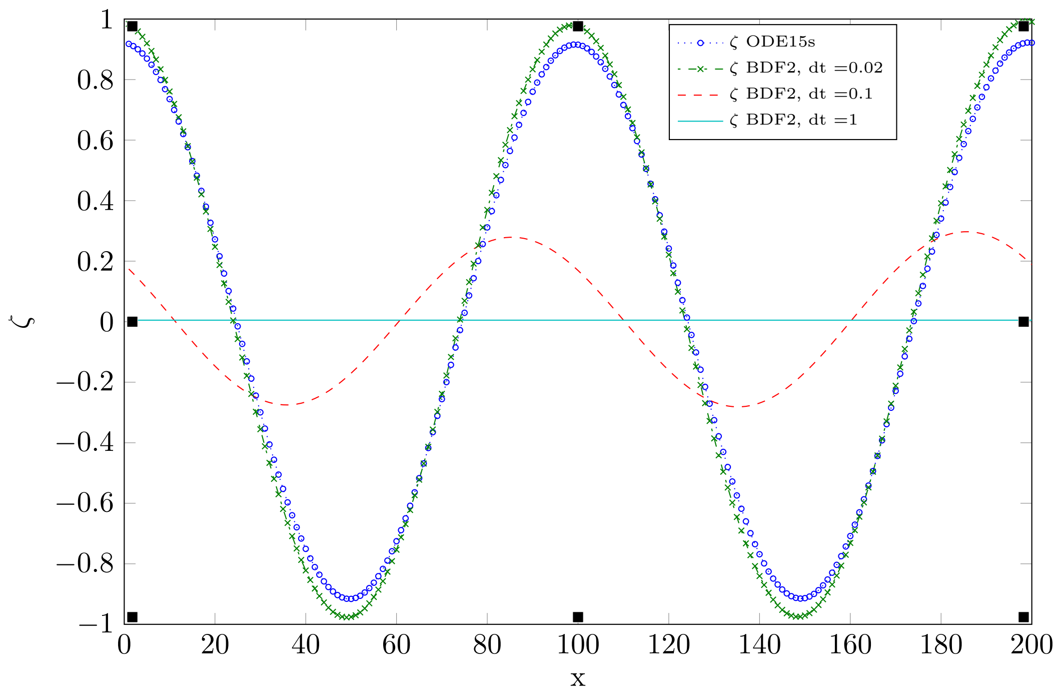


Figure 5.7: Comparison of ζ variation with length for ODE15s vs Implicit BDF2 Method

BDF2 is a multi-step second order method, and is not Symplectic in nature. As can be seen from Figure 5.7, the solution suffers from very large numerical damping at larger time steps.

In Table 5.2, the error norm for various implicit schemes at different time steps is provided. In terms of accuracy, the Trapezoidal method is better than the explicit Leap Frog scheme and BDF2 for a given time step.

Table 5.2: Implicit Methods vs ODE15s, Error Norm

Time step (sec)	Backward Euler-Norm	Trapezoidal-Norm	BDF-Norm
0.1	0.672	0.470	0.673
0.01	0.672	0.0136	0.0364
0.001	0.387	0.0003	0.007

5.1.4 Semi Implicit Schemes

Semi Implicit schemes of the predictor corrector nature were proposed in Chapter 4 in order to make use of the existing structure of the RRB-k solver. The following algorithm is applied while using the Semi-Implicit scheme.

Here itermax represents the maximum sub-iterations performed for the corrector step of the algorithm. Apart from the time step, itermax is important for the stability considerations. Larger value of itermax should typically allow usage of large timestep. After performing numerical experiments, itermax vs maximum allowable timestep has been computed and tabulated below.

Algorithm 5.1 Semi Implicit Algorithm to solve Equation (5.2)

```
Input $\vec{\zeta}_0$, $\vec{\varphi}_0$ (Initial Vector), $\Delta t$, itermax
  while $t \leq tmax$ do
    while $iter \leq itermax$ do
      if $iter == 1$ then
        Step 1.
        if $t == 1$ then
          Advance $\vec{\varphi}$ explicitly using Forward Euler method on Equation (5.2)
        else
          Advance $\vec{\varphi}$ explicitly using Leap Frog method on Equation (5.2)
        end if
      else
        Advance $\vec{\zeta}$ implicitly using Implicit Trapezoidal Scheme with the implicit value of $\vec{\varphi}$
        taken from Step1.
        Correct $\vec{\varphi}$ implicitly using Implicit Trapezoidal Scheme with implicit value of $\vec{\zeta}$ taken
        from above Step.
      end if
    end while
  end while
```

Table 5.3: Semi Implicit Methods: itermax vs. Δt

itermax	Δt maximum	$\frac{\Delta t}{\Delta t_{cfl}}$
3	0.019	1.68
5	0.020	1.77
7	0.021	1.86
9	0.021	1.86
11	0.021	1.86
19	0.022	1.95
21	0.022	1.95
23	0.022	1.95
100	0.022	1.95

It is observed that the time step can be increased upto two times the Δt_{CFL} by increasing the number of iterations. The error norm for the Semi-implicit schemes follow the similar trend as that of the Explicit Leap-Frog scheme. For the parameters and the numerical techniques used for the test case, Semi-Implicit methods provides a little advantage over Explicit methods in terms of Stability.

Impact of depth h

The depth h appears in the maximum eigenvalue of the system, and influences the stability criteria of any scheme. In order to ascertain the impact of the depth on the stability of the Semi-implicit schemes, a sensitivity analysis has been carried out. The CFL number is given by Equation (5.3) and varies with the depth h.

For a given itermax, it is observed that the ratio between the maximum Δt achieved and the Δt given by CFL number remains the same.

5.2 Study of Implicit Trapezoidal Scheme

As discussed in the previous section, the implicit Trapezoidal scheme provides a stable solution without much damping as compared to other methods. Although it suffers from the phase shift or phase error at larger time steps resulting in larger error norms.

In this section, the impact of phase difference on the amplitude of the solution for the coupled set of equations and non-coupled set of equations is discussed.

In Figure 5.8, the values of variable ζ is plotted at 50 seconds. The system of equations have been decoupled by setting $h = 0$ and $g = 0$. No phase error is observed for this case even when large time steps are used. As the initial solution do not contain sharp discontinuities, Central discretization method does not introduce oscillations and the phase errors are minimal. This changes when we introduce the coupling between the two equations by setting $h = 50$ and $g = 9.81$.

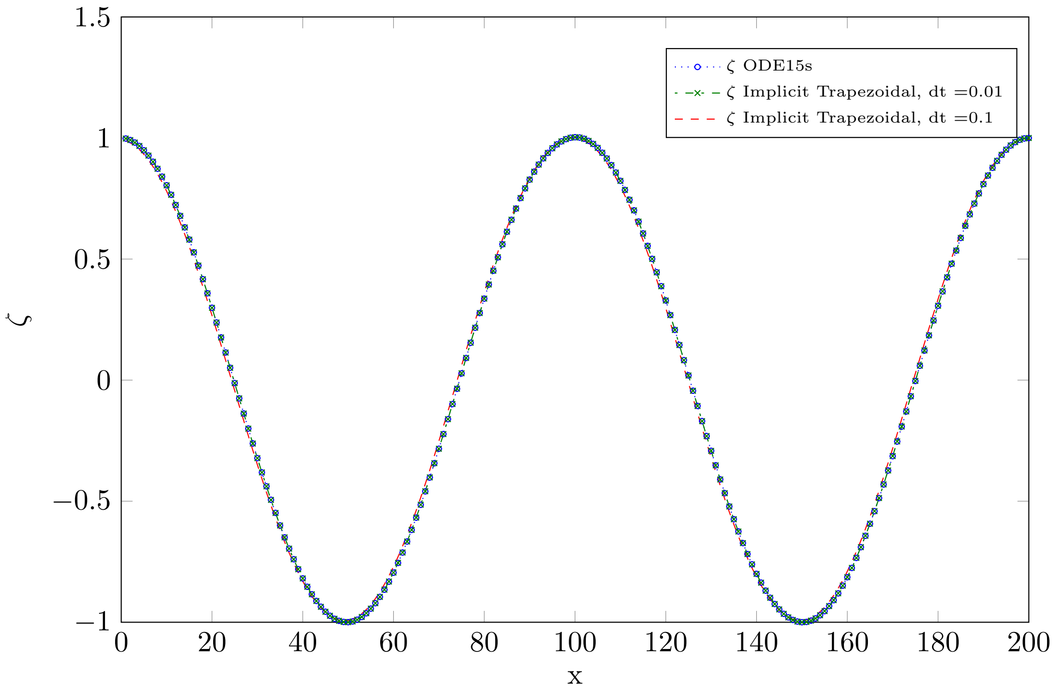


Figure 5.8: Implicit Trapezoidal method without Coupling at t = 50sec for various time steps

For the coupled equations (Figure 5.9), it is observed that the amplitude of the solution does not remain constant due to the impact of the source term, and varies with the phase. This impact can be seen in Figure 5.9 where the benchmark solution is plotted at three different time intervals. Without any coupling between the equations, the solution should travel with the wave speed without changing the shape or the amplitude. With the coupling, a change in amplitude is seen at different time intervals.

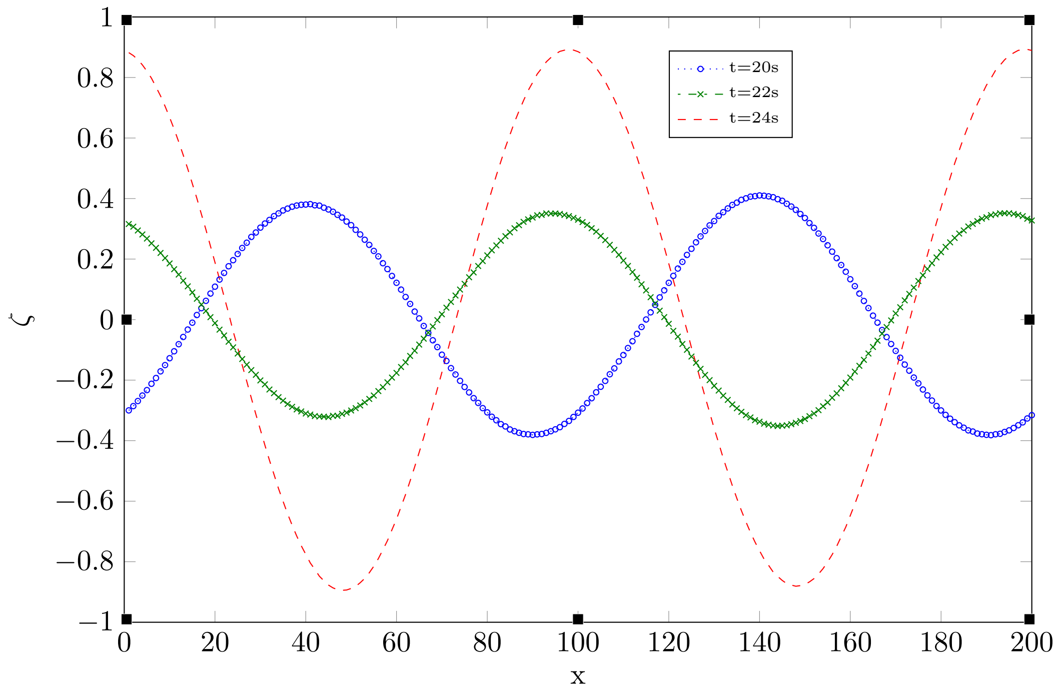


Figure 5.9: Solution of coupled equation at t = 20,22 and 24 sec with Matlab ODE integrator

This change in the amplitude with the phase for the coupled system causes large error norms for Implicit Trapezoidal method. As Implicit Trapezoidal method induces a phase error, the change in the phase results in a change in amplitude. This is reflected in the Figure 5.10

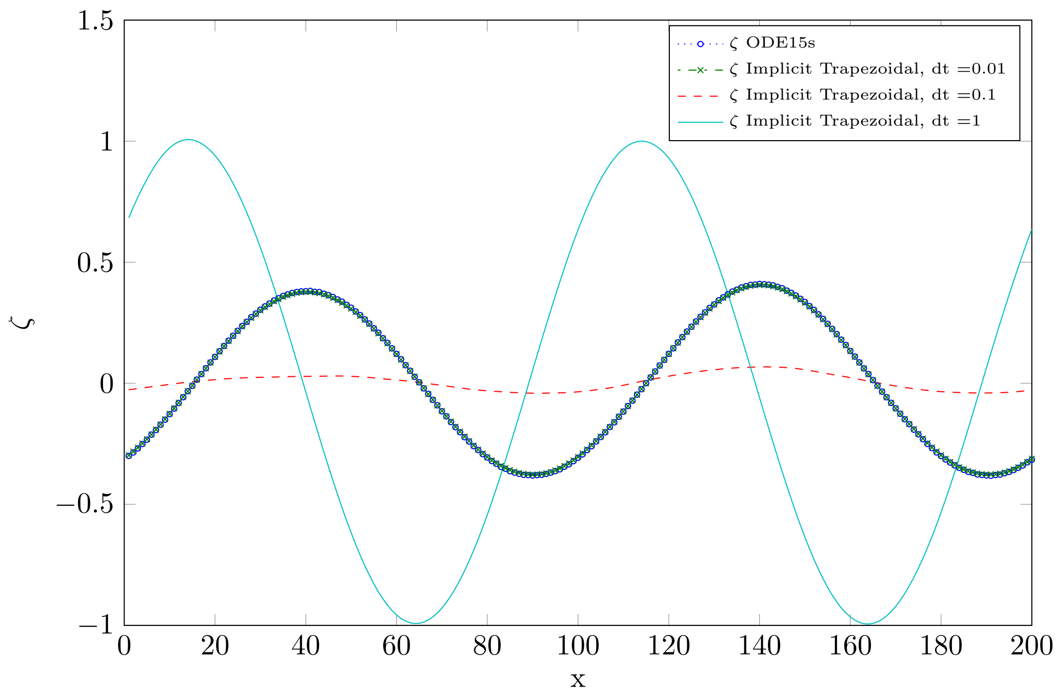


Figure 5.10: Implicit Trapezoidal method with Coupling at t = 20 sec for various time steps

For time step $\Delta t = 0.01s$, the solution is accurate and does not suffer from phase error. For larger timer steps, both phase change and amplitude variation is observed. It is important to observe that, for $\Delta t = 0.1s$, the solution appears to be already damped at time 20s. If there was damping in the solution, then it would be greater for larger time steps, which is not seen for $\Delta t = 1s$. Instead the amplitude at $\Delta t = 1s$ is greater than the benchmark solution.

It could be argued that the larger Δt is impacting the stability of the solution. In order to check this, the solution is plotted for time 100 sec in Figure 5.11.

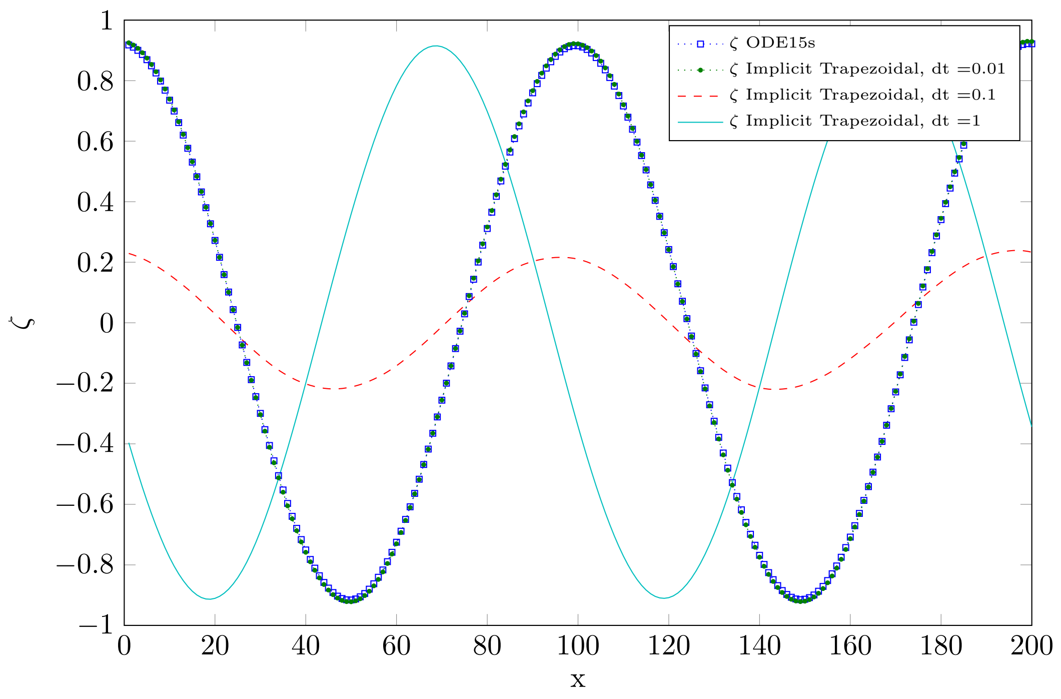


Figure 5.11: Implicit Trapezoidal method with Coupling at t = 100 sec for various time steps

It is observed that the solution remains stable, and only phase error and amplitude difference is observed for larger time steps.

5.2.1 Higher Order Schemes and Impact on Phase Error

Higher order (Two stage, Fourth ordered) Symplectic Implicit Runge Kutta scheme has been implemented to analyze the impact on the phase error due to the larger time steps. These methods are computationally more intense, but provides better accuracy. Given the coupled set of equations as:

$$\frac{d\mathbf{q}}{dt} + L\mathbf{q} = 0$$

we have

$$\frac{q^{n+1} - q^n}{\Delta t} = \frac{1}{2}(K_1 + K_2) \tag{5.5}$$

where

$$K_1 = -L(q^n + \Delta t/4K_1 + (1/4 + \sqrt{3}/6)\Delta t K_2)$$
$$K_2 = -L(q^n + (1/4 - \sqrt{3}/6)\Delta t K_1 + 1/4\Delta t K_2)$$

At every time step, above system of equation is simultaneously solved and then plugged into Equation (5.5). Please note that the coefficients are chosen such that the method remains Symplectic in nature.

The results are shown in Figure 5.12.

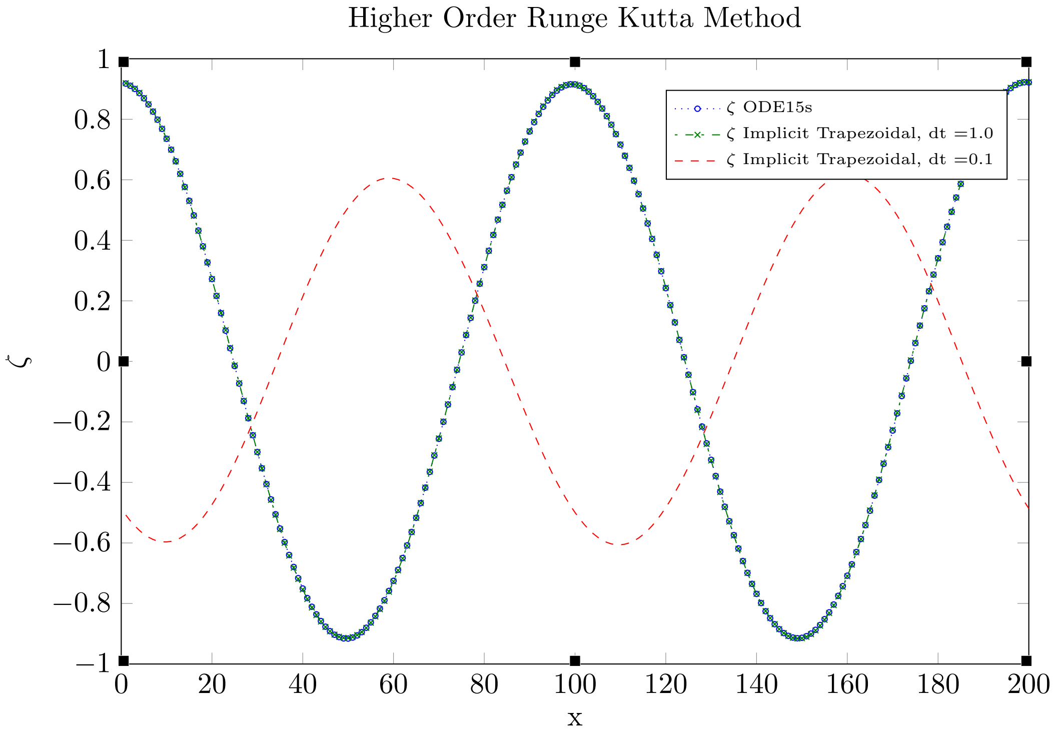


Figure 5.12: Higher order Runge Kutta method at time 100s

It is observed that time step $\Delta t = 0.1s$, the solution is accurate and there is no phase error. As compared to the Implicit Trapezoidal method, higher order Runge Kutta method is more accurate. A phase error is observed when the time step is further increased.

5.2.2 Impact of Initial Conditions

In a real case scenario, the incoming waves are generated by mixing various waves of different frequencies and amplitudes. The initial condition in this case is chosen as:

$$\zeta^0(x) = 0.25(\sum_{i=0}^{3} cos(\frac{2^i 4\pi x}{L})) \tag{5.6}$$

For the initial conditions with single wave, the solution was accurate for $\Delta t = 0.01s$. The solution for same $\Delta t = 0.01$ but the mixed initial conditions is plotted in Figure 5.13.

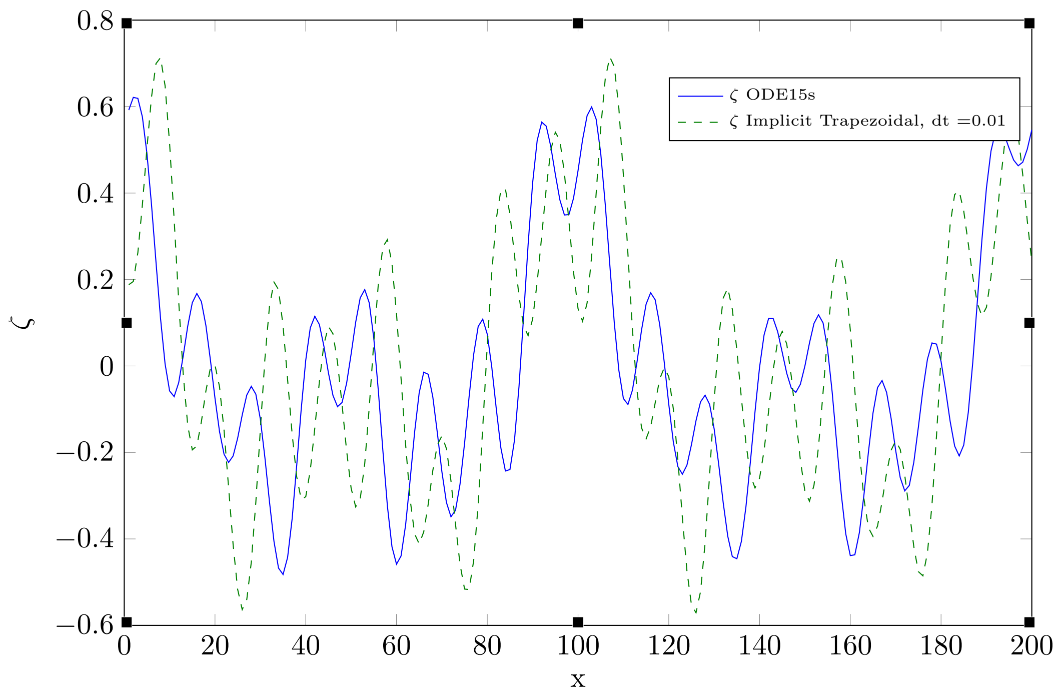


Figure 5.13: Implicit Trapezoidal Method with mixed initial conditions, Δt =0.01s

It is observed that the solution is not accurate anymore. This can be attributed to high frequency waves. The phase error for Implicit Trapezoidal schemes is generally proportional to square of the frequency of the wave.

Instead of a mix of waves, simulation is carried with initial condition as $\zeta^0(x) = cos(\frac{32\pi x}{L})$. The result for $\Delta t = 0.01s$ is plotted in Figure 5.14. Phase error is observed even for a single initial wave with high frequency at $\Delta t = 0.01s$. This implies that for high frequency waves, a smaller time step is required to maintain accuracy.

Spatial discretization also influences the solution with the high frequency waves. As a rule of thumb, a minimum of 15 grid points per wavelength should be used to accurately represent the wave. In our study, we have used 12.5 gridpoints per wavelength. Increasing the number of grid points reduces Δx, and thus will result in lower CFL limit. In general, it is the low frequency waves, with larges wavelengths which influences the Ships most. The Δx should be chosen to represent these low frequency waves accurately.

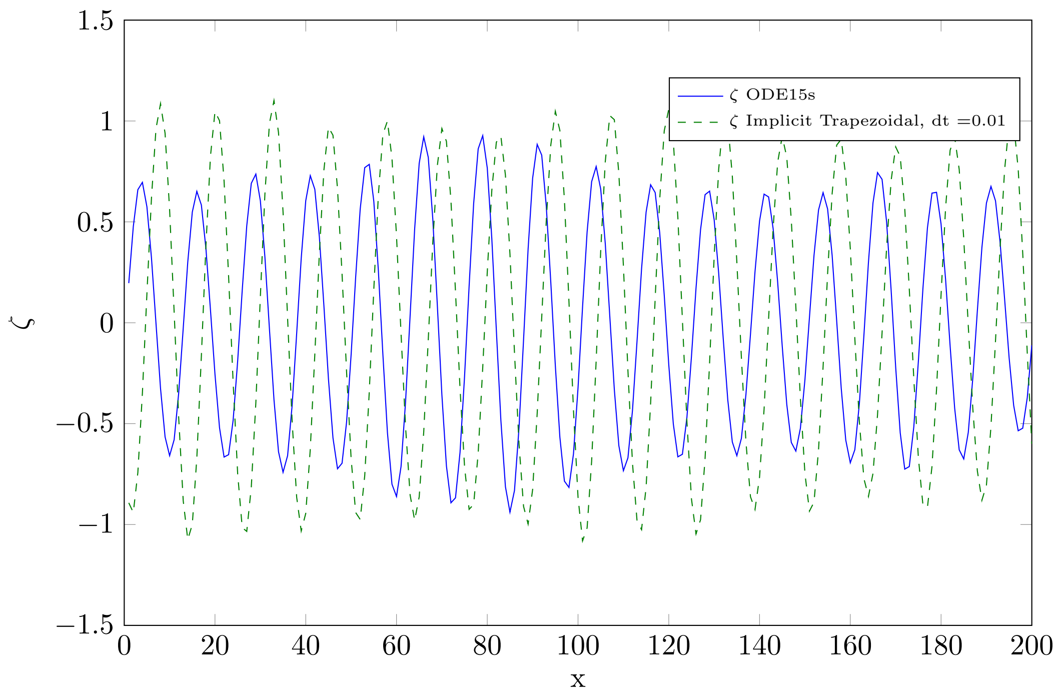


Figure 5.14: Implicit Trapezoidal Method with high frequency inlet conditions, Δt =0.01s

5.2.3 Impact of Boundary Conditions

Periodic boundary conditions are prone to accumulation of phase error as the time grows. In this section, we try to analyze the impact on the solution when different boundary conditions are used.

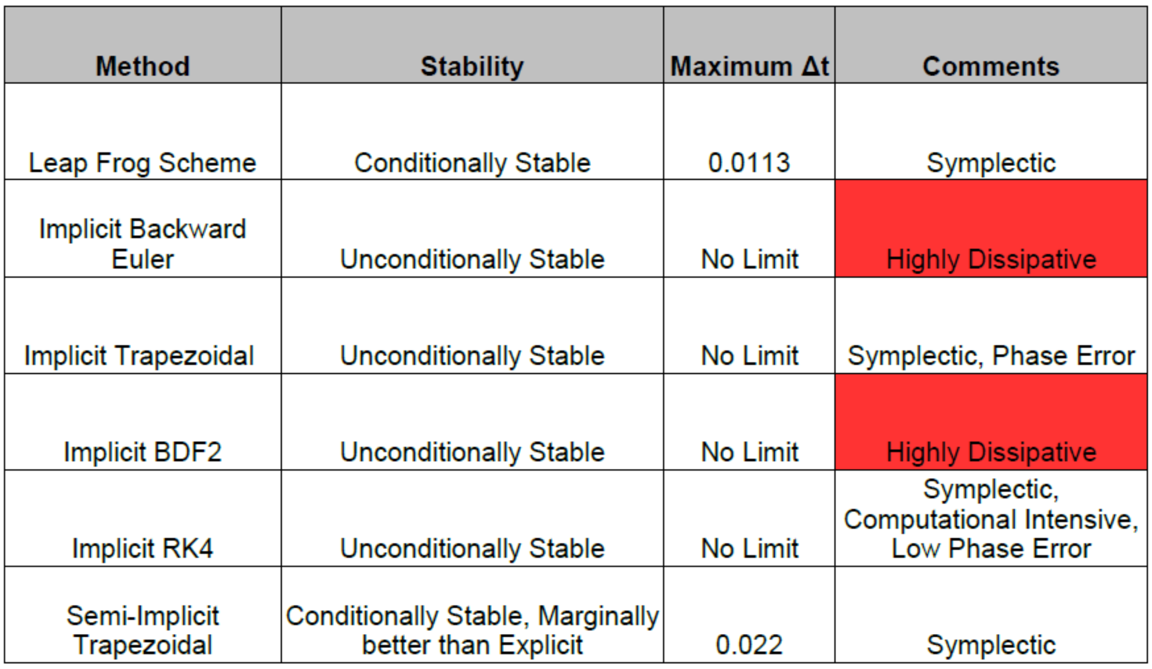
We use the following:

- $\zeta(0,t) = 1.0$: Dirichlet Boundary Condition.
- For numerics, $\zeta_{N+1} = \zeta_N$ where N is the node number at right boundary. This is also equivalent to the Neumann boundary condition.
- Neumann Boundary Condition for φ

We again observe phase error for larger time steps. Though we did not study the difference between the phase error for a Periodic boundary condition and Neumann boundary condition.

5.2.4 Summary

A summary of the methods is provided below.



Method	Stability	Maximum Δt	Comments
Leap Frog Scheme	Conditionally Stable	0.0113	Symplectic
Implicit Backward Euler	Unconditionally Stable	No Limit	Highly Dissipative
Implicit Trapezoidal	Unconditionally Stable	No Limit	Symplectic, Phase Error
Implicit BDF2	Unconditionally Stable	No Limit	Highly Dissipative
Implicit RK4	Unconditionally Stable	No Limit	Symplectic, Computational Intensive, Low Phase Error
Semi-Implicit Trapezoidal	Conditionally Stable, Marginally better than Explicit	0.022	Symplectic

Figure 5.15: Comparison of various time integration schemes and stability estimates

5.3 Impact of Spatial Discretization

As discussed in Section 5.1.1, wiggles/oscillations are observed in ζ when the initial condition has a discontinuity. Following are the possible reasons.

- Spatial Discretization
- Coupling of equations

In order to analyze the impact of the coupling on the oscillations, Equation (5.2) is decoupled by setting $h = 0$ and $g = 0$. This results into two decoupled wave equations, whose analytical solution is available. Let the initial condition be given as $\zeta^0(x)$ and $\varphi^0(x)$. Then the solution at (x, t) is given by :

$$\zeta(x,t) = \zeta^0(x - Ut)$$
$$\varphi(x,t) = \varphi^0(x - Ut)$$

Based on the above equations, if the initial condition has a discontinuity, then it should propagate with time without changing any shape. Spatial discretization usually impacts the solution of the wave equation in the case of discontinuities or shock-waves.

Again, equations are discretized using Central discretization and solved using the Matlab benchmark code (ODE15s). The results obtained for ζ are given in Figure 5.16.

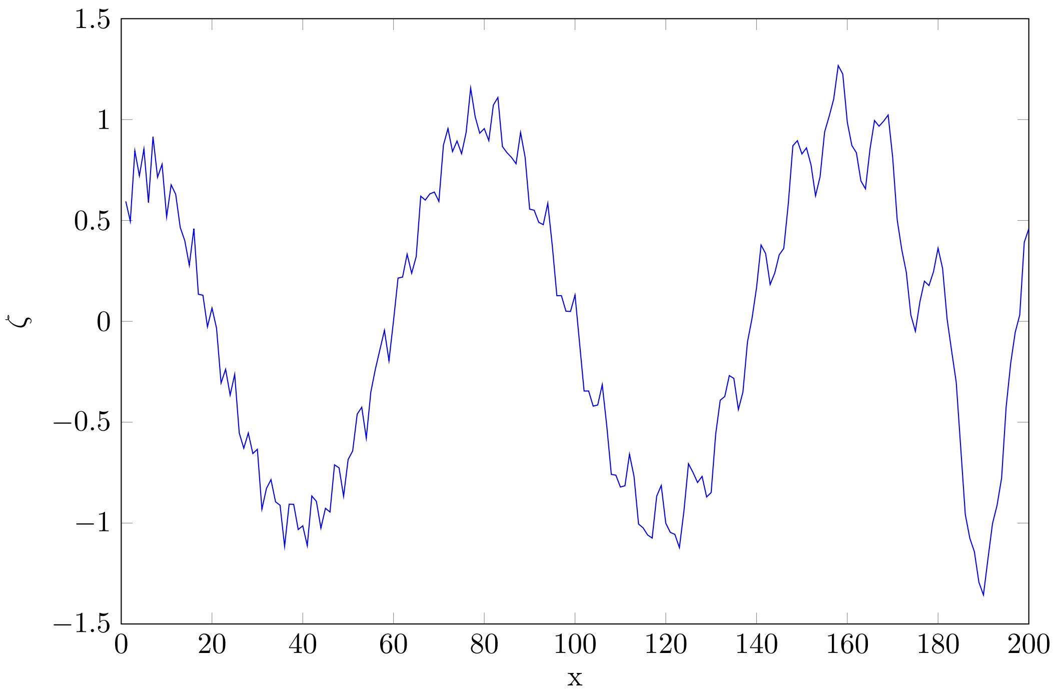


Figure 5.16: ζ for decoupled wave equation after one time period with Central discretization and with discontinuity in the initial condition

Wiggles and oscillations are still present in the solution, which leads us to explore the other possibility of Spatial discretization. Whereas Central discretization causes oscillations around the discontinuity, the error introduced by the Upwind discretization can be classified as follows:

- Smearing: The length of Smearing after the discontinuity.
- Phase Error

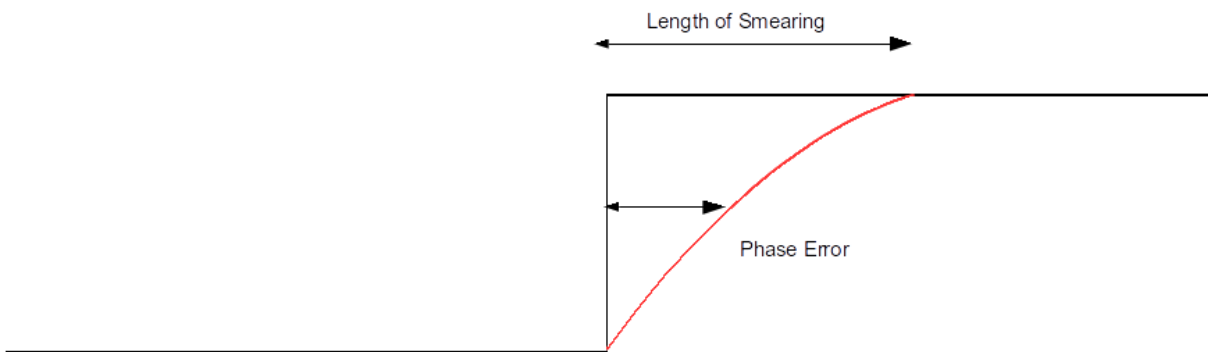


Figure 5.17: Classification of Errors induced by Upwind Discretization

In Figure 5.17, the anlaytical solution is shown as the black line, whereas the solution from the Upwind discretization is shown in red.

Instead of Central discretization, the Upwind scheme is now used to discretize Equation (5.1). This results in following discretization stencils:

$$
\begin{aligned}
S_{\zeta\zeta} &: \begin{bmatrix} \frac{-U}{\Delta x} & \frac{U}{\Delta x} & 0 \end{bmatrix} \\
S_{\zeta\varphi} &: \begin{bmatrix} \frac{h}{\Delta x^2} & \frac{-2h}{\Delta x^2} & \frac{h}{\Delta x^2} \end{bmatrix} \\
S_{\varphi\zeta} &: \begin{bmatrix} 0 & g & 0 \end{bmatrix} \\
S_{\varphi\varphi} &: \begin{bmatrix} \frac{-U}{\Delta x} & \frac{U}{\Delta x} & 0 \end{bmatrix}
\end{aligned}
$$

Using Matlab ODE15s and the above discretization stencils, the obtained solution is plotted in Figure 5.18:

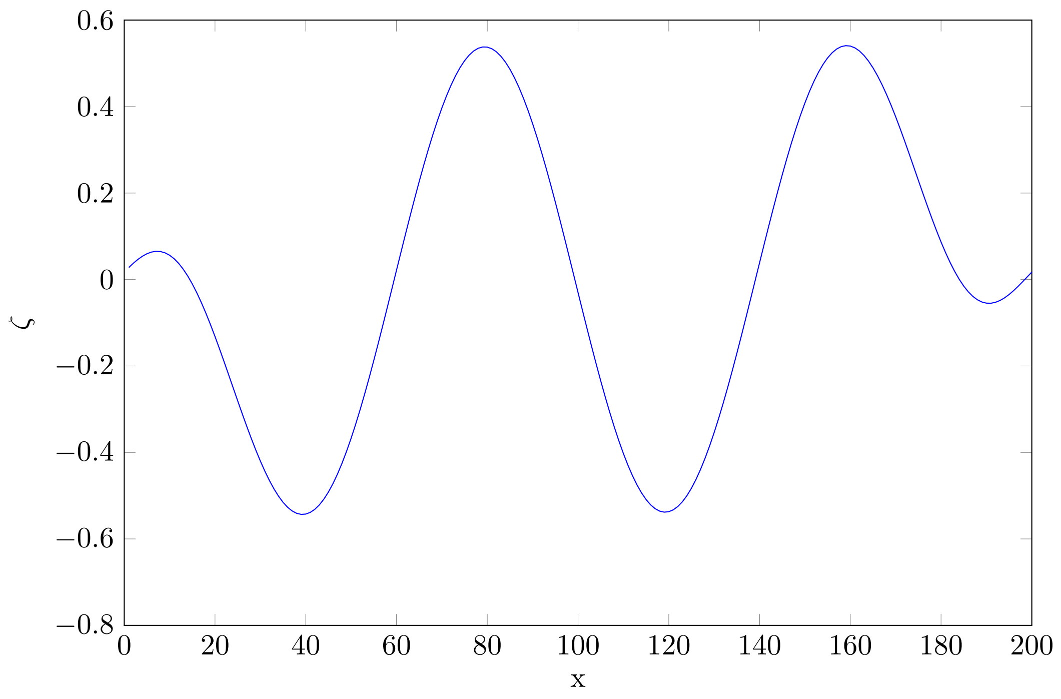


Figure 5.18: ζ for decoupled wave equation after one time period with Upwinding discretization

In Figure 5.18, we observe that the solution is damped, but no oscillations or wiggles are present in the solution. Also, larger damping at $x = 0$ is observed which can be attributed to the discontinuity.

Damping is a characteristic of the Upwinding discretization and can be removed by adopting higher order or higher resolution methods. In order to confirm that the wiggles are generated only because of the central discretization scheme and not because of the coupling, we run the simulation with the Upwinding discretization and by setting $h = 50$ and $g = 9.81$, which again gives us coupled set of equations. Again running the simulation with Matlab ODE15s integrator, we obtain the following result:

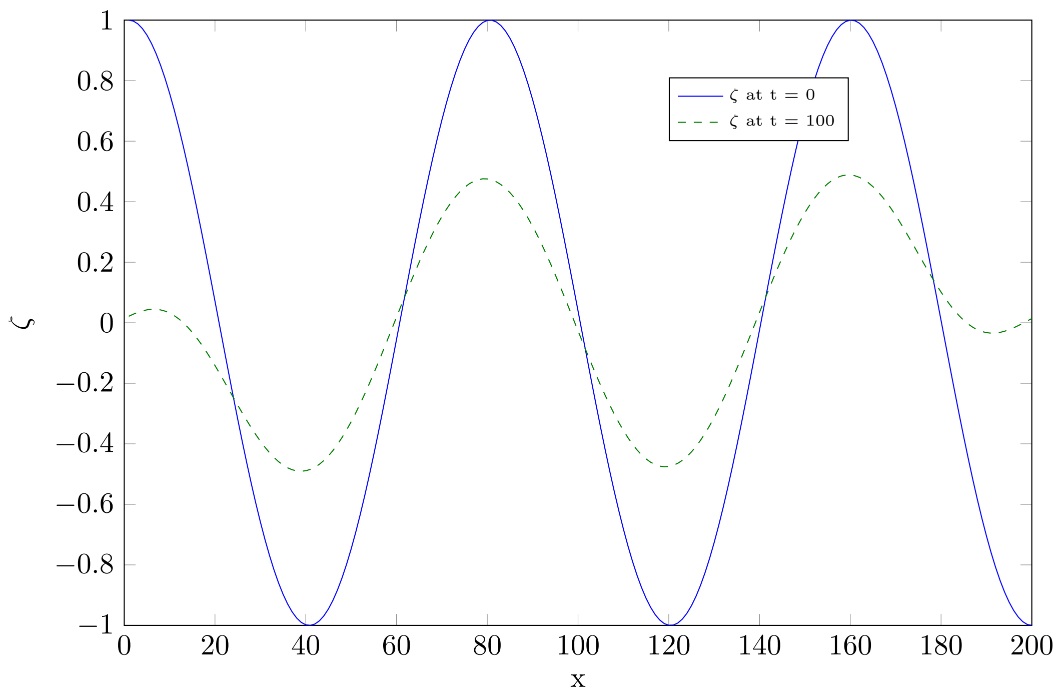


Figure 5.19: ζ for Coupled wave equation after one time period with Upwinding discretization

In Figure 5.19, for the coupled equations, the Upwinding discretization removes the wiggles and also results in a smooth but damped solution.

We conclude that the wiggles are generated in the simulation if a discontinuity is present in the initial condition, with the Central discretizations scheme. However, it does not impact the stability of the Semi-implicit schemes.

5.3.1 Impact of Grid size on Accuracy

One of the goals of the current project is to make the mesh finer near the regions surrounding a ship or obstacles. As we have seen that varying the time step impacts the accuracy of the solution, we will try to assess the time step requirements for similar accuracy for different grid sizes.

Simulations are run with the implicit Trapezoidal scheme, and varying grid spacing and adjusting time step. As the Implicit Trapezoidal method is second order accurate, reduction in grid size implies reduction in time step if the CFL number is not changed. Thus the accuracy of the solution improves by Δt^2, and a better benchmark is required to compare the solution. Again the Matlab ODE15s method is used for the benchmarking along with an absolute error estimate of $1e^{-9}$ as compared to the previous absolute error of $1e^{-6}$. The results obtained are indicated in Table 5.4

Δx	**CFL Number**	**Error Norm**
2	1.8	0.717622
	0.9	0.235133
	0.4	0.053416
1	1.8	0.201493
	0.9	0.046899
	0.4	0.015783
0.5	1.8	0.011764
	0.9	0.0063474
	0.4	0.004944
0.25	1.8	0.005404
	0.9	0.005445
	0.4	0.003001

Table 5.4: Error Norm for various Grid sizes and CFL numbers

For the same CFL number, an error reduction from 0.71 to 0.2 is observed when the grid spacing is reduced from 2 to 1. This approximate factor of 4 is attributed to the reduction in time step by two times and the second order temporal accuracy. Another observation is that, for the combination of small grid size ($\Delta x = 0.25$) and small Δt, the error norm does not improve with further reduction in Δt.

5.4 Stability Analysis with Vertical Structure ψ included

Based on the observations in Section 5.1.3, the fully Implicit Trapezoidal scheme provides the best stability properties for the test case given by Equation (5.1). We still need to evaluate the impact of the vertical structure ψ which is given by an elliptic equation and does not include a time derivative term. To assess the impact of this additional variable and equation, the following test case has been considered.

$$\frac{\partial \zeta}{\partial t} + \mathbf{U}\frac{\partial \zeta}{\partial x} + h\frac{\partial^2 \varphi}{\partial x^2} - h\mathcal{D}\frac{\partial^2 \psi}{\partial x^2} = 0, \tag{5.7a}$$

$$\frac{\partial \varphi}{\partial t} + \mathbf{U}\frac{\partial \varphi}{\partial x} + g\zeta = 0, \tag{5.7b}$$

$$\mathcal{M}\psi + hD\frac{\partial^2 \varphi}{\partial x^2} - \mathcal{N}\frac{\partial^2 \psi}{\partial x^2} = 0, \tag{5.7c}$$

Three additional discretization matrices are formulated. The model parameters $\mathcal{M}$,$\mathcal{N}\mathcal{D}$ are derived from the vertical structure shape. For the linearized parabolic model we obtain,

$$\mathcal{D} = \frac{1}{3}h, \quad \mathcal{M} = \frac{1}{3}h, \quad \textbf{and } \mathcal{N} = \frac{2}{15}h^3 \tag{5.8}$$

The discretization matrices are given as:

$$\begin{aligned}
S_{\zeta\zeta} &: \begin{bmatrix} \frac{-U}{2\Delta x} & 0 & \frac{U}{2\Delta x} \end{bmatrix} \\
S_{\zeta\varphi} &: \begin{bmatrix} \frac{h}{\Delta x^2} & \frac{-2h}{\Delta x^2} & \frac{h}{\Delta x^2} \end{bmatrix} \\
S_{\varphi\zeta} &: \begin{bmatrix} 0 & g & 0 \end{bmatrix} \\
S_{\varphi\varphi} &: \begin{bmatrix} \frac{-U}{2\Delta x} & 0 & \frac{U}{2\Delta x} \end{bmatrix} \\
S_{\zeta\psi} &: \begin{bmatrix} \frac{h^2}{3\Delta x^2} & \frac{-2h^2}{3\Delta x^2} & \frac{h^2}{3\Delta x^2} \end{bmatrix} \\
S_{\psi\varphi} &: \begin{bmatrix} \frac{h^2}{3\Delta x^2} & \frac{-2h^2}{3\Delta x^2} & \frac{h^2}{3\Delta x^2} \end{bmatrix} \\
S_{\psi\psi} &: \begin{bmatrix} \frac{-2h^3}{15\Delta x^2} & \frac{4h^3}{15\Delta x^2} & \frac{-2h^3}{15\Delta x^2} \end{bmatrix}
\end{aligned}$$

Periodic boundary conditions for ψ has been assumed. The model parameters h, U and g remain the same as in the case of two equation model.

5.4.1 Benchmark

Equations for ζ and φ are coupled and represented as a Lumped model. It contains terms from ψ which is explicitly not known as it depends on φ.

Let the lumped equation be represented as :

$$\frac{d\vec{q}}{dt} + L\vec{q} = C\vec{\psi} \tag{5.9}$$

where $L = \begin{bmatrix} S_{\zeta\zeta} & S_{\zeta\varphi} \\ S_{\varphi\zeta} & S_{\varphi\varphi} \end{bmatrix}$ and $C = -\begin{bmatrix} S_{\zeta\psi} \\ S_{\varphi\psi} \end{bmatrix}$

The set of differential algebraic equations given by Equation (5.7) can be solved by the Matlab ODE15s solver, an implicit method capable of solving stiff problems and DAE's. The derivatives in time are described as :

$$\begin{bmatrix} \frac{d\vec{q}}{dt} \\ \frac{d\vec{\psi}}{dt} \end{bmatrix} = \begin{bmatrix} -L\vec{q} + C\vec{\psi} \\ -S_{\psi\varphi}\vec{\varphi} - S_{\psi\psi}\vec{\psi} \end{bmatrix} \tag{5.10}$$

In order to solve the DAE by Matlab ODE15s, an additional input given by $M = \begin{bmatrix} I_{2x_s \times 2x_s} & 0 \\ 0 & 0 \end{bmatrix}_{3x_s \times 3x_s}$ is required, where x_s is the number of grid points. This matrix M, also called as the Singular Matrix characterizes the DAE. One has to be careful while chosing the initial conditions for the system of DAE's. An unique solution is obtained when the initial conditions also satisfy the the algebraic equation. We have chosen the initial values of $\vec{\psi}$ and $\vec{\varphi}$, such that they satisfy the algebraic equation.

5.4.2 Chorin's Projection technique

In order to solve this coupled equation, a projection method similar to Chorin's projection method [8] has been used. Chorin's Projection scheme was originally used to solve the incompressible unsteady state Navier Stokes equations. The key advantage of the projection method is that the computation of the dependent variables are decoupled (Pressure and velocity in the case of Navier Stokes equations). Typically the algorithm consists of two stage fractional step schene, a method which uses multiple calculation steps for each numerical time-step.

The steps are split as following:

- Ignore the ψ term from the lumped equation, and solve the system $\frac{d\vec{q}}{dt} = -L\vec{q}$ with the Implicit Trapezoidal method. Denote the solution after one time step as $\vec{q}*$
- Compute $\vec{\psi}_{n+1}$ from $S_{\psi\varphi}\vec{\varphi} + S_{\psi\psi}\vec{\psi} = 0$ by solving the system of equations where $\vec{\varphi}$ is derived from $q*$.
- Compute $\vec{q}_{n+1}$ as $\vec{q}_{n+1} = \vec{q}* + \Delta t C\vec{\psi}_{n+1}$

In Figure 5.20, the solution obtained from Chorin's projection scheme for different time steps is compared with the Benchmark solution.

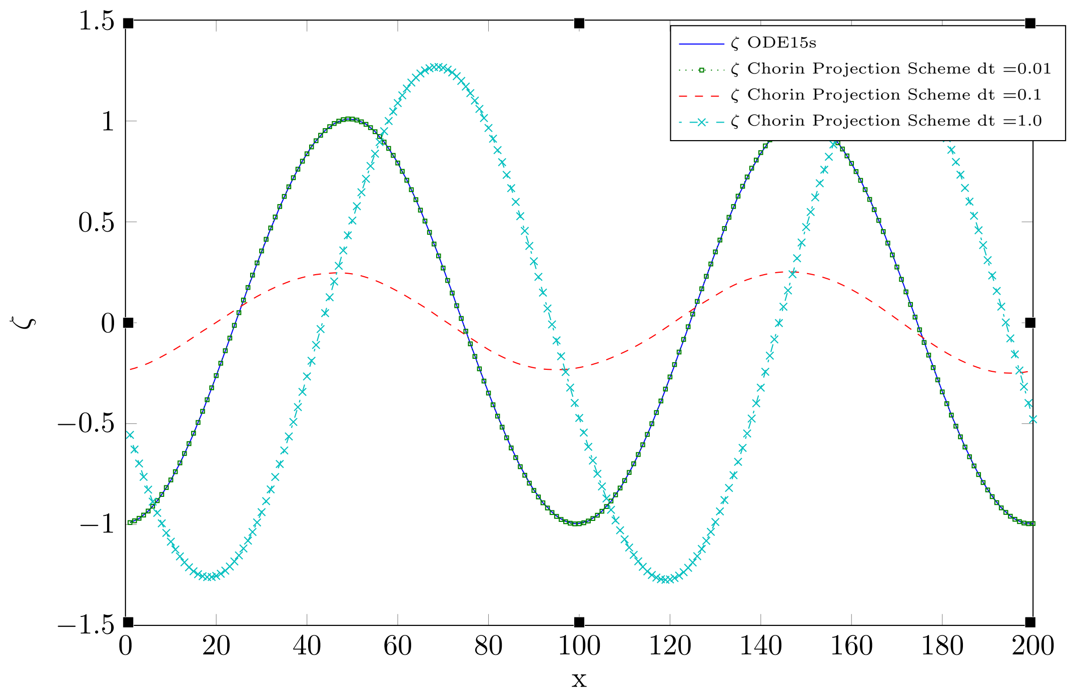


Figure 5.20: ζ variation with length at different times using ODE15s using Chorin's Projection

As the time step is reduced, the solution is more close to the Benchmark solution. As the time step is increased, a larger phase error can be observed. High error norms for the solution at the larger time steps can be attributed to this phase error.

5.5 Stability Analysis for Two Dimensional Case

In order to analyze the stability of the complete system, it is required to study the two dimensional case which results in a Pentadiagonal matrix instead of a Tridiagonal matrix.

Following system of equations are considered as the test case:

$$\frac{\partial \zeta}{\partial t} + \mathbf{U}_x \frac{\partial \zeta}{\partial x} + \mathbf{U}_y \frac{\partial \zeta}{\partial y} + h(\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2}) - h\mathcal{D}(\frac{\partial^2 \psi}{\partial x^2} - \frac{\partial^2 \psi}{\partial y^2}) = 0, \tag{5.11a}$$

$$\frac{\partial \varphi}{\partial t} + \mathbf{U}_x \frac{\partial \varphi}{\partial x} + \mathbf{U}_y \frac{\partial \varphi}{\partial y} + g\zeta = 0, \tag{5.11b}$$

$$\mathcal{M}\psi + hD(\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2}) - \mathcal{N}(\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2}) = 0, \tag{5.11c}$$

Finite volume method based on Central discretization has been used for the spatial discretization of the system of equations. A rectangular grid has been assumed with the possibility of different grid sizes for the X and Y direction. The discretization matrices are Pentadiagonal matrices represented by 5-point stencil as described in Chapter 2 in detail.

It is important to consider the boundary conditions while constructing the Pentadiagonal matrices. In our case, Periodic boundary conditions in both X and Y directions have been used.

The model parameters are given below for the Two dimensional case:

- L_x =100 m
- L_y =100 m
- N_x = 50 Number of grid points
- N_y = 50 Number of grid points
- $U_x = U_y$ = 1.0 m/s
- h = 50 m
- time-max = 50 sec
- Initial Condition: $\zeta^0(x, y) = \cos(\frac{4\pi x}{L_x})$
- $\varphi^0(0, x, y) = 1$
- $\psi^0(x, y) = 0$

The equations in lumped form for the benchmark solution and the Chorin's projection remains the same. Instead of the tridiagonal matrices, we have pentadiagonal block matrices. The grid is ordered lexicographically in Column major format (Similar to Fortran ordering).

Stability, accuracy and the Symplectic nature of the solution from Chorin's projection method is compared with the Benchmark solution. The Implicit Trapezoidal method has been used as the substep of the Chorin's projection method.

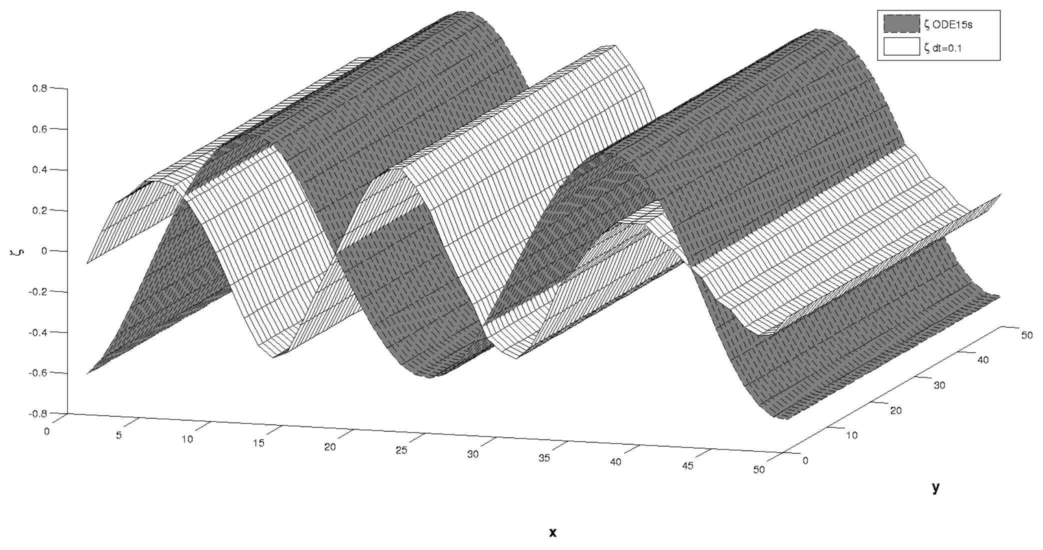


Figure 5.21: ζ variation using ODE15s vs. Chorin's Projection for $\Delta t = 0.01s$ and finer grid

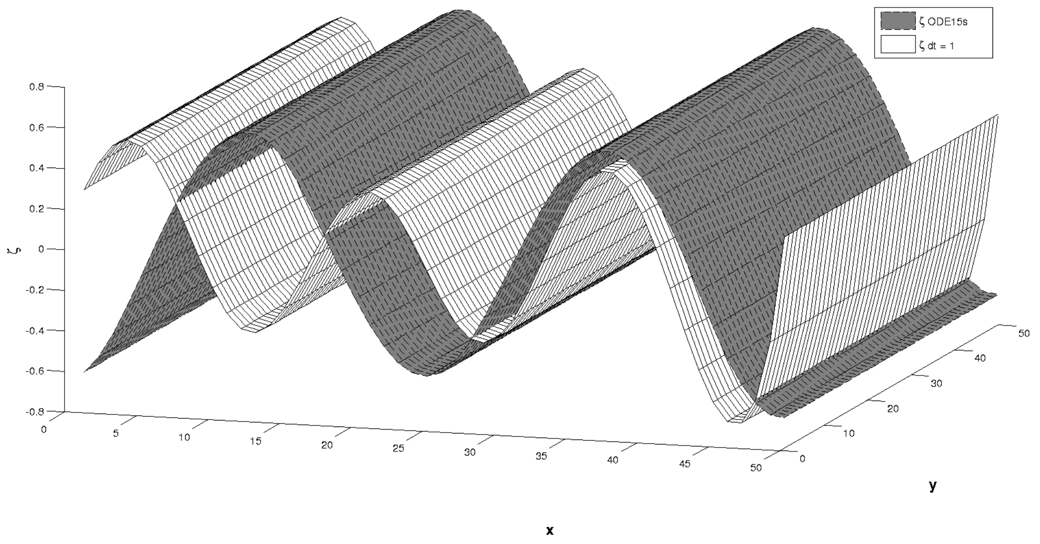


Figure 5.22: ζ variation using ODE15s vs. Chorin's Projection for $\Delta t = 1s$ and finer grid

Phase error is observed at larger time steps, though the stability fo the system is not impacted. This gives us confidence in proceeding with Chorin's Projection scheme for the real case of the interactive waves.

Chapter 6

Iterative Solvers

6.1 Background and Previous Work

In [3] and [1], methods are given to solve the system of equations represented by Equation (2.13b) also given below:

$$S_{\psi\psi}\vec{\psi} = -S_{\psi\varphi}\vec{\varphi} = \mathbf{b}$$

In this section, the solver and the properties of the matrix $S_{\psi\psi}$ will be presented.

6.1.1 Properties of the Discretization Matrix $S_{\psi\psi}$

Numerical discretization of Equation (2.3c) gives Equation (2.13b). $S_{\psi\psi}$ is given by the five point stencil given in 2.12. When using lexicographical ordering of the grid points, this leads to a pentadiagonal matrix. Since the model parameters (functionals $\mathcal{D}, \mathcal{M}$ and $\mathcal{N}$) are positive [3], the center term in the stencil becomes positive and the outer elements are negative. The center term is equal to the sum of the absolute value of the outer terms plus one additional term which is also positive as $h > 0$ and $\mathcal{M}_{0C} > 0$. For a row of the matrix formed by the five point stencil, the diagonal entry is given by the central point of the five point stencil, and the off-diagonal entries are given by the remaining four entries of the stencil. As the diagonal entry is greater than the sum of absolute value of other entries in a row of the matrix, the matrix is strictly diagonally dominant.

To verify the symmetry of the matrix, one has to compare the outer diagonals with each other. For example the contribution of the west term and the east term should be the same. Thus it is required that $\left(\frac{\Delta y}{\Delta x}\bar{\mathcal{N}_W}\right)_{i,j} = \left(\frac{\Delta y}{\Delta x}\bar{\mathcal{N}_E}\right)_{i-i,j}$. In the case of non-uniform mesh, $(\Delta x)_{i,j}$ will be different from $(\Delta x)_{i-1,j}$, and a different discretization scheme will be required to ensure symmetry. For the uniform mesh, $(\Delta x)_{i,j} = (\Delta x)_{i-1,j}$. With the overbar notation, then the requirement can be rewritten as $(\mathcal{N}_W + \mathcal{N}_C)_{i,j} = (\mathcal{N}_C + \mathcal{N}_E)_{i-1,j} \Leftrightarrow (\mathcal{N}_{i-1,j} + \mathcal{N}_{i,j}) = (\mathcal{N}_{i-1,j} + \mathcal{N}_{i,j})$, which is clearly satisfied. Applying the same reasoning to the north and the south neighbors, we conclude that the matrix is symmetrical. A symmetrical diagonally dominant matrix is positive definite and special methods like Conjugate Gradient can be used to solve Equation (2.13b).

6.1.2 Preconditioned Conjugate Gradient Method

The Conjugate Gradient(CG) method is an element of the class of Krylov subspace methods. The Krylov subspace is created by repeatedly applying a matrix to a vector. These are iterative methods, i.e., for a starting vector x^0, the method generates a series of vectors $x^1, x^2, \cdots$ converging to $x = S^{-1}\mathbf{b}$.

A detailed description and derivation of the CG method is provided in [1], [3]. Here we will describe the convergence characteristics and the requirements for the method, along with the algorithm pseudo-code.

The CG method can only be applied to the matrices which are symmetric and positive definite. Convergence of the CG Method for solving the linear system of equations depends on the wavelength spectrum of the matrix S. The improvement in the error is bounded by the Condition Number of the matrix which is given as $k(S) = \frac{\lambda_{max}}{\lambda_{min}}$, where λ_{max} and λ_{min} are the maximum and minimum eigenvalues of the matrix S. Higher the condition number, slower the convergence of the method. It can be derived [3] that for the given system represented by the stencil given in 2.12, the number of iterations required for the convergence of the CG method is inversely proportional to the mesh size.

Preconditioner

If the condition number of the matrix S, $(k(S))$ is large, preconditioning is used to replace the original system $S\psi - \mathbf{b} = 0$ by $\mathbf{M}^{-1}(S\psi - \mathbf{b} = 0)$ so that $(k(M^{-1}S))$ gets smaller than $(k(S))$. In most cases, preconditioning is necessary to ensure fast convergence of the CG method. Algorithm 6.1 provides the pseudo-code for the PCG method.

The choice of the preconditioner is important to achieve a fast solver. Using a preconditioner should ultimately lead to a reduction in work coming from a reduction in required number of iterations for CG to converge. This implies that solving the system $Mz = r$ should be relatively cheap to solve in the case of PCG. Martijn [1] worked on two preconditioners and translated them in an effective CUDA solver. We will describe the RRB (Repeated Red Black) preconditioner in brief here. The other preconditioner (Incomplete Poisson) is specific to the SPD (Symmetric Positive Definite) matrices, and may not be applicable for non-symmetric matrices (for example the matrix given by the stencil in Equation (2.6)).

6.1.3 Repeated Red Black (RRB) ordering as preconditioner

The RRB method uses a renumbering of the grid points according to a red-black ordering. Red-black ordering is the classic way of parallelizing the Gauss Seidel iterative solver by removing the dependencies between the adjacent cells.

What the RRB- method basically does is making the following decomposition:

$$A = LDL^T + R \tag{6.1}$$

where L is a lower triangular matrix, D a block diagonal matrix and R a matrix that contains the adjustments made during the lumping procedure.

In a two dimensional grid, the red nodes are given by the points $x_{i,j}, y_{i,j}$ with $i + j$ as even, and the black nodes with $i + j$ as odd. First the black nodes are numbered in lexicographical ordering and then the red points. For the RRB ordering, the red nodes are again split up into red and black nodes, and we repeat the procedure. When the black node elimination is repeated k times, the method is named RRB-k method. With an elimination process on the finest level,

Algorithm 6.1 Preconditioned CG Algorithm to solve $Sx = \mathbf{b}$

Input x (Start Vector), r (Right Hand Side) , S (Matrix),
M (Preconditioner), ϵ (Tolerance)

```
  $r = b - Sx$
  $solve : Mz = r$
  $\rho_{new} = r^T z$
  $i = 0$
  **while** $\rho_{new} > \epsilon^2 ||b||$ **do**
    $i = i + 1$
    **if** $i = 1$ **then**
      $p = z$
    **else**
      $\beta = \frac{\rho_{new}}{\rho_{old}}$
      $p = z + \beta p$
    **end if**
    $q = Ap$
    $\sigma = p^T q$
    $\alpha = \frac{\rho_{new}}{\sigma}$
    $x = x + \alpha p$
    $r = r - \alpha q$
    $solve : Mz = r$
    $\rho_{old} = \rho_{new}$
    $\rho_{new} = r^T z$
  **end while**
```

a stencil is obtained on the next coarser level, which is then simplified by lumping some outer elements (Converting the 9 point stencil on the coarser level back to 5 point stencil). The process of elimination and lumping is then repeatedly applied on the resulting stencil, until the grid is coarse enough to use a direct solver on it.

6.1.4 Implementation of RRB preconditioner

Martijn [1] has taken great efforts in describing the implementation of the RRB solver both in C++ programming, and CUDA kernels. Instead of digressing into minor details, the main points which will later impact the implicit time integration are described in brief here.

6.1.5 Effect of RRB ordering on sparsity pattern

The matrix S in the system $S\psi = b$ is given by a five point stencil, see Section 2.2. This results in a matrix whose structure is pentadiagonal. The sparsity pattern changes when RRB ordering is applied.

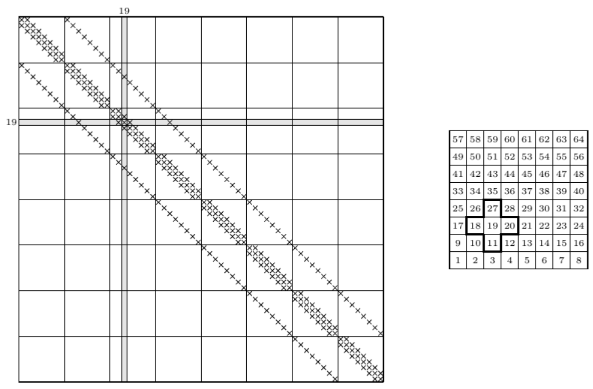


Figure 6.1: Sparsity pattern of $S \in \mathcal{R}^{64x64}$ before RRB ordering [1]

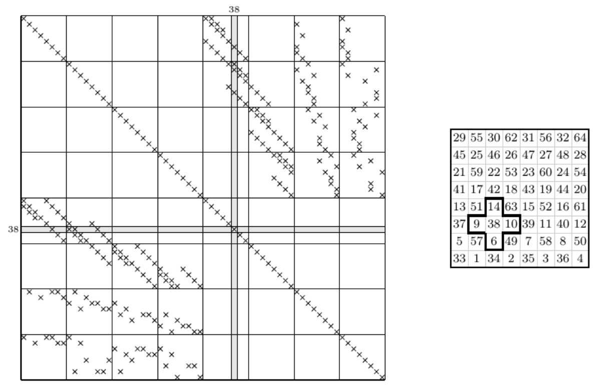


Figure 6.2: Sparsity pattern of $S \in \mathcal{R}^{64x64}$ after RRB ordering [1]

6.1.6 RRB-k method

The maximum number of levels of the RRB ordering could be computed given the dimensions of the grid in x and y direction. Though, it is not required to go upto the coarsest level. It is possible to stop at any level of coarsening. Such level is named k, and hence the RRB-k method. On level k, the nodes are numbered naturally (not in red-black fashion). From these nodes a matrix E is formed, which is a symmetric pentadiagonal matrix. The idea of stopping earlier is that we want to solve the remaining nodes accurately and exactly. In order to keep the computational cost lower, it is important to keep the size of the matrix E small.

On level k, we have to solve the system of like $Ex = b$, where x is formed by level k nodes of z and b by the level k nodes of r in the problem $Mz = r$. It is possible to use direct solvers (Complete Cholesky factorization) or Gaussian elimination for such small problem.

6.1.7 The Schur complement and the 9-point Stencil

With the basic red-black numbering, the matrix S can be written in a block matrix format. The system $S\psi = b$ can then be written in the form:

$$\begin{bmatrix} D_b & S_{br} \\ S_{rb} & D_r \end{bmatrix} \begin{bmatrix} \psi_b \\ \psi_r \end{bmatrix} = \begin{bmatrix} b_b \\ b_r \end{bmatrix}$$

Subscript b and r denotes the black and red nodes respectively. D_r and D_b are the diagonal matrices, and for a symmetric matrix S, $S_{rb} = S_{br}^T$ (matrices with 4 off-diagonals).

Next, the black nodes are eliminated by applying Gaussian elimination. This yields:

$$\begin{bmatrix} D_b & S_{br} \\ 0 & D_r - S_{rb}D_b^{-1}S_{br} \end{bmatrix} \begin{bmatrix} \psi_b \\ \psi_r \end{bmatrix} = \begin{bmatrix} b_b \\ b_r - S_{rb}D_b^{-1}S_{br} \end{bmatrix}$$

The matrix $S_1 := D_r - S_{rb}D_b^{-1}S_{br}$ is called the *1st Schur complement* and is given by a 9-point stencil, the vector $b_1 := b_r - S_{rb}D_b^{-1}S_{br}$ is the corresponding right-hand side. Using the solution of red nodes, it is thus possible to compute the solution of black nodes. The catch here is that instead of the five point stencil of S, more expensive 9-point stencil of S_1 has to be solved.

6.1.8 Constructing the preconditioning matrix M

The preconditioning matrix $M = LDL^T$ is constructed in following four steps:

- Elimination of black nodes: 9-point stencil is generated from the 5-point stencil (creation of the Schur complement). In the case of C++ code, all the five diagonals are stored separately, whereas in CUDA code only three out of five diagonals are stored. West and South stencil are used to access the East and North stencil respectively (utilizing the symmetry of the matrix).
- Lumping from the 9-point stencil to the 5-point stencil for the remaining red nodes. This ensures removing stencil dependencies by adding the respective coefficients to other coefficients.
- Elimination of the first level red nodes using the lumped 5-point stencil (creating the second level red nodes with 9-point stencil).

- Lumping on the second level. The resulting matrix on the coarse grid is pentadiagonal and has the same properties as the original matrix S.

The above procedure is only one RRB iteration. It consists of an elimination $S = L_1 S_1 L_1^T$, lumping $S_1 = \underline{S_1} + R_1$ and again an elimination $\underline{S_1} = L_2 S_2 L_2^T$ and lumping $S_2 = \underline{S_2} + R_2$. Combined we have,

$$\begin{aligned} S &= L_1 L_2 \underline{S_2} L_2^T L_1^T + L_1 L_2 R_2 L_2^T L_1^T + L_1 R_1 L_1^T \\ &= LDL^T + R. \end{aligned}$$

6.1.9 Solving $Mz = r$

As given in Algorithm 6.1, in each iteration the preconditioning step $Mz = r \implies LDL^T z = r$ has to be solved for z. This is done in three steps:

- $La = r$ is solved using forward-substitution. This is done level-wise going from finer grid to the coarser grid.
- $b = D^{-1}a$. As D is block diagonal matrix, its inverse is easy to compute.
- $L^T z = b$ is solved using back-substitution. This is done by going from coarser grids to fine grids level-wise.

6.1.10 Efficient CUDA and C++ Code Implementation

In order to boost performance, Martijn has implemented efficient kernels for the following:

- Matrix Vector product
- Dot product : Performing mass reduction on GPU and using Kahan summation algorithm on CPU. Kahan summation algorithm greatly reduces error in floating point additions (It makes possible to sum n numbers with an error that only depends on the floating-point precisions)
- Vector Updates

These optimized codes could be reused even if the PCG algorithm is changed with minor modifications.

6.2 Solvers for Non-Symmetric Matrices

As we have seen, most of the implicit time integration methods described in Chapter 4 (except MR-PC) will require solving a set of equations where the matrices have stencils as given in Section 2.2. The matrices involved are not symmetric as compared to the matrix given by Stencil $S_{\psi\psi}$. For this reason, the PCG method as described in Chapter 3 cannot be applied to solve the linear system of equations arising from the implicit time integration.

The objective is therefore to determine the solution vector for a large, sparse and linear system of equations which is not symmetric. An overview of these methods can be found in [6], which forms the basis for this Chapter.

As mentioned before in Chapter 3, the idea of some iterative methods is to project an m-dimensional problem into a lower-dimensional Krylov subspace. Given a matrix A of size $m \times m$ and a vector b of size $m \times 1$, the associated Krylov sequence is the set of vectors $b, Ab, A^2b, A^3b, \cdots$, which can be computed by matrix-vector multiplications in the form $b, Ab, A(Ab), A(A(Ab)))$ The corresponding Krylov subspaces are the spaces spanned by successively larger group of these vectors.

6.2.1 From Symmetric to Non-Symmetric Matrices

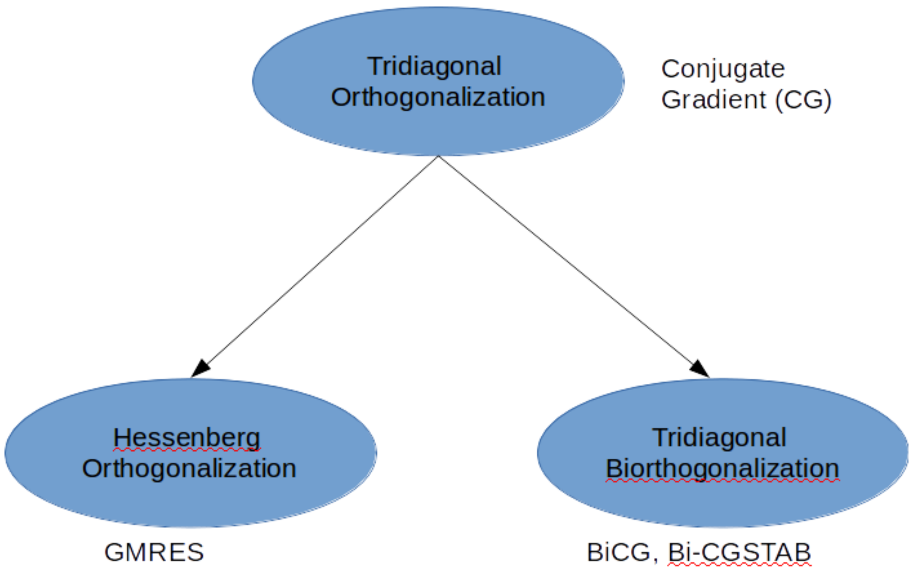

Tridiagonal Orthogonalization
Conjugate Gradient (CG)
Hessenberg Orthogonalization
Tridiagonal Biorthogonalization
GMRES
BiCG, Bi-CGSTAB

Figure 6.3: Classification of Krylov Subspace iterations

Figure 5.1 shows the classification of Krylov Subspace methods as we move from symmetric matrices to non-symmetric matrices. The Conjugate Gradient (CG) method results in the tridiagonal orthogonalization of the original matrix, which can be described as $A = QTQ^T$, where Q is the Unitary matrix and T is a tridiagonal matrix. When A is non-symmetric, this result cannot be obtained from a CG iteration. Two approaches can be followed :

- Use of the so-called Arnoldi Iteration, a process of Hessenberg orthogonalization. This results in $A = QHQ^T$ where Q is a Unitary matrix and H is an upper Hessenberg matrix. (An upper Hessenberg matrix has zero entries below the first sub-diagonal).

- Bi-orthogonalization methods are based on the opposite choice. If we insist on obtaining a tridiagonal result, then we have to give up the unitary transformations, which gives us tridiagonal biorthogonalization : $A = VTV^{-1}$, where V is non-singular but generally not Unitary. The term 'biorthogonal' refers to the fact that though all the columns of V are not orthogonal to each other, they are orthogonal to the columns of $(V^{-1})^T = (V^T)^{-1}$.

  Let V be a non-singular matrix such that $A = VTV^{-1}$ with T tridiagonal and define $W = (V^T)^{-1}$. Let v_j and w_j denote the j^th columns of V and W respectively. These vectors are biorthogonal in the sense that $w_i^T v_j = \delta_{ij}$ where δ_{ij} is the Kronecker delta function. For each n with $1 \leq n \leq m$, define the $m \times n$ matrices such that:

$$V_n = \left[v_1 | v_2 | \cdots | v_n\right], \quad W_n = \left[w_1 | w_2 | \cdots | w_n\right] \tag{6.2}$$

In matrix form, the biorthogonality can be written as $W_n^T V_n = V_n^T W_n = I_n$, where I_n is the identity matrix of dimension n. The iterations in biorthogonalization methods can be summarized as:

$$AV_n = V_{n+1}\tilde{T}_n \tag{6.3a}$$

$$A^T W_n = W_{n+1}\tilde{S}_n \tag{6.3b}$$

$$T_n = S_n^T = W_n^T A V_n \tag{6.3c}$$

Here V_n and W_n have dimensions $m \times n$, $\tilde{T}_{n+1}$ and $\tilde{S}_{n+1}$ are tridiagonal matrices with dimensions $(n+1) \times n$, and $T_n = S_n^T$ is the $n \times n$ matrix obtained by deleting the last row of $\tilde{T}_{n+1}$. BiConjugate gradient algorithms are described in Section 5.3.

6.2.2 Arnoldi Iteration and GMRES

Arnoldi iteration can be understood as the analogue of a Gram-Schmidt type iteration for similarity transformations to upper Hessenberg form. It has the advantage that it can be stopped part-way, leaving one with a partial reduction to Hessenberg form that is exploited when dimensions upto n (dimension of the Krylov subspace) are considered. A simple algorithm of Arnoldi iteration is given below:

Algorithm 6.2 Arnoldi Iteration

```
$b$ = arbitrary, $q_1 = b/||b||$
for $n = 1, 2, 3, \cdots$ do
    $v = Aq_n$
    for $j = 1$ do
        $h_{jn} = q_j^* v$
        $v = v - h_{jn} q_j$
    end for
    $h_{n+1,n} = ||v||$
    $q_{n+1} = v/h_{n+1,n}$
end for
```

The above algorithm can be condensed in the following form:

- The matrices $Q_n = \left[q_1 | q_2 | \cdots | q_n\right]$ generated by the Arnoldi iteration are reduced QR factors of the Krylov matrix:

$$K_n = Q_n R_n \tag{6.4}$$

  where K_n is the $m \times n$ Krylov matrix.

- The Hessenberg matrices H_n are the corresponding projections : $H_n = Q_n^T A Q_n$
- The successive iterates are related by the formula: $AQ_n = Q_{n+1} H_n^{'}$, where $H_n^{'}$ is the $(n+1) \times n$ upper-left section of H.

The idea of GMRES is straightforward. At step n, the exact solution ($x^o = A^{-1}b$) is approximated by the vector $x_n \in K_n$ that minimizes the norm of the residual $r_n = b - Ax_n$, hence the name : Generalized Minimal Residuals (GMRES). It was proposed in 1986 and is applicable to a system of equations when the matrix is a general (Non-singular) square matrix. Arnoldi iteration is used to construct a sequence of Krylov matrix Q_n whose columns $q_1, q_2, \cdots$ successively span the Krylov subspace K_n. Thus we can write $x_n = Q_n y$, where y represents the vector such that

$$||AQ_n y - b|| = \text{minimum} \tag{6.5}$$

Using the similarity transformation, this equation can be written as:

$$||Q_{n+1}H_n^{'}y - b|| = \text{minimum} \tag{6.6}$$

Multiplication by a Unitary matrix does not change the 2-norm, thus we can rewrite above equation as:

$$||Q_{n+1}^{*}Q_{n+1}H_n^{'}y - Q_{n+1}^{*}b|| = \text{minimum}||H_n^{'}y - Q_{n+1}^{*}b|| = \text{minimum} \tag{6.7}$$

Finally, by construction of the Krylov matrices Q_n, we have: $Q_{n+1}^{*}b = ||b||e_1$ where $e_1 = (1, 0, 0, \cdots)^T$. Thus we obtain:

$$||H_n^{'}y - ||b||e_1|| = \text{minimum} \tag{6.8}$$

The GMRES algorithm (unPreconditioned) can be written as :

Algorithm 6.3 GMRES

```
$q_1 = b/||b||$
**for** $n = 1, 2, 3, \cdots$ **do**
    <step n of Arnoldi iteration, Algorithm 6.2>
    Find $y$ to minimize $||H_n^{'}y - ||b||e_1||(= ||r_n||)$
    $x_n = Q_n y$
**end for**
```

In order to find y, a QR factorization can be used which requires $O(n^2)$ flops, because of the Hessenberg structure of the matrix $H^{'}$. Here n is the dimension of the Krylov subspace. Also it is possible to get the QR factorization of $H_n^{'}$ from that of $H_{n-1}^{'}$ by using Given's Rotation.

One of the disadvantages of the GMRES method is the storage requirements. As it requires storing the whole sequence of the Krylov subspace, a large amount of storage is required as compares to the Conjugate Gradient method. For this reason, restarted versions of this method are used, where computational and storage costs are limited by specifying a fixed number of vectors to be generated.

6.2.3 BiConjugate Gradient methods

The BiConjugate Gradient method (BiCG) is the other extension for non-symmetric matrices. As we saw in the previous section, the principle of GMRES is to pick vector x_n such that the residual corresponding to x_n is minimized. The principle of BiCG algorithm is to pick x_n in the same sub-space, i.e. $x_n \in K_n$, but to enforce that the residual is orthogonal to $w_1, A * w_1, \cdots, (A*)^{n-1}w_1$, where $w_1 \in R^m$ is an arbitrary vector satisfying the dot product, $w_1^* v_1 = 1$. Its advantage is that it can be implemented with three-term recurrences rather than the (n+1) - term recurrences of GMRES (Difference arising from Hessenberg form of matrix vs. tridiagonal form).

There are two major problems with the BiCG method :

- Convergence is slower as compared to GMRES and often erratic. Also, it may have the consequence of reducing the ultimately attainable accuracy because of rounding errors.
- It required multiplication with A^T (transpose) as well as A. Computing the transpose brings serialization to the code and thus is not preferred.

To address this problem, other variants of BiCG method were developed. One of them is stabilized BiCG method (Bi-CGSTAB). As for the Conjugate Gradient method, any other Krylov subspace method needs a good preconditioner to ensure fast and robust convergence. The algorithm for the preconditioned Bi-CGSTAB method is given below. One of the future task is to adjust this algorithm so as to use the same building blocks as developed by Martijn in [1] for the RRB-k method.

Algorithm 6.4 Bi-CGSTAB

```
Solve the system of equation given by $Ax = b$ by Bi-CGSTAB
Compute $r^0 = b - Ax^0$ for some initial guess $x^0$.
Choose $\tilde{r}$ (for example $\tilde{r} = r^0$)
for $i = 1, 2, 3, \cdots$ do
  $\rho_{i-1} = \tilde{r}^T r^{i-1}$
  if $\rho_{i-1} = 0$ then
    Method Fails
  end if
  if $i = 1$ then
    $p^i = r^{i-1}$
  else
    $\beta_{i-1} = \frac{\rho_{i-1}}{\rho_{i-2}} \frac{\alpha_{i-1}}{\omega_{i-1}}$
    $p_i = r^{i-1} + \beta_{i-1}(p^{i-1} - \omega_{i-1} v^{i-1}$
  end if
  Solve $M\tilde{p} = p^i$
  $v^i = A\tilde{p}$
  $\alpha_i = \frac{\rho^{i-1}}{\tilde{r}^T} v^i$
  $s = r^{i-1} - \alpha_i v^i$
  Check norm of s; if small enough, set $x^i = x^{i-1} + \alpha_i \tilde{p}$
  Solve $M\tilde{s} = s$
  $t = A\tilde{s}$
  $\omega_i = \frac{t^T s}{t^T t}$
  $x^i = x^{i-1} + \alpha_i \tilde{p} + \omega_i \tilde{s}$
  $r^i = s - \omega_i t$
  Check Convergence, continue if necessary
  For continuation it is required that $\omega_i \neq 0$
end for
```

6.3 Combined Solver - GMRES/ Bi-CGStab

We formulate the strategy for the implementation of the Solver for the Test Case discussed in the Chapter 4, which can then be benchmarked with the solution from Matlab. We will implement Chorin's Projection scheme in C++ for the Two-Dimensional Case. As we intend to use the same framework as used in Martijn's Code, we should be able to plugin Martijn's RRB solver into our code to solve the system of equation represented by the symmetric matrix without any modification. Step 1 of the Chorin's projection scheme requires a different strategy than the current implementation of the preconditioned Conjuate Gradient method. This is discussed in detail in following sub sections.

For the first step of the Chorin's Projection Scheme, the system of equation to be solved is given by :

$$\dot{\mathbf{q}} + L\mathbf{q} = 0, \tag{6.9}$$

with $\mathbf{q} = \begin{bmatrix} \vec{\zeta} \\ \vec{\varphi} \end{bmatrix}$ and $\dot{\mathbf{q}}$ its time derivative. The matrix $L = \begin{bmatrix} S_{\zeta\zeta} & S_{\zeta\varphi} \\ S_{\varphi\zeta} & S_{\varphi\varphi} \end{bmatrix}$ is the spatial discretization matrix as discussed in Chapter 2.

The Implicit Trapezoidal method is used to integrate the system of ODE's.

$$(\frac{I}{\Delta t} + \beta L)q^{n+1} = (\frac{I}{\Delta t} - (1-\beta)L)q^n \tag{6.10}$$

This can also be written as :

$$\begin{bmatrix} \frac{I}{\Delta t} + \beta S_{\zeta\zeta} & \beta S_{\zeta\varphi} \\ \beta S_{\varphi\zeta} & \frac{I}{\Delta t} + \beta S_{\varphi\varphi} \end{bmatrix} \begin{bmatrix} \vec{\zeta} \\ \vec{\varphi} \end{bmatrix}^{n+1} = \begin{bmatrix} \frac{I}{\Delta t} - (1-\beta) S_{\zeta\zeta} & (1-\beta) S_{\zeta\varphi} \\ (1-\beta) S_{\varphi\zeta} & \frac{I}{\Delta t} + (1-\beta) S_{\varphi\varphi} \end{bmatrix} \begin{bmatrix} \vec{\zeta} \\ \vec{\varphi} \end{bmatrix}^{n} \tag{6.11}$$

The system matrix given by the matrix on the left hand side can be seen as combination of 4 block matrices. The diagonal blocks arise from the spatial discretization of the hyperbolic system of equations, and thus are Skew-Symmetric in nature. The off-diagonal blocks represent the coupling between the variables ζ and φ.

6.3.1 Preconditioners

Each GMRES or Bi-CGSTAB iteration requires the preconditioning step, namely solving a system of equation $Mz = r$, where r is some residual and M is the combined preconditioner. For the diagonal block matrices, it should be possible to construct a preconditioner similar to the RRB preconditioner of the symmetric pentadiagonal matrix. These are represented as P, and let Q represent the off-diagonal blocks in the preconditioner.

In general form, the Combined preconditioner M is given by:

$$M = \begin{bmatrix} P_{\zeta\zeta} & Q_{\zeta\varphi} \\ Q_{\varphi\zeta} & P_{\varphi\varphi} \end{bmatrix} \tag{6.12}$$

Various preconditioners can be constructed based on the structure of Q .

By setting $Q = 0$, we obtain : $M_1 = \begin{bmatrix} P_{\zeta\zeta} & 0 \\ 0 & P_{\varphi\varphi} \end{bmatrix}$. This is a block Jacobi preconditioner for the combined system. Equation $M_1 z = r$ can be written as:

$$\begin{bmatrix} P_{\zeta\zeta} & 0 \\ 0 & P_{\varphi\varphi} \end{bmatrix} \begin{bmatrix} z_\zeta \\ z_\varphi \end{bmatrix} = \begin{bmatrix} r_\zeta \\ r_\varphi \end{bmatrix} \tag{6.13}$$

which can be split into solving $P_{\zeta\zeta} z_\zeta = r_\zeta$ and $P_{\varphi\varphi} z_\varphi = r_\varphi$

Similarly, block Gaussian forms of the preconditioner are given by:

$$M_2 = \begin{bmatrix} P_{\zeta\zeta} & 0 \\ S_{\varphi\zeta} & P_{\varphi\varphi} \end{bmatrix} \tag{6.14}$$

$$M_3 = \begin{bmatrix} P_{\zeta\zeta} & S_{\zeta\varphi} \\ 0 & P_{\varphi\varphi} \end{bmatrix} \tag{6.15}$$

In order to solve the system of equation $M_2 z = r$, we can solve:

$$\begin{bmatrix} P_{\zeta\zeta} & 0 \\ S_{\varphi\zeta} & P_{\varphi\varphi} \end{bmatrix} \begin{bmatrix} z_\zeta \\ z_\varphi \end{bmatrix} = \begin{bmatrix} r_\zeta \\ r_\varphi \end{bmatrix} \tag{6.16}$$

This can be split up into solving $P_{\zeta\zeta} z_\zeta = r_\zeta$ and $P_{\varphi\varphi} z_\varphi = r_\varphi - S_{\varphi\zeta} z_\zeta$

Similarly we can solve $M_3 z = r$.

6.3.2 Preconditioner for Skew-Symmettric Matrix

As discussed above, it is required to formulate preconditioner for the diagonal blocks, which are skew-symmetric in nature. We will again use Repeated Red Black method to construct the preconditioner.

The same strategy as described in Section 6.1.8 will be followed to construct the preconditioner. In the CUDA code, now instead of storing only three diagonals, all five diagonals will be used. In the current implementation by Martijn [1], a clever trick is used which saves memory (Phase 2c of the construction step) by utilizing the symmetry. The matrix S is split into LDL^T and the stencils corresponding to L^T are not stored.

In the case of the matrix being skew symmetric, it can be derived that the preconditioning matrix has the form $P = LDU$, where $U = -L^T$. Hence, we do not need to store the diagonals corresponding to U, and we can use the same preconditioning code in C++ and CUDA as used by Martijn.

Algorithm

In order to construct the preconditioner represented by the Matrix Equation (6.11), the stencils that are part of the four blocks, namely $S_{\zeta\zeta}$, $S_{\varphi\varphi}$, $S_{\zeta\varphi}$ and $S_{\varphi\zeta}$ are copied into the solver class. The impact of boundary conditions are incorporated while formulating these stencils, which will be discussed later in detail.

Copied stencils are modified to represent the left-hand side of the Equation (6.11), and then preconditioning matrices, $P_{\zeta\zeta}$ and $P_{\varphi\varphi}$ are constructed respectively.

The same algorithm is also implemented in Matlab, and the matrices generated are stored in order to analyze their convergence characteristics as discussed below.

6.3.3 Impact of different Preconditioners on Eigenvalue Spectrum

The effectiveness of a preconditioner for the Symmetric matrix can be evaluated from the reduction in the condition number. For the non-symmetric matrices, the eigenvalues can also contain complex imaginary components. Thus, the condition number cannot be used to determine the convergence properties of a given preconditioner.

To understand the convergence property of methods like GMRES and Bi-CGSTAB on a given matrix A, the eigenvalue spectrum is plotted. Let us assume a hypothetical system whose eigenvalue spectrum is given in Figure 6.4

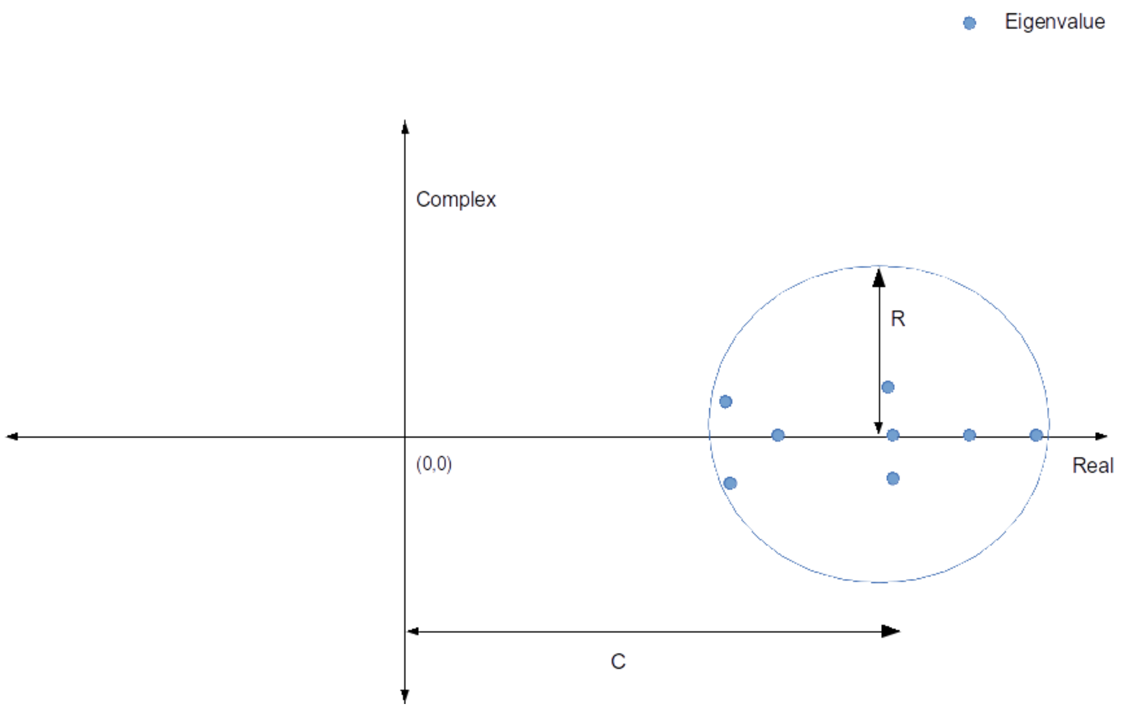


Figure 6.4: Eigenvalue Spectrum of matrix A

A circle is drawn which encapsulates all the eigenvalues. Let the distance of the circle from the origin be C, and radius of the circle be R, as shown in Figure 6.4. The convergence property of the matrix can be related to the ratio $\frac{R}{C}$. Lower the number, less number of GMRES/Bi-CGSTAB iterations will be required. We will now study the eigenvalue spectrum of the coupled system given in Equation (6.11). Let $A = \begin{bmatrix} \frac{I}{\Delta t} + \beta S_{\zeta\zeta} & \beta S_{\zeta\varphi} \\ \beta S_{\varphi\zeta} & \frac{I}{\Delta t} + \beta S_{\varphi\varphi} \end{bmatrix}$.

Then for a sample test case, where $\Delta x = \Delta y = 5$, $\Delta t = 0.1$, $\beta = 0.5$, $h = 50$ and $U_x = U_y = 1$, the spectrum obtained is shown in Figure 6.5.

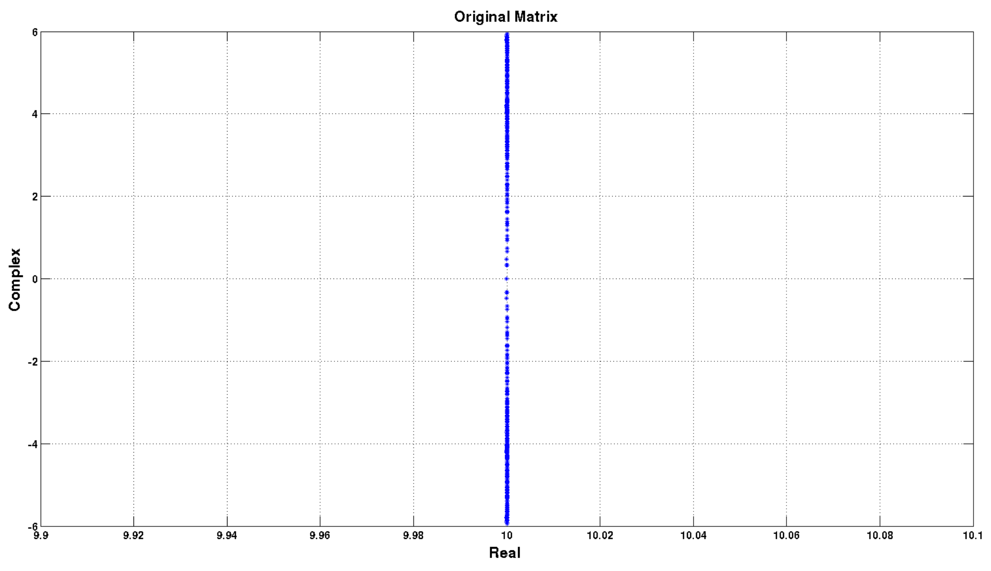


Figure 6.5: Eigenvalue Spectrum of matrix A

In order to analyze the impact of different preconditioners, the eigenvalue spectrum of $M^{-1}A$ is plotted in the following figures. The inverse of the preconditioner and matrix matrix multiplication is carried out in Matlab.

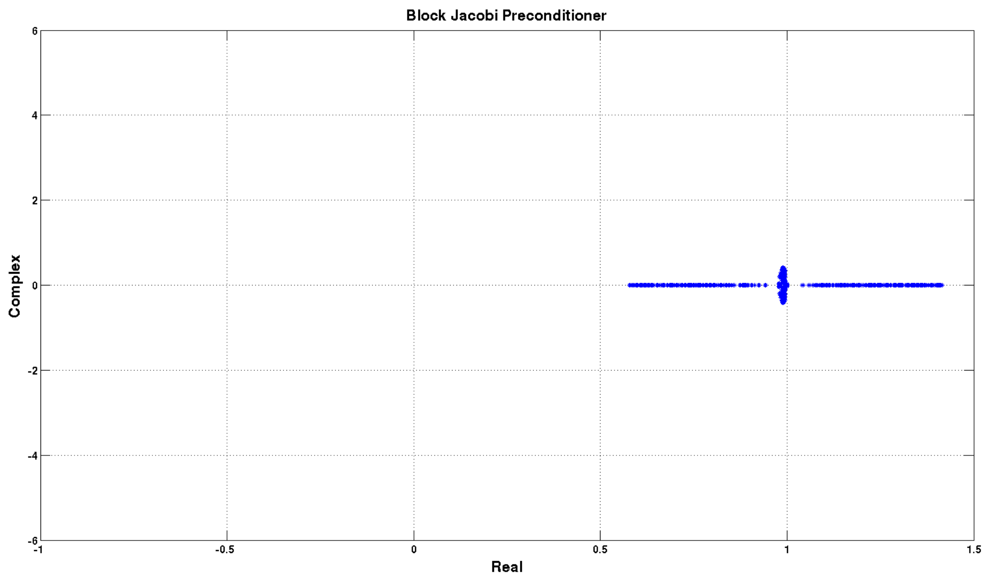


Figure 6.6: Eigenvalue Spectrum of matrix $M_1^{-1}A$

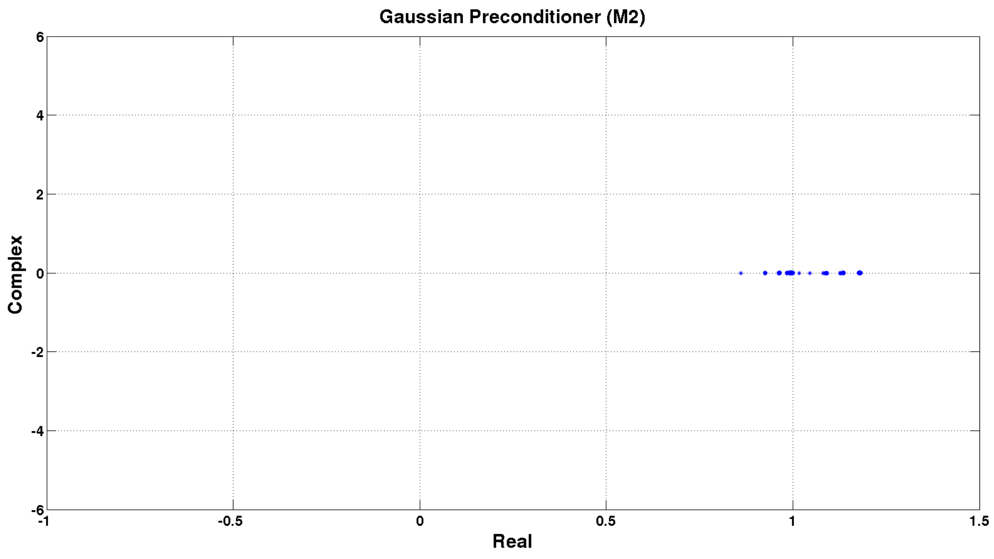


Figure 6.7: Eigenvalue Spectrum of matrix $M_2^{-1}A$

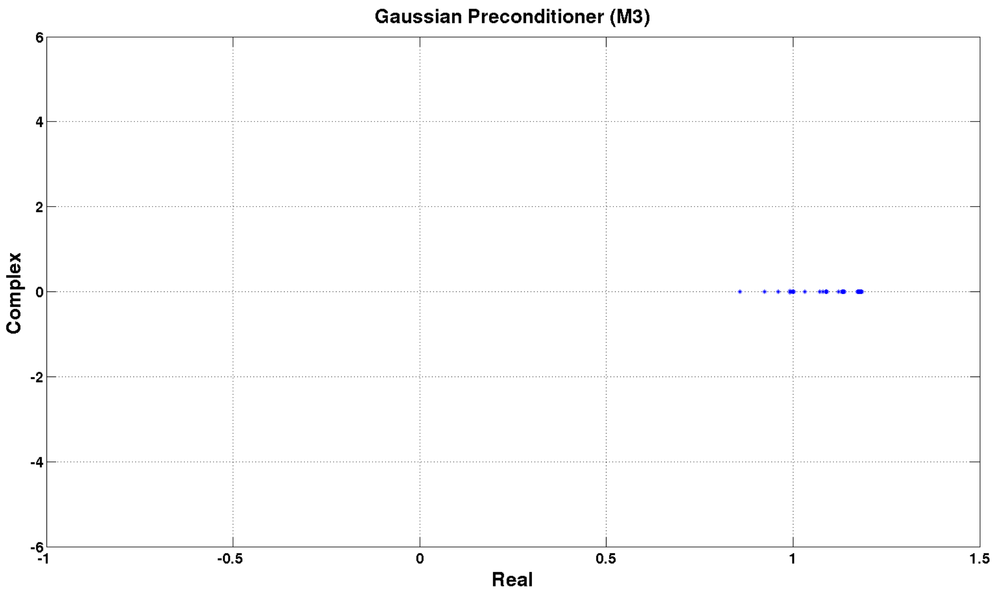


Figure 6.8: Eigenvalue Spectrum of matrix $M_3^{-1}A$

For all the preconditioners, the ratio R/C is computed for two different values of Δt and tabulated below.

Table 6.1: Convergence characteristics with different preconditioners

Preconditioner	$\frac{R}{C}$, $\Delta t = 0.1$	$\frac{R}{C}$, $\Delta t = 1.0$
Without Preconditioner	0.6	6
M_1	0.4	4
M_2	0.18	1
M_3	0.18	1

It can be seen that the Jacobi block preconditioner M_1 does not impact the convergence characteristics, whereas the Gaussian preconditioners M_2 and M_3 results in a reduction of more than 3-4 times. This can be attributed to the coupling between the two governing equations. As discussed before, the maximum eigenvalue of the governing system of equations is given by the suface gravity wave $\sqrt{gh}$. We have previously seen in the Semi-Implicit schemes, that ignoring the coupling between the equations for ζ and ϕ do not give us a stable solution. Similar behaviour is observed when we choose the Jacobi block preconditioner, as we ignore the contribution from the off-diagonal blocks which contain the components representing the coupling and determined by the parameters g and h.

Also, we can oberseve that an increase in the time step results in a larger $\frac{R}{C}$ ratio. As the time step is increased, $\frac{I}{\Delta t}$ reduces and the matrix A loses its diagonal dominance. If we use a large time step, then we should observe in an increase in number of GMRES/Bi-CGSTAB iterations.

Chapter 7

C++ and CUDA Implementation

The solver for the Implicit Trapezoidal method in C++ and CUDA is first developed for the two-dimensional test case. We will employ the Chorin's Projection scheme as the algorithm to march in time. An important consideration while developing the code has been to keep the modularity as it was developed by Martijn [1]. This helps in plugging different solvers with ease into the code. The kernels developed for CUDA computation by Martijn have been modified in order to reflect the requirements of implicit set of equations.

The complete flowchart for the Implicit solver for the simplified test problem is given in Figure 7.4. The idea is to benchmark the solver routines, and then use them in the Interactive wave program. In this chapter, the implementation of the major steps which are also used later in the Interactive code in detail are explained.

7.1 General comments on implementation

Throughout the C++ code, the class Array2D is used which is a very efficient custom-built C++ class exploiting pointers that let us store and manipulate 2D arrays in a comparatively cheap manner. The memory structure in C++ is shown in Figure 7.1. Here COLUMNS = Nx1+3 and ROWS=Nx2+3. This includes the Ghost layers which are used for boundary computations.

In the CUDA environment, the arrays from the host (C++) are copied to the device (GPU) and stored in an embedded grid. When the original grid has size Nx1 (x) by Nx2 (y) on the host, on the device it arrives as a Nx2 (x) by Nx1 (y) array. This mirroring effect is due to the storage pattern difference between C++ and GPU (row-major vs. column-major order respectively). The structure of the grid in CUDA is shown in Figure 7.2. The embedding is carried out by choosing a appropriate BORDER_WIDTH. Martijn[1] has taken BORDER_WIDTH of 16 which makes that data stored according to the 'optimal 16- spacing (= 64 byte)' thus allowing access to the elements in a coalesced fashion.

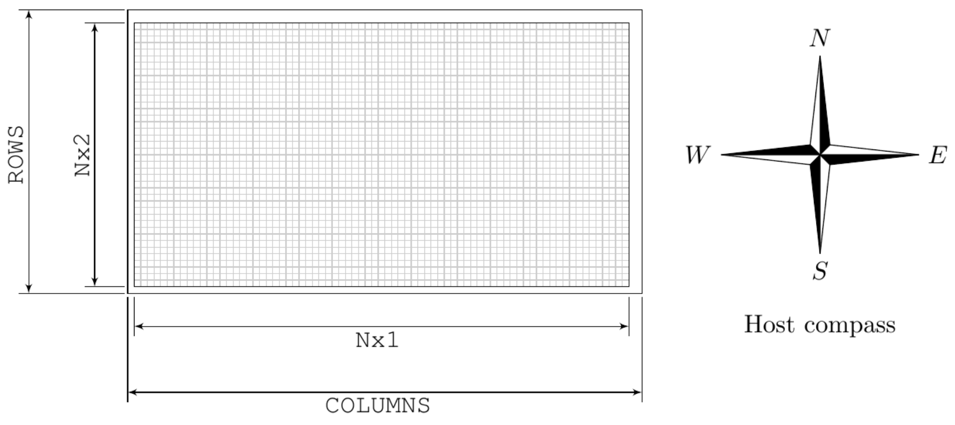


Figure 7.1: Array2D structure in C++ [1]

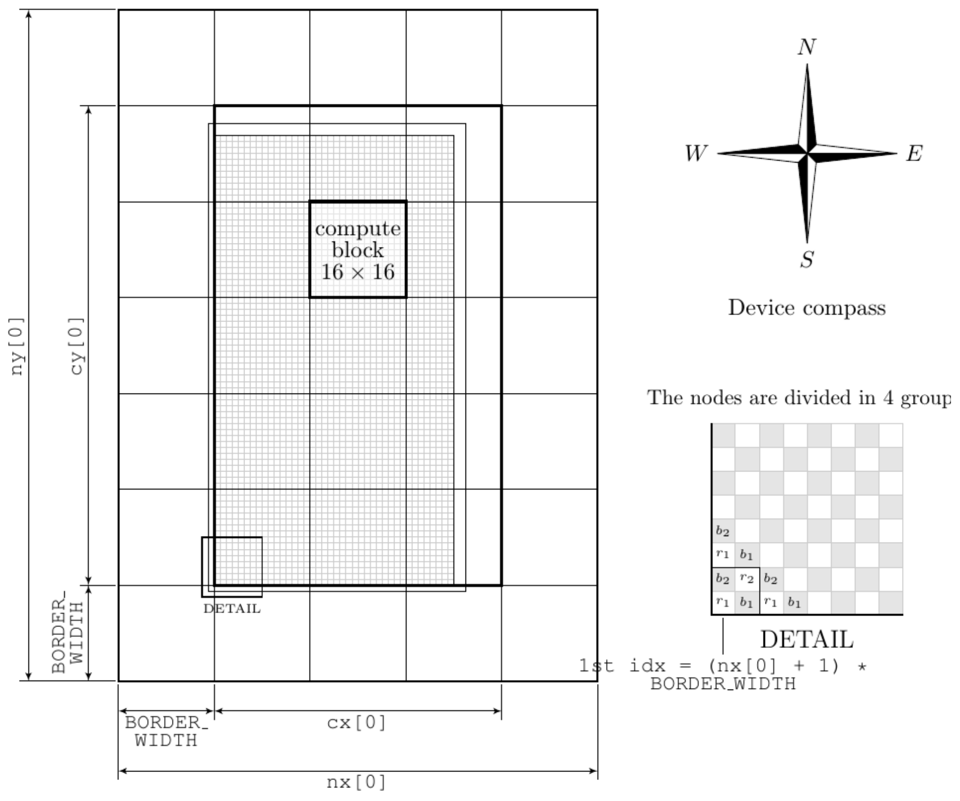


Figure 7.2: Embedded grid structure in CUDA [1]

The new array is nx[0] by ny[0]. Also a so-called compute-block is indicated. A compute-block is the basic identity on which CUDA will do the computations. The compute-block is square given by the dimension DIM_COMPUTE_BLOCK: 32 x 32 elements . The computing area is given by cx[0] by cy[0], which is a multiple of the compute-block.

Since the repeated red black ordering is used in the preconditioning step, the data is restored in a r1/r2/b1/b2 storage format. It is divided into four groups representing the repeated red black ordering: The r1-nodes (even/even red nodes), the b1-nodes (the odd/even black nodes), the r2 nodes (the odd/odd red nodes), and the b2-nodes (the even/odd black nodes). "Even/Odd" here implies that the first coordinate is even and the second coordinate is odd. After the data is copied in the device from the host, it is restored in this r1/r2/b1/b2 storage format. The storage structure is shown in Figure 7.3. More details on the storage format, and how to access the elements by thread indexing can be found in [1].

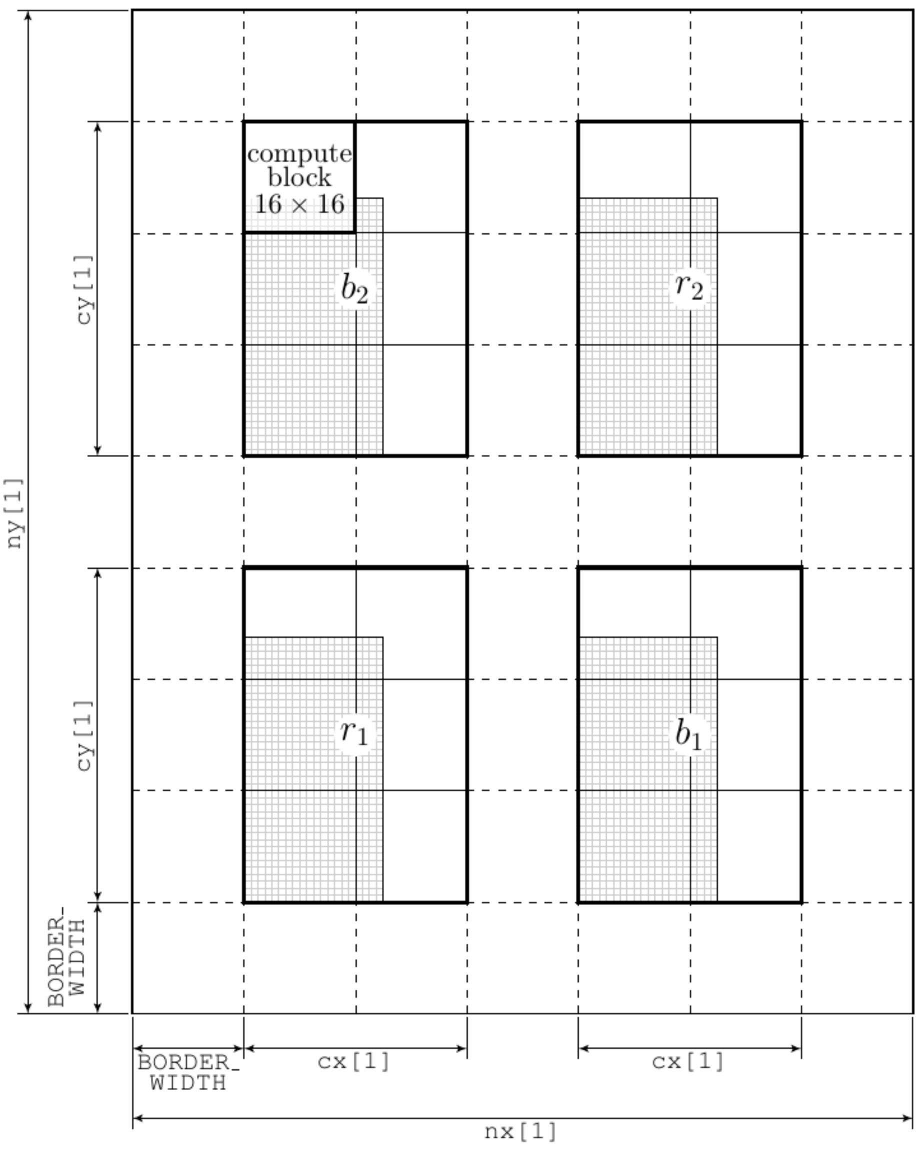


Figure 7.3: r1/r2/b1/b2 Storage Format [1]

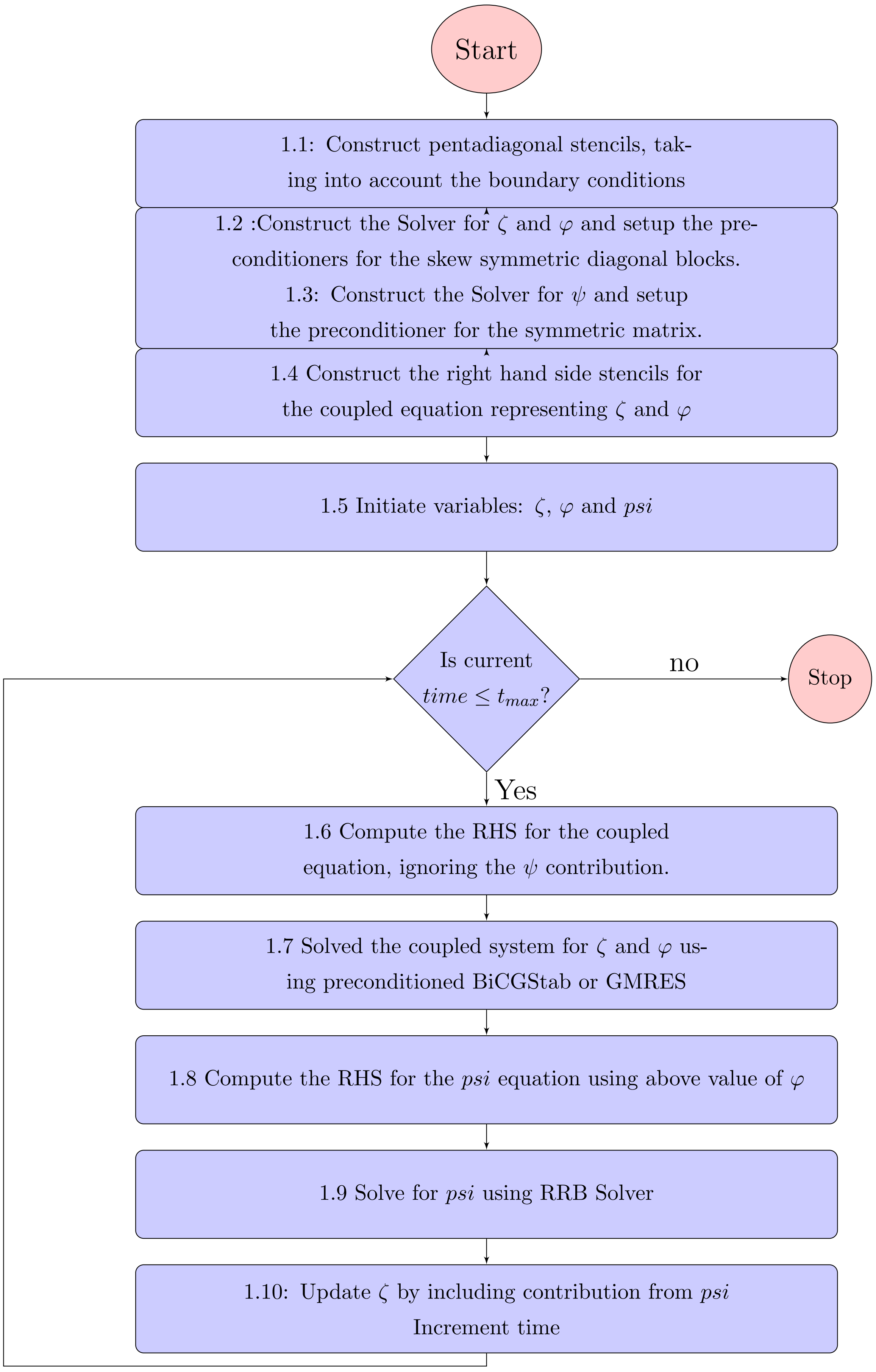

Start
1.1: Construct pentadiagonal stencils, taking into account the boundary conditions
1.2 :Construct the Solver for ζ and φ and setup the preconditioners for the skew symmetric diagonal blocks.
1.3: Construct the Solver for ψ and setup the preconditioner for the symmetric matrix.
1.4 Construct the right hand side stencils for the coupled equation representing ζ and φ
1.5 Initiate variables: ζ, φ and psi
Is current $time \leq t_{max}$?
no
Stop
Yes
1.6 Compute the RHS for the coupled equation, ignoring the ψ contribution.
1.7 Solved the coupled system for ζ and φ using preconditioned BiCGStab or GMRES
1.8 Compute the RHS for the psi equation using above value of φ
1.9 Solve for psi using RRB Solver
1.10: Update ζ by including contribution from psi Increment time

Figure 7.4: Flowchart for the Two Dimensional Simplified Test Problem

7.2 Construction of Preconditioner for the Implicit Equation

The spatial discretization matrices are stored in the Array2D structures in C++. Say for $S_{\zeta\zeta}$ which represents the pentadiagonal stencil for the hyperbolic term give in Equation (2.6), the five diagonals are stored in the variables mat_{zz}C, $mat_{zz}S$, $mat_{zz}W$, $mat_{zz}N$, $mat_{zz}E$, where the subscript zz denotes $\zeta\zeta$, and S to E denotes the directions. In the CUDA implementation, a structure *Grid* is used which contain the 9 stencils as the structure members, including the stencils corresponding to Noth-East, South-East, North-West and South-West directions. It also has structure members nx, ny, cx and cy which represent the dimensions of the corresnponding embedded grid and compute area respectively. More details can be found in [1].

The stencils are then adjusted for the boundary conditions. As we have used the Ghost cell approach in specifying the spatial discretization, it is easy to adjust the stencil at the boundary for the Neumann boundary conditions. The discretization stencil in two dimensions for $S_{\zeta\zeta}$ is given as:

$$S_{\zeta\zeta} : \begin{bmatrix} 0 & \frac{1}{2\Delta y}\overline{V_N} & 0 \\ -\frac{1}{2\Delta x}\overline{U_W} & \frac{1}{2\Delta y}\overline{V_N} - \frac{1}{2\Delta y}\overline{V_S} - \frac{1}{2\Delta x}\overline{U_W} + \frac{1}{2\Delta x}\overline{U_E} & \frac{1}{2\Delta x}\overline{U_E} \\ 0 & -\frac{1}{2\Delta y}\overline{V_S} & 0 \end{bmatrix} \tag{7.1}$$

Say for example, at the North boundary, the Neumann boundary conditions are applicable. Then we set $\zeta_{i,Nx_2+1} = \zeta_{i,Nx_2}$, which translates into the following stencil at the North boundary. Here (i, Nx_2+1) is the Ghost node.

$$S_{\zeta\zeta} : \begin{bmatrix} 0 & 0 & 0 \\ -\frac{1}{2\Delta x}\overline{U_W} & \frac{1}{2\Delta y}\overline{V_N} + (\frac{1}{2\Delta y}\overline{V_N} - \frac{1}{2\Delta y}\overline{V_S} - \frac{1}{2\Delta x}\overline{U_W} + \frac{1}{2\Delta x}\overline{U_E}) & \frac{1}{2\Delta x}\overline{U_E} \\ 0 & -\frac{1}{2\Delta y}\overline{V_S} & 0 \end{bmatrix} \tag{7.2}$$

This is programmed in C++ as follows:

Listing 7.1: Stencil construction for Neumann boundary conditions

```
for (int i1 = 1; i1 <= Nx1; ++i1)
    {
    // North Boundary
        mat_zz_c[i1][Nx2] += mat_zz_n[i1][Nx2];
        mat_zz_n[i1][Nx2] =  0;
    // South Boundary
        mat_zz_c[i1][1]   += mat_zz_s[i1][1];
        mat_zz_s[i1][1]   =  0;
    }
```

In order to construct the preconditioner for the system of equations represented by the Equation (6.11), the stencils that are part of the four blocks, namely $S_{\zeta\zeta}$, $S_{\varphi\varphi}$, $S_{\zeta\varphi}$ and $S_{\varphi\zeta}$ are copied into the solver class. The off-diagonal block matrices are symmetric in nature, whereas for the main diagonal block matrices: $L = -U^T$, where L is the strictly lower triangular part of matrix and U is the strictly upper part of the matrix. Hence, only three diagonals are required to be copied into the Solver class: South, West and Center. The North and East stencils can then be reconstructed inside the Solver class. The corresponding C++ code for the reconstruction of stencils for the diagonal block matrices is shown below.

Listing 7.2: Stencil Reconstruction inside the solver class

```
    for (int jj = 0; jj <= Nx2 + 1; ++jj)
    {
        for (int ii = 0; ii <= Nx1 + 1; ++ii)
        {
            mat_zz_n[ii][jj] = -mat_zz_s[ii  ][jj + 1];
            mat_zz_e[ii][jj] = -mat_zz_w[ii + 1][jj  ];
        }
    }
```

Please note that this reconstruction is valid only when the Neumann boundary conditions have already been applied to the stencils.

Inside the Solver class, the stencils are modified to represent the left-hand side of the Equation (6.11). This also involves division by the time step (Δt), which is copied to the Solver class. After the modification, the stencils are stored in C++ standard library format (std::vector <Array2D <REAL>>). This allows an easy access to the member matrices, and provides modularity to the preconditioner construction and preconditioning solve steps. The preconditioning matrices, $P_{\zeta\zeta}$ and $P_{\varphi\varphi}$ are then constructed respectively by following the algorithm given in Section 6.3.2.

The stencils for the right-hand side of the Equation (6.11) are constructed similarly, and stored in memory for the computation of right-hand side through Sparse matrix vector multiplication routines.

The Solver for ψ, which is represented by the elliptic equation, $S_{\psi\psi}\vec{\psi} = -S_{\psi\varphi}\vec{\varphi}$ is constructed in exactly the same manner as in [1].

7.3 Implementation of Chorin's Projection Scheme

The time loop in the Figure 7.4 follows the Chorin's projection scheme described in Chapter 6.

7.3.1 Sparse Matrix Vector Multiplication

The right-hand side of the Equation (6.11) is computed through sparse matrix vector multiplication. In Martijn's code [1], this step is carried out while computing the time derivatives for the explicit Leap Frog method. The stencils are computed on the fly. In our algorithm, it is done differently, as we need to store the matrices in any case. Thus we avoid recomputing the stencils at every time step. As we have a coupled system, the computations are done in two parts.

$$RHS_{\zeta} = \left[\frac{I}{\Delta t} - (1-\beta)S_{\zeta\zeta}\right]\vec{\zeta} + \left[(1-\beta)S_{\zeta\varphi}\right]\vec{\varphi} \tag{7.3a}$$

$$RHS_{\varphi} = \left[(1-\beta)S_{\varphi\zeta}\right]\vec{\zeta} + \left[\frac{I}{\Delta t} + (1-\beta)S_{\varphi\varphi}\right]\vec{\varphi} \tag{7.3b}$$

Computing right-hand sides for both variables ζ and φ require two sparse matrix vector mutiplications each.

Following code snippet shows the sparse matrix vector multiplication in C++. The diagonals are represented by $cn, \ldots$. They are obtained by accessing the elements of the std::vector <Array2D <REAL>>. The multiplication by X is stored in Y which has the structure of Array2D. OpenMP directives have been used to employ parallel programming on the CPU.

Listing 7.3: Sparse matrix vector multiplication in C++

```
#pragma omp for
    for (int i1 = 1; i1 < nx1; ++i1)
    {
        int i1e = i1+1;
        int i1w = i1-1;
        for (int j1 = 1; j1 < nx2; ++j1)
        {
            int j1n = j1+1;
            int j1s = j1-1;
            Y[i1][j1] = cn[i1    ][j1  ] * X[i1    ][j1n] +
                        ce[i1  ][j1    ] * X[i1e][j1    ] +
                        cs[i1    ][j1    ] * X[i1    ][j1s] +
                        cw[i1    ][j1    ] * X[i1w][j1    ]
                        + X[i1][j1]*cc[i1][j1];
        }
    }
```

The computation is done in similar fashion in CUDA. The grid representing the stencil $S_{\zeta\zeta}$ is passed to the CUDA kernel along with the vectors Y and X. A two-dimensional thread indexing is used here. The computation is done over the compute area shown in Figure 7.2. Please note that this implementation is different from the sparse matrix vector mupltiplication on the r1r2b1b2 storage format which will be explained later in the solve step.

Listing 7.4: Sparse matrix vector multiplication in CUDA

```
template <class T>
__global__ void kernel_matv1(T *y, const T *x, const Grid &g)
{
  // Computation of the Thread ID————————
    int ld = g.nx;
    int u = BORDER_WIDTH + bx * Bx + tx;
    int v = BORDER_WIDTH + by * By + ty;
    int tid = v * ld + u;
  // ————————————————————————————————————
    y[tid] = g.cc[tid]*x[tid] +
             g.nn[tid]*x[tid+ld] +
             g.ss[tid]*x[tid-ld] +
             g.ee[tid]*x[tid+1] +
             g.ww[tid]*x[tid -1];

}
```

7.3.2 Solve Step for the Coupled System

This is the backbone of the Chorin's projection scheme, and one of the most important step in current thesis. This solves the system of equations arising from the Implicit Trapezoidal approach. The algorithm has already been discussed. We first present the implementation of major functions and kernels which are used in the combined solver, whether GMRES/Bi-CGStab.

In the CUDA implementation, variables ζ and φ are restored in the r1r2b1b2 format as the preconditioning step is involved.

Vector updates

Vector updates (AXPY) is a very easy to implement routine. Following is the equivalent routine in C++.

Listing 7.5: Vector update (axpy) in C++

```
// Computes y = a*y + b*x
void axpy(Array2D<REAL> &y,  const Array2D<REAL> &x,
          const REAL a, const REAL b)
{
#pragma omp parallel for private(i1,j1)
    for (int i1 = 0; i1 < y.dim1(); i1++)
    {
        for (int j1 = 0; j1 < y.dim2(); j1++)
        {
            y[i1][j1] = y[i1][j1]*a + x[i1][j1]*b   ;
        }
    }
}
```

In the actual implementation in C++, the function axpy() is split into vector addition and scalar multiplication separately, in order avoid incurring extra FLOPS. In the CUDA implementation, the vector update is carried out over the r1r2b1b2 storage. First the location or thread index of the respective nodes are computed and then the vector is updated. In the previous implementation by Martijn [1], computation on only red nodes was carried out. Here as we do not operate on the Schur Complement as discussed before, computation is carried out over both red and black nodes.

The Code snippet is shown below:

Listing 7.6: Sparse Matrix Vector Multiplication in C++

```
 template <class T>
    __global__ void kernel_axpy(T *y, const T *x,
                                const T a, const T b, const Grid g)
    {
        int cgx = g.cx;
        int cgy = g.cy;
        int ld  = g.nx;

        int v_r1 = BORDER_WIDTH + by * By + ty;
        int u_r1 = BORDER_WIDTH + bx * Bx + tx;
        int v_r2 = BORDER_WIDTH2 + cgy + by * By + ty;
        int u_r2 = BORDER_WIDTH2 + cgx + bx * Bx + tx;

        int u_b1 = BORDER_WIDTH2 + cgx + bx * Bx + tx;
        int v_b1 = BORDER_WIDTH + by * By + ty;
        int u_b2 = BORDER_WIDTH + bx * Bx + tx;
        int v_b2 = BORDER_WIDTH2 + cgy+ by * By + ty;

        int loc_r1 = ld * v_r1 + u_r1;
        int loc_r2 = ld * v_r2 + u_r2;
        int loc_b1 = ld * v_b1 + u_b1;
        int loc_b2 = ld * v_b2 + u_b2;

        y[loc_r1] = a * y[loc_r1] + b * x[loc_r1];
        y[loc_r2] = a * y[loc_r2] + b * x[loc_r2];
        y[loc_b1] = a * y[loc_b1] + b * x[loc_b1];
        y[loc_b2] = a * y[loc_b2] + b * x[loc_b2];

    }
```

Let's take the example of the r1 nodes to understand the above code snippet. First the two-dimensional index (u,v) of the r1 node is determined using the structure shown in Figure 7.3. Then the global thread index is computed which is given by $loc_{r1} = ld * v_{r1} + u_{r1}$. Then the vector y is updated. Similarly the computation is carried out for r2, b1 and b2 nodes.

Dot Products

Two implementations of the dot product, also called as the inner product for a vector of structure Array2D<REAL> is provided for the C++ code. The first implementation uses the BLAS library. The second implementation is a generic one, which can be used if the BLAS library is not installed. This uses openmp reduction scheme. This implementation excludes the operation over the Ghost boundary nodes, and computes the dot product of only inner nodes.

Listing 7.7: Dot Product version 2 in C++

```
REAL dotproduct(const Array2D<REAL> &Vec1, const Array2D<REAL> &Vec2)
{
    REAL sum =0;
 #pragma omp parallel for reduction (+:sum)
    for (int i1 = 1; i1 < Vec1.dim1()-2; ++i1)
    {
        for (int j1 = 1; j1 < Vec1.dim2()-2; ++j1)
        {
            sum += Vec1[i1][j1]*Vec2[i1][j1];
        }
    }
    return sum;
}
```

On a GPU, it is difficult to use the existing libraries like CUBLAS as the variables are stored in r1r2b1b2 format, and these library routines require the input vectors as linear one dimensional arrays. Also, as majority of the routines for the RRB-solver are already custom built, we do not strive to use the CUBLAS library, but build upon the existing implementation from Martijn [1].

As discussed before, this is done in two steps.

- Do a mass reduction on GPU over the compute blocks, and store the intermediate results.
- Use Kahan summation on the CPU to add the elements from the intermediate results.

The reader is encouraged to go through Martijn's work[1] to understand the implementation of above two steps. Here we explain the modifications required for the computation of the dot product for all the nodes, instead of only the red nodes. The change required is in the first step, which is the mass reduction on GPU and can be achieved easily. Each compute block is divided in sub-blocks by introduction of the block-factor called DOTP_BF.

Listing 7.8: Dot product in CUDA

```
template <class T>
__global__ void kernel_dotp1(T *odata, const T *y, const T *x,
        const Grid g)
{
    int cx = g.cx;
    int cy = g.cy;
    int ld = g.nx;

    int u_r1 = BORDER_WIDTH + bx * DIM_COMPUTE_BLOCK + tx;
    int v_r1 = BORDER_WIDTH + by * DIM_COMPUTE_BLOCK + ty;
    int u_r2 = BORDER_WIDTH2 + cx + bx * DIM_COMPUTE_BLOCK + tx;
    int v_r2 = BORDER_WIDTH2 + cy + by * DIM_COMPUTE_BLOCK + ty;


    //Also include the Black Nodes
    int u_b1 = BORDER_WIDTH2 + cx + bx * DIM_COMPUTE_BLOCK + tx;
    int v_b1 = BORDER_WIDTH + by * DIM_COMPUTE_BLOCK + ty;
    int u_b2 = BORDER_WIDTH + bx * DIM_COMPUTE_BLOCK + tx;
    int v_b2 = BORDER_WIDTH2 + cy + by * DIM_COMPUTE_BLOCK + ty;

    int loc_r1 = ld * v_r1 + u_r1;
    int loc_r2 = ld * v_r2 + u_r2;
    int loc_b1 = ld * v_b1 + u_b1;
    int loc_b2 = ld * v_b2 + u_b2;
    int tid = Bx * ty + tx;

    __shared__ T sm[DIM_COMPUTE_BLOCK *(DIM_COMPUTE_BLOCK/DOTP_BF)];
    T sum = 0;

    for (int k = 0; k < DOTP_BF; ++k) {

        sum += y[loc_r1] * x[loc_r1];
        sum += y[loc_r2] * x[loc_r2];
        sum += y[loc_b1] * x[loc_b1];
```

```
            sum += y[loc_b2] * x[loc_b2];
            loc_r1 += ld * (DIM_COMPUTE_BLOCK / DOTP_BF);
            loc_r2 += ld * (DIM_COMPUTE_BLOCK / DOTP_BF);
            loc_b1 += ld * (DIM_COMPUTE_BLOCK / DOTP_BF);
            loc_b2 += ld * (DIM_COMPUTE_BLOCK / DOTP_BF);
        }
        sm[tid] = sum;
        __syncthreads();
        for (int k = (DIM_COMPUTE_BLOCK / 2) *
             (DIM_COMPUTE_BLOCK / DOTP_BF); k > 16; k >>= 1) {
            if (tid < k) {
                sm[tid] += sm[tid + k];
                __syncthreads();
            }
        }

        if (tid < 16)
            sm[tid] += sm[tid + 16];
        __syncthreads();
        if (tid < 8)
            sm[tid] += sm[tid + 8];
        __syncthreads();
        if (tid < 4)
            sm[tid] += sm[tid + 4];
        __syncthreads();
        if (tid < 2)
            sm[tid] += sm[tid + 2];
        __syncthreads();
        if (tid < 1)
            sm[tid] += sm[tid + 1];
        __syncthreads();
        if (tid == 0)
            odata[by * gridDim.x + bx] = sm[tid];
}
```

Sparse matrix vector multiplication over r1r2b1b2

Inside the solve routine, the implementation of sparse matrix vector multiplication in C++ remains the same as in Listing 7.3. In CUDA the matrix-vector product is implemented differently as the r1r2b1b2 storage has to be taken into account. For each r1,r2,b1,b2 node, first the location is determined. Then the matrix vector product computed using the stencil values adjacent to the node.

Listing 7.9: Sparse matrix cector multiplication in CUDA

```
template <class T>
__global__ void kernel_matv1(T *y, const T *x, const Grid g)
{
    int ld = g.nx; // leading dimension

    int u_r1 = BORDER_WIDTH + bx * Bx + tx;
    int v_r1 = BORDER_WIDTH + by * By + ty;
    int u_r2 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
    int v_r2 = BORDER_WIDTH2 + g.cy + by * By + ty;

    int u_b1 = BORDER_WIDTH2 + g.cx + bx * Bx + tx;
    int v_b1 = BORDER_WIDTH + by * By + ty;
    int u_b2 = BORDER_WIDTH + bx * Bx + tx;
    int v_b2 = BORDER_WIDTH2 + g.cy + by * By + ty;

    int loc_r1 = ld * v_r1 + u_r1;
    int loc_r2 = ld * v_r2 + u_r2;
    int loc_b1 = ld * v_b1 + u_b1;
    int loc_b2 = ld * v_b2 + u_b2;

    const T x_r1    = __FETCH_XX(u_r1,   v_r1,   ld);
    const T x_r1up1 = __FETCH_XX(u_r1+1, v_r1,   ld);
    const T x_r1vp1 = __FETCH_XX(u_r1,   v_r1+1, ld);
    const T x_r2    = __FETCH_XX(u_r2,   v_r2,   ld);
    const T x_r2vm1 = __FETCH_XX(u_r2,   v_r2-1, ld);
    const T x_r2um1 = __FETCH_XX(u_r2-1, v_r2,   ld);

    const T x_b1    = __FETCH_XX(u_b1,   v_b1,   ld);
    const T x_b1um1 = __FETCH_XX(u_b1-1, v_b1,   ld);
    const T x_b1vp1 = __FETCH_XX(u_b1,   v_b1+1, ld);
    const T x_b2    = __FETCH_XX(u_b2,   v_b2,   ld);
    const T x_b2vm1 = __FETCH_XX(u_b2,   v_b2-1, ld);
    const T x_b2up1 = __FETCH_XX(u_b2+1, v_b2,   ld);

    y[loc_b1] = g.cc[loc_b1]*x_b1 +
                g.nn[loc_b1]*x_r2 +
                g.ss[loc_b1]*x_r2vm1 +
                g.ee[loc_b1]*x_r1up1 +
                g.ww[loc_b1]*x_r1;

    y[loc_b2] = g.cc[loc_b2]*x_b2 +
                g.nn[loc_b2]*x_r1vp1 +
                g.ss[loc_b2]*x_r1 +
                g.ee[loc_b2]*x_r2 +
                g.ww[loc_b2]*x_r2um1;

    y[loc_r1] = g.cc[loc_r1]*x_r1 +
                g.nn[loc_r1]*x_b2 +
                g.ss[loc_r1]*x_b2vm1 +
                g.ee[loc_r1]*x_b1 +
                g.ww[loc_r1]*x_b1um1;


    y[loc_r2] = g.cc[loc_r2]*x_r2 +
                g.nn[loc_r2]*x_b1vp1 +
                g.ss[loc_r2]*x_b1 +
                g.ee[loc_r2]*x_b2up1 +
                        g.ww[loc_r2]*x_b2;
}
```

Preconditioning Solve

As discussed in Section 6.3, each iteration step requires the preconditioning step, where $Mz = r$ has to be solved for z. For the main diagonal block matrices, the preconditioning matrix can be written as $M = LDU$. Here L is the lower triangular matrix with diagonal elements as entry 1. Let L_s be the strictly lower triangular part of L. Then we can assign $U_s = -L_s^T$ where U_s is the strictly upper triangular matrix part of U. The diagonal elements of U also contain the entry 1. During the preconditioner construction step, only the strictly lower triangular part L_s and D are stored. D is the diagonal matrix.

Solving $Mz = r$ can be done in three steps as follows. Set $y := Uz$ and $x := DUz = Dy$, then:

- solve $Lx = r$ using forward substitution
- compute $y = D^{-1}x$
- solve $U^T z = y$ using backward substitution. U can be constructed with the help of $U_s = -L_s^T$.

Implementation of above three steps is described in detail in [1]. The step 1 of solving $Lx = r$ is done grid wise in two phases, where grid refers to the repeated red black ordering level. We go from the finest grid to the coarsest grid. In the case of the symmetric RRB solver, the CG algorithm operates only on the red nodes. The first level black nodes are eliminated while constructing the Schur complement. Phase 1 corresponds to updating the r2-nodes using the r1-nodes in the same level. Phase 2 corresponds to updating r1- and r2- nodes using b1- and b2-nodes in the same level. Thus starting with Phase 1, we end in Phase 1.

In the case of Skew-Symmetric preconditioning step, as the computation is done over all the nodes, we start with the phase 2 of the process instead, in which the r1- and r2- nodes are updated using the b1- and b2- nodes in the finest level.

The step 2 of the preconditioning step is easy, as it only requires division by the diagonals. Again, for the RRB symmetric solver, this division is carried out only for the r1- and r2- nodes in the finest level. In our implementation, this step is carried out over all the nodes in the finest level.

In Step 3 we go from coarse grids to fine grids and backward substitution is used. Here the phase 4 corresponds to updating r1-nodes using r2-nodes in the same level, and the phase 3 corresponds to updating b1- and b2-nodes using r1- and r2- nodes in the same level. The backward substitution starts with phase 4 and ends with phase 4, as there is no need of updating the black nodes at the finest level.

Again, two changes are required for the implementation in the case of Skew-Symmetric preconditioning step. In the RRB solver, symmetry ($L = L^T$) has been used to carry out the computations in phase 4 and phase 3. We will employ the skew symmetry here using $L = (-L)^T$. Secondly, our backward substitution will end with phase 3 instead, where the black nodes at the finest level are also updated using the red nodes at the finest level.

7.3.3 RRB Solver for ψ, and ζ correction

In order to solve the elliptic equation 2.13b for ψ , first the right-hand side is computed. This is done in C++ with a simple sparse matrix vector multiplication. The stencil corresponding to $S_{\psi\varphi}$ has already been stored in the system, and thus computing $-S_{\psi\varphi}\vec{\varphi}$ is easy. In CUDA, the previous routine 'solvepsi()' from Martijn [1] has been used.

After computing the right-hand side of the equation, the RRB solve function is called. Depending on the predefined directives, it either calls the C++ or the CUDA RRBSolver.

With the updated value of ψ, now ζ can be updated, which is the last step in the Chorin's Projection scheme. This step only requires sparse matrix multiplication and a vector update. Please note that in CUDA, the computations outside the solve steps are carried out on the embedded grid given in Figure 7.2. Only for the solve routines, the computations are carried out on the r1r2b1b2 structure given in Figure 7.3.

7.4 Plugging the Solver into the Interactive waves Code

The Solver developed above is then plugged into the Interactive wave code. As the solver for the test case has been developed only for the Neumann boundary conditions, some changes are required while taking into account the impact of Incoming Waves. We will first discuss the previous implementation of the Boundary conditions by de Jong[1] for the RRB solver and then employ the ideas in the implicit solver.

7.4.1 Incoming Boundary Conditions- Previous Implementation

Dirichlet or Incoming boundary conditions depends on the current time, and fixed parameters like wavenumber and frequency which are inputs to the model. At every time step, the incomingboundary() routine is called to generate the Incoming boundary conditions. Lets see how it is applied to Equation (2.13b). The stencil $S_{\psi\psi}$ has been constructed considering Neumann boundary conditions in mind, whereas during construction of $S_{\psi\varphi}$, the Neumann boundary conditions are not applied. First the routine boundaryzetaphi() is called which is shown below.

Listing 7.10: Dirichlet/Incoming boundary conditions

```

 if (m_bcN == 0) /* reflective Neumann boundary */
{
    for (int j1 = 1; j1 <= m_Nx1; ++j1)
    {
      phi [j1][Nx2+1] = phi [j1][m_Nx2]
                              + phi_in_N_ext [j1]
                              - phi_in_N_cen[j1];
    }
}
```

In the above code, the Incoming boundary conditions given by variables phi_in_N_ext and phi_in_N_cen are applied to the variable ϕ. These variables are generated by the incomingboundary() routine. Similarly the Incoming boundary conditions are applied to the variable ψ.

The right-hand side of Equation (2.13b) is computed with sparse matrix vector multiplication. On the North boundary, the stencil $S_{\psi\varphi}$ takes the following form:

$$S_{\psi\varphi} : \Delta x \Delta y \begin{bmatrix} 0 & \frac{1}{\Delta y^2}\overline{h_N D_N} & 0 \\ \frac{1}{\Delta x^2}\overline{h_W D_W} & -(\frac{1}{\Delta y^2}\overline{h_N D_N} + \frac{1}{\Delta x^2}\overline{h_W D_W} + \frac{1}{\Delta x^2}\overline{h_E D_E} + \frac{1}{\Delta y^2}\overline{h_S D_S}) & \frac{1}{\Delta x^2}\overline{h_E D_E} \\ 0 & \frac{1}{\Delta y^2}\overline{h_S D_S} & 0 \end{bmatrix} \tag{7.4}$$

Please note that the Neumann boundary conditions are not applied while constructing this stencil. Thus the stencil component at the North boundary given by $\frac{1}{\Delta y^2}\overline{h_N D_N}$ is not zero. When the sparse matrix multiplication at the North boundary is carried out, we get the following code:

Listing 7.11: Right-hand side contribution at the North boundary

```
    for (int j1 = 1; j1 <= m_Nx1; ++j1)
    {
      bpsi [j1][Nx2] -= phi[j1][Nx2]*S_psi_phi_C[j1][Nx2]
                      + phi[j1][Nx2+1]*S_psi_phi_N[j1][Nx2]
                      + phi[j1][Nx2-1]*S_psi_phi_S[j1][Nx2]
                      + phi[j1+1][Nx2]*S_psi_phi_E[j1][Nx2]
                      + phi[j1 -1][Nx2]*S_psi_phi_W[j1][Nx2];
    }
```

We can see that the value of ϕ at the Ghost node [j1][Nx2+1], which was set in the Listing 7.10 has been used in Listing 7.11. Now, we will see what happens if the Neumann boundary conditions ared to the stencil $S_{\psi\psi}$, which is the stencil on the left-hand side of Equation (2.13b). In that case, for the North boundary, the term $S_{\psi\psi\text{-}N}$ will be zero and $S_{\psi\psi\text{-}C'} = S_{\psi\psi\text{-}C} + S_{\psi\psi\text{-}N}$ as discussed in Listing 7.1. The matrix vector multiplication will result in :

$$\begin{aligned} S_{\psi\psi}\vec{\psi} &= S_{\psi\psi\text{-}C'}\psi[j1[Nx2] + \cdots \\ &= (S_{\psi\psi\text{-}C} + S_{\psi\psi\text{-}N})\psi[j1[Nx2] + \cdots \end{aligned} \tag{7.5}$$

whereas if we have Dirichlet/Incoming boundary conditions instead of Neumann boundary conditions, then sparse matrix multiplication should result in following:

$$\begin{aligned} S_{\psi\psi}\vec{\psi} &= S_{\psi\psi\text{-}C}\psi[j1][Nx2] + S_{\psi\psi\text{-}N}\psi[j1][Nx2+1] + \cdots \\ &= S_{\psi\psi\text{-}C}\psi[j1][Nx2] + S_{\psi\psi\text{-}N}(\psi[j1][Nx2] + \psi.in.N.ext[j1] - \psi.in.N.cen[j1]) + \cdots \\ &= (S_{\psi\psi\text{-}C} + S_{\psi\psi\text{-}N})\psi[j1][Nx2]) + S_{\psi\psi\text{-}N}(\psi.in.N.xt[j1] - \psi.in.N.cen[j1]) + \cdots \\ &= S_{\psi\psi\text{-}C'}\psi[j1][Nx2] + S_{\psi\psi\text{-}N}(\psi.in.N.xt[j1] - \psi.in.N.cen[j1]) + \cdots \end{aligned} \tag{7.6}$$

Comparing Equation (7.5) and Equation (7.6), we can easily see that there is a missing term. This has to be additionally taken into account while computing the right-hand side of the Equation (2.13b). As Neumann boundary conditions have only been applied to $S_{\psi\psi}$, we need to subtract right hand side by $S_{\psi\psi\text{-}N}(\psi.in.N.xt[j1]-\psi.in.N.cen[j1])$, for all directions respectively. This can be seen as superimposition of Dirichlet boundary conditions over Neumann boundary conditions. A similar method need to be adopted while computing the right-hand side for the implicit set of equations. This is explained in the next section.

7.4.2 Incoming Boundary Conditions- Current Implementation

Let us consider the system of coupled equations arising from the Implicit Trapezoidal time integration given in Equation (7.7).

$$(\frac{I}{\Delta t} + \beta L)q^{n+1} = (\frac{I}{\Delta t} - (1-\beta)L)q^n \tag{7.7}$$

Here, assume that Neumann boundary conditions have been applied while constructing the matrix L. Now assume that on the North boundary, we have Dirichlet (Incoming) boundary conditions, which is generated by the routine incomingwaves(). In that case, we need to superimpose the Incoming boundary conditions on the Neumann boundary conditions, as was done in the previous section. There is one catch though. In the previous section, Incoming conditions were taken at only next time step. In our case of Implicit Trapezoidal scheme, we need to account for the Incoming boundary conditions at previous time step and the new time step. This is crucial for the stability of the Trapezoidal scheme, as the Incoming boundary conditions depend on time. In order to avoid the double computation of the Incoming boundary conditions, following steps are carried out.

- Introduce a Start-up step, which initializes the Incoming waves and also solves for ψ based on the initial values of ϕ. This step is called before the time loop.
- Inside the time loop, initialize the right-hand side for the coupled implicit equation based on the previous Incoming waves. The correction due to the superimposition of Incoming waves is computed.
- Increment time and call the incomingwaves() routine to compute the Incoming boundary conditions at the new time step.
- Update the right-hand side for the coupled implicit equation by sparse matrix vector multiplication and correction due to the superimposition of Incoming waves at the new time step.

Incoming waves have a great influence over the way the final solution behaves. Devising a proper strategy to handle the boundary conditions has been a very important step in current thesis for maintaining the stability of the implicit time integration method. We encourage to implement a simplified two-dimensional test case with Incoming boundary waves in future so that the required properties can be studied in more detail and simplified way.

Chapter 8

Results - Two Dimensional Test case

In this chapter, we discuss the results of the two dimensional test problem for the C++ and CUDA implementation.

We have seen the benchmark results from MATLAB in Section 5.5. There is one difference. The boundary conditions used in 5.5 is periodic, whereas the test case built in C++ or CUDA makes use of the Neumann boundary conditions. The MATLAB code has been modified to take into account the Neumann boundary conditions by adjusting the stencil formulation.

We will directly jump to the important problem related results, which include discussion on the role of different preconditioners, number of iterations required for GMRES/Bi-CGTAB, storage requirements, and floating point operations. Then we will discuss the timing results corresponding to the C++ and the CUDA code.

8.1 Convergence properties

Here we discuss the number of iterations required to solve the coupled set of equations either by the Bi-CGSTAB or GMRES algorithm. The number of iterations required depends on the variables given below.

- stucture of the preconditioner
- the accuracy or the solver tolerance
- the time step and the grid size
- the problem size

The other parameters which are fixed during this section are:

- Ux = Uy = 1 m/s
- h = 50 m
- g = 9.81 m/s^2

8.1.1 Termination Criteria

The number of iterations required to achieve convergence depends on the tolerance chosen. We have used the relative error of the 2-norm of residual r_i with respect to the 2-norm of the initial right-hand side vector b, as the termination criterion, i.e. stop the iterative process when:

$$\frac{< r_i, r_i >}{< b_0, b_0 >} \leq psitol^2 \tag{8.1}$$

This criterion is scale invariant, which implies that increasing the number of grid points does not lead to a more stringent criterion. We have fixed the tolerance to be $1e^{-5}$. The iterative process is stopped either when we satisfy the termination criteria, or reach the maximum number of iterations.

8.1.2 Variation with Δt for various preconditioners

The matrix solved by the linear iterative solvers is given in Equation (6.11). The diagonal elements contain an $\frac{1}{\Delta t}$ term, which results in reduction in diagonal dominance of the matrix with increase in the time step. Also the overall convergence properties depend on the preconditioner chosen. This has been previously discussed in Section 6.3.2. Here M_1 refers to the block-Jacobi preconditioner given by: $M_1 = \begin{bmatrix} P_{\zeta\zeta} & 0 \\ 0 & P_{\varphi\varphi} \end{bmatrix}$. M_2 and M_3 refer to the two variants of block-Gaussian preconditioners given by: $M_2 = \begin{bmatrix} P_{\zeta\zeta} & 0 \\ S_{\varphi\zeta} & P_{\varphi\varphi} \end{bmatrix}$ and $M_3 = \begin{bmatrix} P_{\zeta\zeta} & S_{\zeta\varphi} \\ 0 & P_{\varphi\varphi} \end{bmatrix}$.

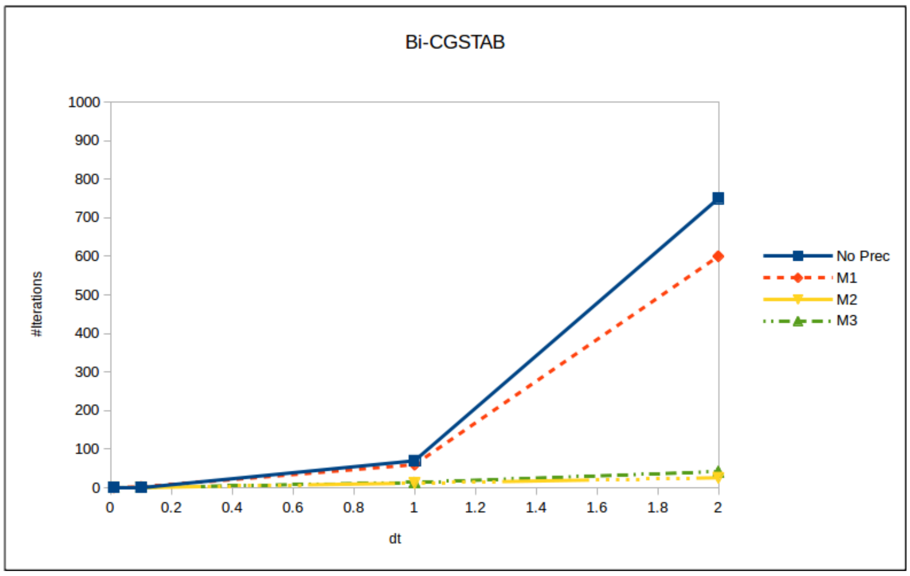


Figure 8.1: Variation of number of iterations with timestep: Bi-CGSTAB

Figure 8.1 shows the variation of number of iterations with the time step, for different preconditioners. Here Bi-CGSTAB is used as the iterative solver. The results are close to our expectations as discussed in Section 6.3.3. While the block Jacobi preconditioner does not greatly help in reducing the number of iterations required to solve the system, the Gauss Seidel preconditioners results in a large reduction in the number of iterations.

It is well known that in certain situations, the Bi-CGSTAB method suffers from stagnation. In order to analyze this, we implemented the GMRES(m) method, which is the restarted version of GMRES method discussed in Section 6.2.2. The major drawback of the GMRES method is that the amount of work and storage required per iteration rises linearly with the iteration count. In the restarted version, after a chosen number of iterations m, the accumulated data is cleared and the intermediate results are used as the initial data for the next m iterations. This procedure is repeated until convergence is reached. Here we have chosen $m = 30$. The results with the GMRES(m) method are shown in Figure 8.2.

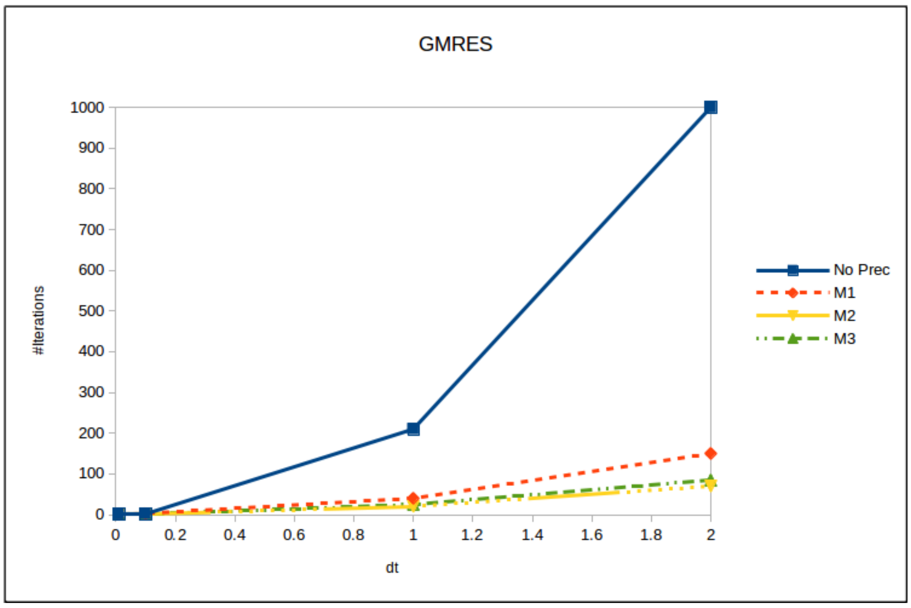


Figure 8.2: Variation of the number of iterations with timestep: GMRES(30)

Let us consider the results with preconditioner M_2 and M_3. The number of iterations required for convergence is higher for GMRES(m) and the difference increases as we take higher time steps. The comparison between the two methods is shown in Figure 8.3

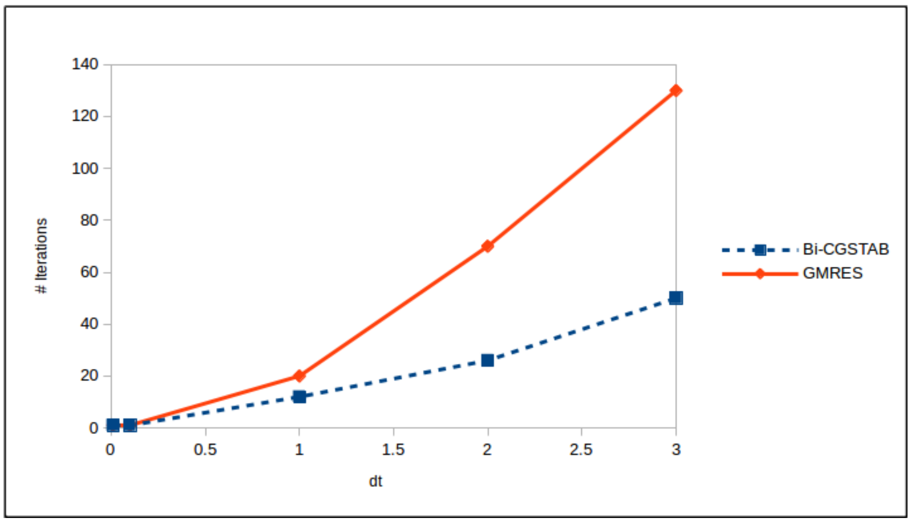


Figure 8.3: Variation of the number of iterations with timestep: GMRES vs Bi-CGSTAB for preconditioner M_2

The results encourage us to proceed with the implementation of Bi-CGSTAB on CUDA. Though there is a catch here. If we look at the time steps below 1 second, the different in number of iterations is not very large. Also, from the computation side, GMRES method only requires one sparse matrix vector multiplication and preconditioner solve per iteration whereas the Bi-CGSTAB method requires two sparse matrix vector multiplication and two preconditioner solves per iteration. A good comparison would be the timing results between the two methods.

For the CUDA implementation of the implicit solver, we have only considered Bi-CGSTAB algorithm as it was relatively easier to switch from the existing CUDA CG solver to CUDA Bi-CGSTAB solver. As the preconditioner M_2 provides the optimum number of iterations amongst the preconditioners compared, we have only considered it for further analysis, though functionality is provided in the code to switch between various preconditioners.

8.1.3 Variation with time step ratio

The Δt based on the CFL number depends on the spatial step size. If the spatial step size is reduced, then Δt_{CFL} will also reduce proportionally. We examined the behavior of the Bi-CGSTAB method with the preconditioner M_2, and plotted the required number of iterations for various $\dfrac{\Delta t}{\Delta t_{CFL}}$ ratio. This is shown in Figure 8.4.

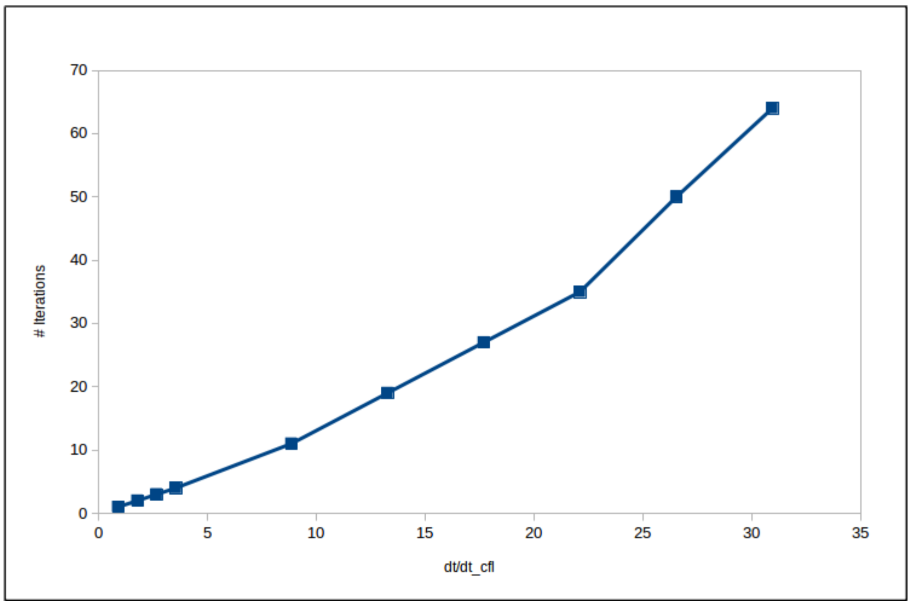


Figure 8.4: Varitaion of number of iterations with $\frac{\Delta t}{\Delta t_{CFL}}$ ratio: Bi-CGSTAB with preconditioner M_2

We do this for various step sizes (keeping the number of grid points the same). It is observed that irrespective of the step size, if the $\frac{\Delta t}{\Delta t_{CFL}}$ ratio is constant, the number of iterations remains constant too. [1] Also, we can see a linear increase in the number of iterations with the time step. Although after a certain time step, the slope is larger. It can be attributed to the impact of initial guess. If the time step is higher than the wave period, then we lose information about the wave. In this case, the initial guess for the iterative solver, which is the solution from the previous time step, may not be good enough.

Further research can be carried out in determining a suitable initial guess by extrapolating values from previous time steps. Although this would result in extra storage requirement, it is required to evaluate the trade-off between storage and efficiency for such case.

8.1.4 Impact of the problem size

To analyze the impact of the problem size, we have chosen a square grid with $N = 32, 64, 128, 256, 512, 1024$, where N is the number of unknowns in both $x-$ and $y-$ direction. The step size is fixed at $\Delta x = 5m$, and $\Delta t = 1s$. The results are gathered in Figure 8.5.

[1] This is not presented in this report.

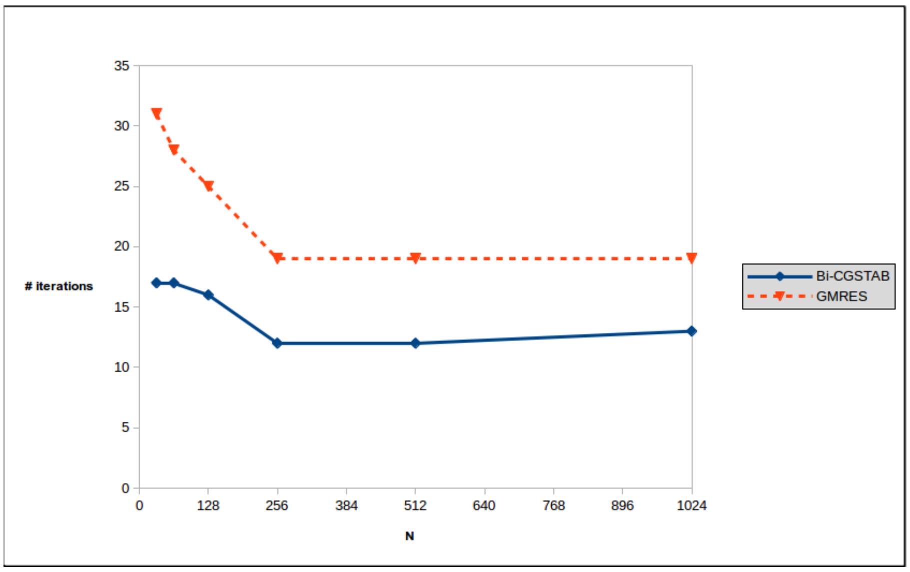


Figure 8.5: Variation of number of iterations with problem size

We can see that the iteration count is almost constant for $N > 256$. We have checked this for both GMRES and Bi-CGSTAB methods. The realistic test cases like River IJssel or Plymouth Sound have larger problem sizes. This observation becomes more attractive for them. We can say that the required number of iterations for convergence will be constant for realistic problem size.

There is a catch here. In the two-dimensional test case, the waves occupy the complete grid starting from the initial conditions. The solver then has to compute the waves at all the grid points from the first time step itself. This is different for the realisitc case. Initially the sea is at a steady-state position, and then the waves are introduced through incoming boundary waves. The required number of iterations may vary with time as the waves start filling the grid. We will discuss this in detail in the next Chapter.

8.2 Bi-CGSTAB vs CG: Memory requirement and Operation Count

In this section, we discuss the extra memory and operation count required in converting the symmetric RRB solver based on preconditioned CG to Non-symmetric RRB solver based on preconditioned Bi-CGSTAB

8.2.1 Memory requirement: C++

In C++, computing the memory requirement is straightforward as we do not need to allocate memory in different storage scheme as in CUDA.

The storage requirement for the preconditioned CG method is given in Table 8.1 [1]. Here $n = Nx1 \times Nx2$ is the total number of grid points.

Table 8.1: Storage requirements for the preconditioned CG algorithm

Variable	Type	Description	Memory
A	Matrix	System Matrix	5n
P	Matrix	Preconditioner	5n
b	Vector	Right hand Side	n
x	Vector	Solution	n
r	Vector	Residual	n
p	Vector	Search direction	n
q	Vector	$q = Ap$	n
z	Vector	$z = M^{-1}r$	n
			Total $= 16 \times n$

In the Bi-CGSTAB C++ solver, we need to allocate memory for all system matrices, preconditioners, and for the variables corresponding to both ζ and φ. Table 8.2 shows the memory requirement of the Bi-CGSTAB solver.

Table 8.2: Storage requirements for the Bi-CGSTAB algorithm

Variable	Type	Description	Memory
$S_{\zeta\zeta}, S_{\zeta\varphi}, S_{\varphi\varphi}, S_{\varphi\zeta}$	Matrix	System Matrices	$5n \times 4$
$P_{\zeta\zeta}, P_{\varphi\varphi}$	Matrix	Preconditioners	$5n \times 2$
b_ζ, b_φ	Vector	Right hand Side	$n \times 2$
x_ζ, x_φ	Vector	Solution	$n \times 2$
R_ζ, R_φ	Vector	Residual	$n \times 2$
$\tilde{R}_\zeta, \tilde{R}_\varphi$	Vector	Initial Residual	$n \times 2$
P_ζ, P_φ	Vector	p	$n \times 2$
$\hat{P}_\zeta, \hat{R}_\varphi$	Vector	$M^{-1}p$	$n \times 2$
S_ζ, S_φ	Vector	s	$n \times 2$
$\hat{S}_\zeta, \hat{S}_\varphi$	Vector	$M^{-1}s$	$n \times 2$
T_ζ, T_φ	Vector	t	$n \times 2$
			Total $= 48 \times n$

We can see that there is a three times increase in the memory required. When the PCG method is used with the explicit Leap-Frog scheme, other stencils are not stored but computed on the fly.

Lets discuss if we really need to store the stencils for the Bi-CGSTAB (or GMRES) method. The stencils $S_{\zeta\zeta}$ and $S_{\varphi\varphi}$ are used to compute the preconditioner, and they are required to be stored in memory. Though it is possible to store only three diagonals and use skew-symmetry, but that makes the code reading more complex.

The stencils $S_{\zeta\varphi}$ and $S_{\varphi\zeta}$ are used in the Bi-CGSTAB algorithm while computing the matrix-vector product. As this operation is required at every iteration (we will discuss the required number of operations in the next section), we have decided to store the stencil in memory instead of computing it at every iteration. In the previous implementation by Martijn[1], these stencils were constructed only once per time step, whereas in our case, they are required at-least two times every iteration of Bi-CGSTAB or GMRES. Though it is possible to use the symmetry of the stencil $S_{\zeta\varphi}$ and diagonal nature of stencil $S_{\varphi\zeta}$, and reduce the memory requirement from $48n$ to $42n$.

8.2.2 Memory requirement: CUDA

In this section, we shall give indications of the memory requirement for the Bi-CGSTAB solver on the GPU and compare it with the CG requirements.

In the CUDA implementation, we have seen the use of an embedded grid and the r1r2b1b2 storage format. It can be argued that the embedding results in a larger memory requirements, but that is valid only for small size problems. The embedding width is given by 16 grid points, and is not significant for larger problems that we intend to solve.

For storing the preconditioners which are constructed with the repeated red black ordering scheme, extra storage will be required, as we need to store the grid at each RRB level. At every level, we reduce the number of grid points by a factor of 4.

Thus we end up with a geometric series starting with n and ratio of 1/4. The total memory requirement (in the ideal case of very large initial grid) is given by:

$$\text{Total memory required} = \sum_{i=0}^{\infty} \frac{n}{4^i} = \frac{4 \times n}{3} \tag{8.2}$$

For smaller problems, the ratio of $\frac{4}{3}$ will be higher due to the embedding grid effect.

The list of data objects required in PCG algorithm are given below:

Table 8.3: Storage requirements for the preconditioned CG algorithm in CUDA

Variable	Type	Storage format	Memory
*m_prec	preconditioner stencils	repeated r1/r2/b1/b2	$5 \times \frac{4n}{3}$
*m_orig	Original Stencil	single r1/r2/b1/b2	$5 \times n$
*m_dX	vector x	Standard	n
*m_dB	vector b	Standard	n
*m_dXr1r2b1b2	vector x	single r1/r2/b1/b2	n
*m_dBr1r2b1b2	vector b	single r1/r2/b1/b2	n
*m_dYr1r2b1b2	vector y	single r1/r2/b1/b2	n
*m_dPr1r2b1b2	vector p	single r1/r2/b1/b2	n
*m_dQr1r2b1b2	vector q	single r1/r2/b1/b2	n
*m_dRr1r2b1b2	vector r	single r1/r2/b1/b2	n
*m_dZr1r2b1b2	vector z	repeated r1/r2/b1/b2	$\frac{4n}{3}$
*m_drem	for dot products	standard	n
			Total $= 23 \times n$

The vectors x and b are stored twice in different storage formats. Compared to the C++ PCG implementation there is an increase in storage requirement of minimum 1.4 times.

We now tabulate the storage requirements for the Bi-CGSTAB method in CUDA. In the Table 8.4, we have used the nomenclature 'lex' representing the lexicographic or standard ordering.

Table 8.4: Storage requirements for the preconditioned Bi-CGSTAB algorithm in CUDA

Variable	Type	Storage format	Memory
*m_prec$_{\zeta\zeta}$,*m_prec$_{\varphi\varphi}$	preconditioner stencils	repeated r1/r2/b1/b2	$2 \times 5 \times \frac{4n}{3}$
*m_orig$_{\zeta\zeta}$,*m_orig$_{\varphi\varphi}$	Original Stencils	single r1/r2/b1/b2	$2 \times 5 \times n$
*m_orig$_{\zeta\varphi}$,*m_prec$_{\varphi\zeta}$	Original Stencils	single r1/r2/b1/b2	$2 \times 5 \times n$
*m_dX$_\zeta$-lex, *m_dX$_\varphi$-lex	vector x	Standard	$2 \times n$
*m_dB$_\zeta$-lex, *m_dB$_\varphi$-lex	vector b	Standard	$2 \times n$
*m_dX$_\zeta$, *m_dX$_\varphi$	vector x	single r1/r2/b1/b2	$2 \times n$
*m_dB$_\zeta$, *m_dB$_\varphi$	vector b	single r1/r2/b1/b2	$2 \times n$
*m_dY$_\zeta$, *m_dY$_\varphi$	vector y	single r1/r2/b1/b2	$2 \times n$
*m_dR$_\zeta$, *m_dR$_\varphi$	vector r	single r1/r2/b1/b2	$2 \times n$
*m_d$\tilde{\text{R}}_\zeta$, *m_d$\tilde{\text{R}}_\varphi$	vector $\tilde{\text{r}}$	single r1/r2/b1/b2	$2 \times n$
*m_dV$_\zeta$, *m_dV$_\varphi$	vector v	single r1/r2/b1/b2	$2 \times n$
*m_dP$_\zeta$, *m_dP$_\varphi$	vector p	single r1/r2/b1/b2	$2 \times n$
*m_d$\hat{\text{P}}_\zeta$, *m_d$\hat{\text{P}}_\varphi$	vector $\hat{\text{p}}$	single r1/r2/b1/b2	$2 \times n$
*m_dT$_\zeta$, *m_dT$_\varphi$	vector t	single r1/r2/b1/b2	$2 \times n$
*m_d$\hat{\text{S}}_\zeta$, *m_d$\hat{\text{S}}_\varphi$	vector $\hat{\text{s}}$	single r1/r2/b1/b2	$2 \times n$
*m_dtemp	vector temp	single r1/r2/b1/b2	n
*m_dZ$_\zeta$, *m_dZ$_\varphi$	vector z	repeated r1/r2/b1/b2	$2 \times 5 \times \frac{4n}{3}$
*m_drem	for dot products	standard	n
			Total $\simeq 72 \times n$

Clearly, the CUDA implementation of the Bi-CGSTAB gives extra strain to the memory requirement as compared to the CG implementation. Martijn [1] reports that the realtistic test problem of Plymouth Sound described in Chapter 3, requires around 180 MB in storage (with 1.5 Million number of grid points). With the Bi-CGSTAB implementation, this number will increase roughly three times to 540 MB; plus the requirement of the CG solver itself. This is still under the global memory limit of 3048 MB for the GTX780 card.

8.2.3 Operation Count

We compare the floating point operations (flops) required for the preconditioned CG and the preconditioned Bi-CGSTAB method. Work is split into the dot product, vector update, matrix vector multiplication and preconditioning solve step. Let the size of the problem given by n and i be the number of iterations required for convergence. Following are the flops for each operation:

- dot product : $n-1$ summations and n multiplications $\sim 2n$
- Vector update: n summations and n multiplications $= 2n$
- sparse matrix vector multiplication : $5n$ multiplications and $4n$ summations $= 9n$
- preconditioning: Solving $LDL^T z = r$: $2n$ multiplications and summations for each lower triangular and upper triangular solve, n multiplications for division by diagonal $= 9n$

Table 8.5 provides the flop count for the PCG algorithm. The second column 'count' specifies the number of times each operation in called.

Table 8.5: Operations Count for the preconditioned CG algorithm

Operation	Count	flops per operation
dot product	$2i+1$	$2n$
vector update	$3i+1$	$2n$
Matrix vector product	$i+1$	$9n$
preconditioning solve	i	$9n$
		Total flops: $n(28i+13)$

In the case of Bi-CGSTAB algorithm, as the computation is done for ζ and φ, we have two times the operations as compared to a regular Bi-CGSTAB method. This is reflected in column 'count' in Table 8.6, where the dot products and vector updates are multiplied by 2. The matrix vector product and preconditioning solve requires special attention.

Let the coupled matrix be given as $A = \begin{bmatrix} S_{\zeta\zeta} & S_{\zeta\varphi} \\ S_{\varphi\zeta} & S_{\varphi\varphi} \end{bmatrix}$. Then the matrix vector multiplication with the vector $\vec{x} = \begin{bmatrix} \vec{\zeta} \\ \vec{\varphi} \end{bmatrix}$ is given by:

$$A\vec{x} = \begin{bmatrix} S_{\zeta\zeta}\vec{\zeta} + S_{\zeta\varphi}\vec{\varphi} \\ S_{\varphi\zeta}\vec{\zeta} + S_{\varphi\varphi}\vec{\varphi} \end{bmatrix} \tag{8.3}$$

We require two matrix vector multiplications and one vector summation for each variable ζ and φ resulting in $19n$ flops per variable.

Flops for the preconditioning solve for the coupled system depends on preconditioner chosen. Here we study the preconditioner M_2 which has the structure:

$$M_2 = \begin{bmatrix} P_{\zeta\zeta} & 0 \\ S_{\varphi\zeta} & P_{\varphi\varphi} \end{bmatrix} \tag{8.4}$$

Solving $M_2 z = r$ requires the following:

- Two preconditioning solve operations which require $9n$ flops each $= 18n$
- one matrix vector product $=9n$
- one vector summation $=n$

resulting in a total of $28n$ flops.

Table 8.6: Operations Count for the preconditioned Bi-CGSTAB algorithm

Operation	Count	flops per operation
dot product	$2 \times (5i + 1)$	$2n$
vector update	$2 \times (6i + 1)$	$2n$
Matrix vector product	$2 \times (2i + 1)$	$19n$
preconditioned solve	$(2i)$	$28n$
		Total flops: $n(176i + 46)$

The flop count of the Bi-CGSTAB is 6 times higher than that of the CG algorithm. This also makes sense, as we now solve a system of equations which has twice the number of variables and is coupled in nature. The extra computational effort gives us the flexibility of choosing a time step irrespective of the CFL number. Also, the Bi-CGSTAB algorithm has the advantage that the number of iterations i required to solve the system does not depend on the problem size. We will discuss more on the impact of the problem size and time step on the required computation time for both Bi-CGSTAB and CG solver in the next sections.

8.3 Timing Results

In this section, we shall discuss how to establish the efficiency of the different solvers. Our aim is to perform the simulations at time steps larger than what is allowed by the CFL number limitation for the explicit time stepping case. This will allow us to carrry out simulations for finer, larger and non-uniform meshes in real time. The following four quantities are defined:

- Implicit solver time: This is the time required by the Bi-CGSTAB implicit solver to compute one time step. The simulation is run for 100 seconds in real time, and an average is computed. This is reported in milli seconds (ms). For example, if the time step is 0.1 seconds, then 1000 time steps are required to compute the 100 seconds in real time, and an average of 1000 steps is considered as the Implicit solver time.

- CG solver time : This is the time required by the RRB-k solver developed by Martijn[1] for the symmetric system of equations to carry out simulation for one time step, similar to the Implicit solver time described above.
- Additional time : This is the time required by the other functions (Computing right hand side etc.). Again an average is considered.
- Total time : This is computed differently. Total time is the time required to carry out the simulation for 100 seconds. In the case if the simulation is carried out for other than 100 seconds, then it will be mentioned. This is reported in seconds(s).

We first try to analyze the Implicit solver time for the implementation in C++. To get an insight on the cost of computation for various functions involved, a profiler like Valgrind has be used. It is possible to generate the so-called callgrind-files which can thereafter be studied using a package called kcachegrind. These profilers are available in OpenSource. It is not possible to profile GPU tasks with Valgrind though. Also, it gives random results in the case OpenMP is used. We have used Valgrind for the C++ case without OpenMP implementation.

It was observed that the maximum time is consumed by the sparse matrix vector multiplication, which is required for the preconditioning solve and the matrix vector multiplications both. Later on, we will profile the CUDA implicit solver with a different profiler called nvprof.

Following tables shows the timing results for the test problem in two dimensions. The grid step size is taken as $\Delta x = \Delta y = 5m$. The relative tolerance is set at $1e^{-5}$. Single precision has been used for this test study. For the real test cases, double floating point precision will be used as some errors creep into the simulation after simiulating for linger times with single precision for the real test cases. It should be possible to remove this discrepancy and will be part of the furture work. We will run the code on C++ and CUDA for various problem sizes and time steps. Furthermore, the C++ run is divided into with or without OpenMP implementation. The algorithm chosen for solving the implicit set of equations is Bi-CGSTAB.

In the tables below, the column 'iteration' refers to the iterations required for the convergence of the Bi-CGSTAB algorithm and the rest are our time quantities. The Total time refers to runtime for 100 seconds of realtime simulation as discussed before.

Table 8.7: Timing results for C++ with a single core. $Nx = 240$ and $Ny = 120$

Δt	Iterations	Total time (s)	Implicit solver time (ms)	CG solver time (ms)
0.05	1	29.79	6.19	7.31
0.1	1	15.69	7.19	7.10
0.25	3	9.61	15.54	6.97
0.5	6	7.62	29.49	6.87
1	13	7.55	66.54	6.79

Table 8.8: Timing results for C++ with OpenMP (4 cores). $Nx = 240$ and $Ny = 120$

Δt	Iterations	Total time (s)	Implicit solver time (ms)	CG solver time (ms)
0.05	1	13.36	2.24	3.65
0.1	1	7.74	3.33	3.60
0.25	2	3.66	4.97	3.43
0.5	6	2.95	10.50	3.42
1	13	2.74	22.83	3.35

Table 8.9: Timing results with CUDA. The CG solver is still in C++ with OpenMP[2]. $Nx = 240$ and $Ny = 120$

Δt	Iterations	Total time (s)	Implicit solver time (ms)	CG solver time (ms)
0.05	1	13.33	2.20	3.61
0.1	1	7.60	3.05	3.54
0.25	3	3.85	4.67	3.77
0.5	7	2.95	10.06	3.37
1	16	2.43	19.66	3.58

Following are the observations from the above tables:

- There is a gain of three times in the Implicit solver time when we go from a single core to four core processor. This corresponds to a parallel efficiency of 75% for the Bi-CGSTAB algorithm.
- A marginal speed-up in the CUDA routines as compared to the C++ routine with OpenMP implementation is observed. This could be attributed to two reasons:
  - Size of the problem (Number of grid points). For smaller problems, it is possible that the communication time between the host (CPU) and device (GPU) become the bottleneck. This is not a concern for the OpenMP parallelization, as it uses shared memory architecture, and the communication time is minimal.
  - The number of iterations required for CUDA convergence are slightly more as compared to the C++ implementation.
- The total time for the simulation reduces till the time step of 1 second, and then it stagnates. As discussed in the previous sections, number of iterations required for convergence of Bi-CGSTAB increases linearly with the time step. Though after a certain time step the slope of the linear curve start increasing. This increases the resulant computation time leading to stagnation.

In order to check the impact of the problem size on the efficiency of the solvers, we ran the code for different problem sizes, and keeping the time step constant at 0.5 seconds.

[2]CUDA CG solver for the two-dimensional test case has not been implemented. Complete integration of the two CUDA solvers has been done only for the real test case and the Non-Uniform grid case.

Table 8.10: Timing results in C++ without OpenMP. $\Delta t = 0.5s$. Variation with problem size

$Nx \times Ny$	Iterations	Total time (s)	Implicit solver time (ms)	CG solver time (ms)
240x120	6	7.56	29.24	6.78
480x240	5	26.90	100.88	26.83
960x480	4	106.87	380.93	123.24

Table 8.11: Timing results in C++ with OpenMP. $\Delta t = 0.5s$. Variation with problem size

$Nx \times Ny$	Iterations	Total time (s)	Implicit solver time (ms)	CG solver time (ms)
240x120	6	2.97	10.42	3.38
480x240	5	12.15	43.21	13.81
960x480	4	45.16	142.32	64.95
1024x1024	4	116.874	344.612	189.287
2048x2048	4	517.471	1281.47	1113.57

Table 8.12: Timing results in CUDA Implicit Solver. $\Delta t = 0.5s$. Variation with problem size

$Nx \times Ny$	Iterations	Total time (s)	Implicit solver time (ms)	CG solver time (ms)
240x120	7	2.95	10.06	3.37
480x240	6	6.82	13.91	14.73
960x480	6	21.65	24.38	67.62
1024x1024	5	58.14	42.31	190.61
2048x2048	5	279.12	122.73	1093.67

The real impact of the CUDA implicit solver can be seen now. For the small problem size of 240x120 grid points, we did not see any improvement, but a speed-up of 10 times can be observed with the problem size of 2048x2048 points. The speedup graph is also shown below.

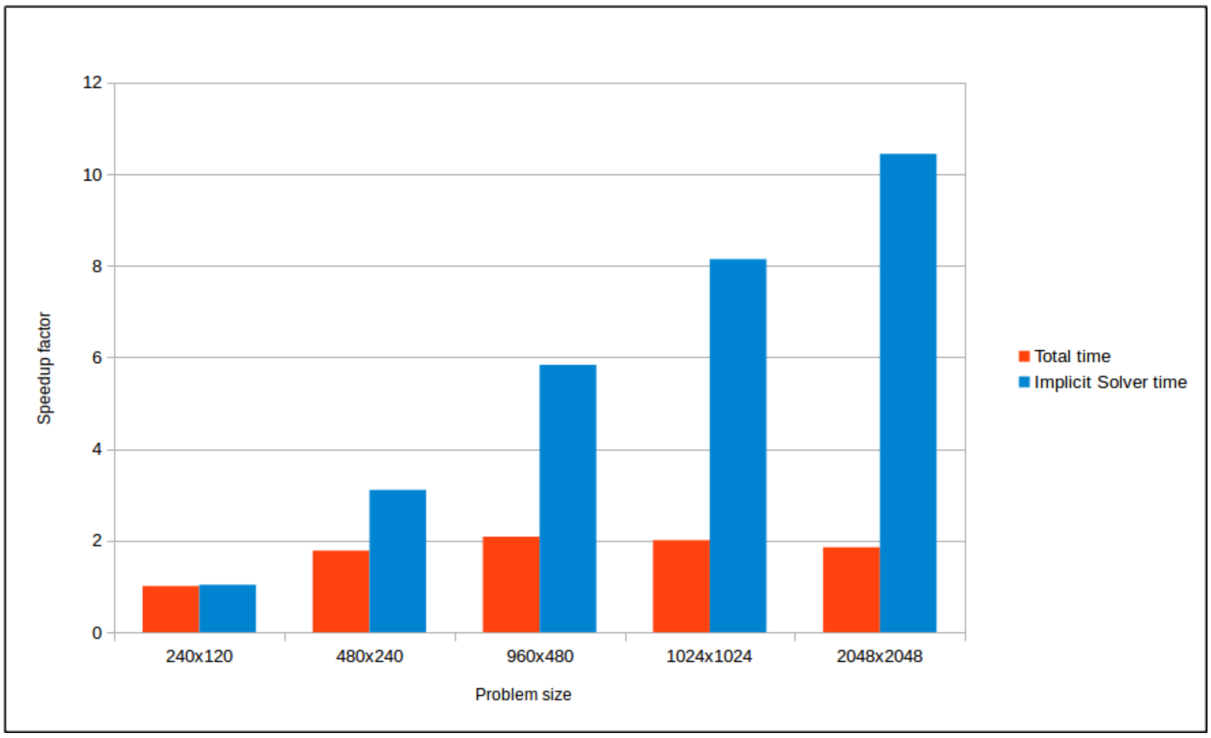


Figure 8.6: Speedup of the CUDA Implicit solver with respect to C++ solver with OpenMP implementation

While we obtain a speedup of 10 times for the implicit solver time for the increasing problem size, the speed-up for the total time saturates. The RRB solver implementation in C++ here now becomes the limiting factor, instead of the implicit solver computation. We will observe the impact of the problem size with the CUDA RRB solver implementation in the next Chapter.

8.4 CUDA profiling

As we carried out the analysis of the C++ code to identify the bottlenecks with the valgrind memory tool, we do the same with the CUDA code using Nvidia's nvprof command line utility tool.

We carried out the simulation for the test case with the problem size of 2048x2048. This allows focusing on the compute intensive parts of the program. A time step of 0.5 second is chosen which is around 5 times the time step based on the CFL condition. We saw that the number of iterations required for this case is 5. To limit profiling to the Bi-CGSTAB algorithm , we have used CUDA functions, cudaProfilerStart() and cudaProfilerStop() to start and stop profile data collection. The timing results obtained from the profile shows the distribution of all the compute kernels (or functions). The kernels then can be distributed into four major operations: preconditioning solve, matrix vector multiplication, vector update and the dot product. The results are shown in following table. The column 'time' represents the actual time spend in these function, for the simulation which was carried out for 100 seconds in real time. This excludes the time spent on communication between the host and the device.

Table 8.13: Profiling of the Bi-CGSTAB algorithm implementation

Function	Time (s)	Time (%)
Preconditioning Solve	6.543	36.18%
Matrix vector multiplication	4.952	27.39%
AXPY (vector update)	4.807	26.59%
Dot Product	1.780	9.84%

The above table provides an insight into the time consuming steps. When the tool was used initially, it showed a lot of time being spent into cuda:memcpy. The implementation at that time included initializing many variables repeatedly, and also required extra copying in the preconditioning step. Following the profiling results, it was possible to remove such extra time consuming steps and optimize the code.

The test code allowed us to study the Mathematical and Computing aspects of solving the implicit set of equations. We will now perform simulations for the real test cases and provide the timing results in the next Chapter.

Chapter 9

Results - Real Test case

In this Chapter we discuss the results for the real test cases of River IJssel and Plymouth Sound. Analysis has been carried out over various problem sizes and time steps. CUDA has been used while simulating routines for both Bi-CGSTAB and CG solvers along with the computation of other functions (right-hand side computation etc.).

Before discussing the performance of the implicit solver, we will present the stability and accuracy of results for various time steps Δt. We have used the Chorin's projection scheme as discussed in the previous Chapters to perform the time integration. It is of interest to see whether we can perform the simulations without any constraint on time step, as this was the intention with which we started our study.

We have used a smaller test case of the Plymouth Sound, with the dimensions $Nx \times Ny = 240 \times 120$ and $\Delta x = \Delta y = 5m$. The accuracy is checked by observing the plots visually and by computing Froude force (F_{ext}). Details of computation of this force is not reported in this report. If the simulations are carried out for a fixed period of time (say 100 seconds), then the Froude force at the end of 100 seconds should not change. We already know that the implicit methods at larger time steps suffer from phase lag. We will try to analyze the differences in Froude force for various time steps. Following three figures show the snapshot after 40 seconds of real-time simulation for various time steps. The CFL condition dictates that the time step Δt should be less than or equal to 0.15 seconds for the explicit methods.

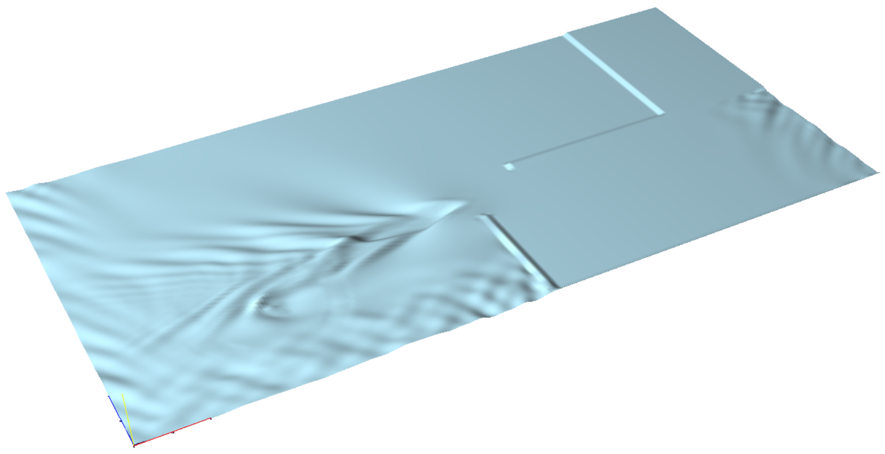


Figure 9.1: Snapshot at t=40 seconds, $\Delta t = 0.2s$

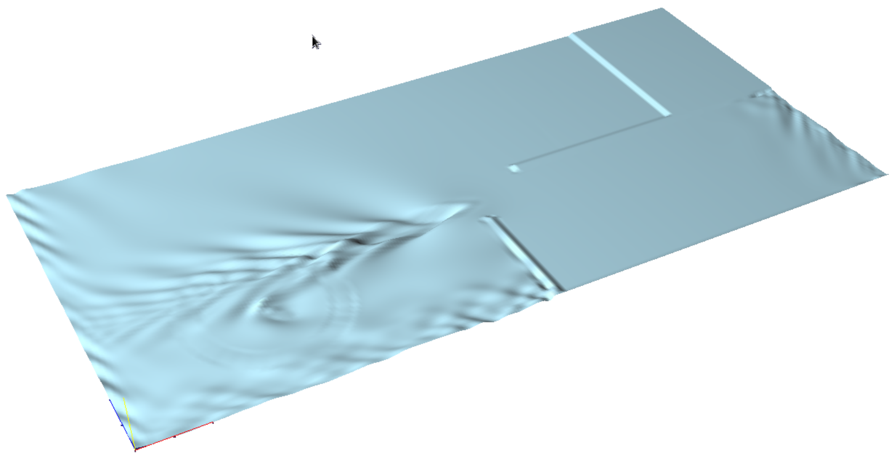


Figure 9.2: Snapshot at t=40 seconds, $\Delta t = 0.4s$

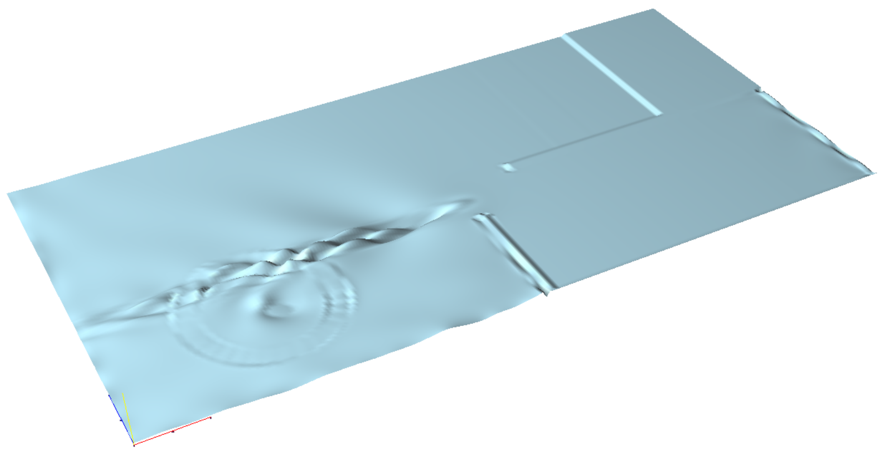


Figure 9.3: Snapshot at t=40 seconds, $\Delta t = 1.0s$

Following are the observations from the above figures:

- The solution remains stable for time steps greater than Δt_{CFL}. We checked the stability in detail by running the simulation over 1000 seconds, and by testing other problem sizes. In the case of the explicit simulator, the time step greater than 0.15 seconds cannot be used as it results in unstable solution.
- As we increase the time step, the solution behind the ship and near the boundary looks distorted. This is certainly not the impact of the phase error, and can be attributed to the following:
  - As the time step is increased, the information of the interaction of the ship with the waves during the intermediate times is lost. For example, let's take $\Delta t = 1s$. Ship's velocity is $15.5m/s$. This implies that the ship would have moved by $15.5m$ (three grid points here, as $dx = 5m$) in one time step and we lose the information of the ship's interaction with the waves for the intermediate grid points.
  - The incoming boundary conditions are computed using some predefined time periods and amplitudes. A time step should be chosen which can resolve the wave form of the

incoming boundary waves. As a rule of thumb, one time period should be covered by 10-15 time steps. For the current simulation, the minimum time period is around 4 seconds. In order to capture this time period, we need the time step to be less than or equal to 0.4 seconds. We will see later for the case of Plymouth Sound, even with a time step of 0.4 seconds, we can have slightly inaccurate results.

Based on the above results, we will limit the time step to 0.4 seconds for our discussions on timing and code profiling. Please note that, this time step limitation is due to the physical considerations and not due to the numerical restrictions imposed by the CFL condition for the explicit methods. The physical limitations remain constant irrespective of the spatial step size.

The timing results are presented in a different way as compared to the results discussed for the two-dimensional test case in the previous Chapter. Our intention is not to compute the speed-up between C++ and CUDA code, but understand the behaviour of the implicit code with the variation in the time step and the problem size, and compare it with the explicit scheme.

The four quantities to quantify time are re-defined here:

- Implicit solver time: This is the time required by the Bi-CGSTAB implicit solver to carry out the simulation for 100 seconds. This is reported in seconds(s).
- CG solver time : This is the time required by the RRB-k solver developed by Martijn for the symmetric system of equations to carry out simulation for 100 seconds
- Additional time : This is the time required by the other functions (Computing right hand side etc.).
- Total time : Total time is the time required to carry out the simulation for 100 seconds. This is the sum of the above times.

We have switched from single precision floating point numbers to double precision for CUDA simulation. It would be part of the future work to understand why the single precision accuracy is not providing stable results for the current CUDA implicit solver. One of the area which will require attention is the sparse matrix vector multiplication with the r1r2b1b2 ordering and the preconditioning solve function. The implicit solver implemented in C++ works fine with single precision.

9.1 Variation with time step

In order to see the impact of the time step on the accuracy and the speed of the solver, we have taken the problem of river IJssel with 0.5 and 1 million nodes. The Δt based on the CFL condition is 0.12 seconds. In this case, the incoming boundary waves are based on deterministic model with maximum frequency of 0.8 rad/second (long waves). Following table shows the timing results for various time steps. The column 'iteration' refers to the average iterations required for the convergence of the Bi-CGSTAB algorithm over 100 seconds.

Table 9.1: Timing results for IJssel-500000

Δt	Iterations	Total time (s)	Implicit solver time (s)	CG solver time (s)
0.1	3.3	35.98	18.43	5.15
0.2	4.97	18.65	13.1	2.34
0.4	8.88	14.63	11.87	1.15

Table 9.2: Timing results for IJssel-1000000

Δt	Iterations	Total time (s)	Implicit solver time (s)	CG solver time (s)
0.1	3.3	49.88	31.20	7.97
0.2	4.96	32.58	23.28	3.95
0.4	8.93	25.77	21.05	1.97

Following are the observations :

- The total simulation time reduces as we increase the time step.
- Increasing the problem size does not impact the required number of iterations for the Bi-CGSTAB solver. This is also similar to the results observed for the two-dimensional test case.
- As we go from $\Delta t = 0.2$ to $\Delta t = 0.4$, the implicit solver time remains almost constant as the number of iterations required for Bi-CGSTAB convergence increases propotionally with the time step.

In order to see the accuracy of the solution with respect to the time step, we plot the F_{ext} with time for various time steps.

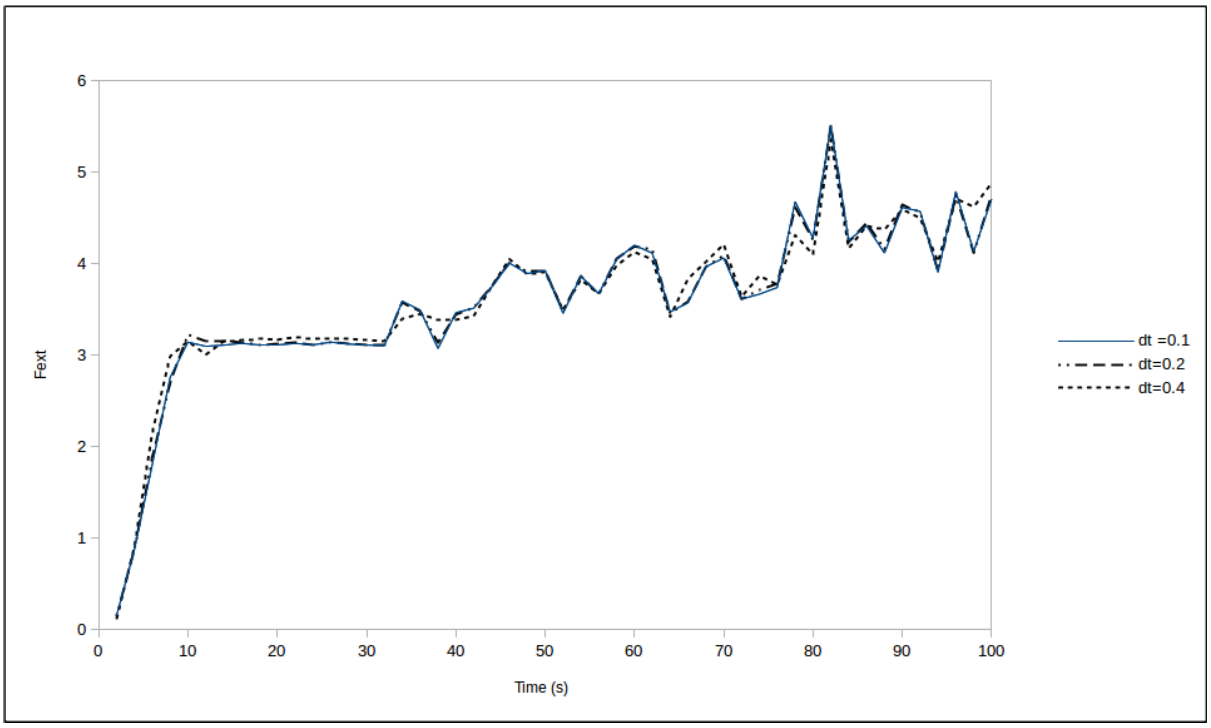


Figure 9.4: Froude force variation with time for various time steps: River IJssel, 1.0 million nodes

As the time step is increased, we can see the difference in Froude force starts increasing which can be attributed to either numerical dispersion or pressure pulse. A snapshot of the simulation for the test case of River IJssel with 1.5 million nodes is shown in Figure 9.5.

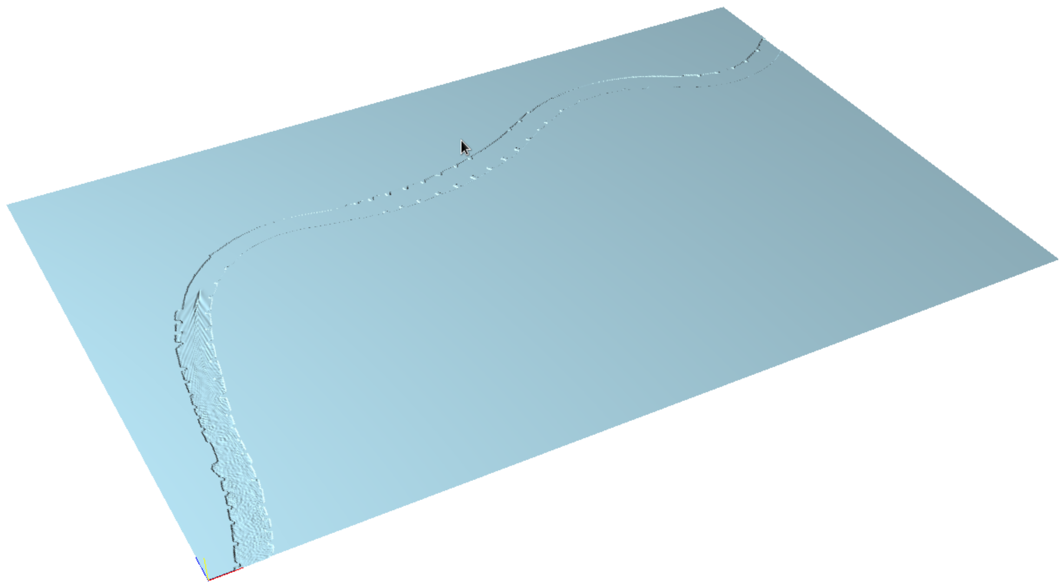


Figure 9.5: Snapshot of River IJssel with 1.5 million nodes at t=100 seconds, $\Delta t = 0.2s$

9.2 Variation with problem size

In order to see the impact of the problem size on the efficiency of the solvers, we carried out simulation for the test case of Plymouth Sound for various problem sizes, keeping the time step constant at 0.2 and 0.4 seconds. The results are shown in Table 9.3 and Table 9.4.

Table 9.3: Timing results for Plymouth. $\Delta t = 0.2s$. Variation with problem size

N	Iterations	Total time (s)	Implicit solver time (s)	CG solver time (s)
50000	4.1	4.83	2.91	0.82
100000	4.1	9.19	5.13	1.59
200000	4.2	11.63	6.53	1.76
500000	5.1	23.28	14.3	2.86
1000000	5.45	41.77	26.63	4.77
1500000	5.86	62.26	41.09	6.65

Table 9.4: Timing results for Plymouth. $\Delta t = 0.4s$. Variation with problem size

N	Iterations	Total time (s)	Implicit solver time (ms)	CG solver time (ms)
50000	6.45	3.28	2.33	0.42
100000	6.63	4.22	3.07	0.48
200000	6.7	6.20	4.62	0.67
500000	8.74	16.57	12.13	1.43
1000000	9.31	30.5	22.96	2.40
1500000	10.58	49.26	37.03	3.28

Following are the observations:

- A propotional increase in the simulation time with the problem size is observed.
- The CG solver time does not increase in proportion to the problem size. This can be attributed to slight reduction in the required number of iterations for the CG solver which was also reported by de Jong [1]
- For the Implicit solver, we can see an increase in the number of iterations. This is different from as observed for the two-dimensional test case and the real case of river IJssel. For the Plymouth Sound, there is large influence of incoming boundary waves. As the problem size increases, the waves start occupying a larger space with time. If there were no waves, then the solution will be zero, and no iterations are required. As the waves start propagating, the problem field gets filled with more and more waves, resulting in more number of iterations required for convergence.
- As we increase the time step, we can see an overall improvement in the solver performance. With the time step ratio of 2 times, the number of iterations required for the convergence of Bi-CGSTAB method does not increase by 2 times.

The snapshots for Plymouth Sound with 1.5 million nodes for different time steps are shown in Figure 9.6 and Figure 9.7.

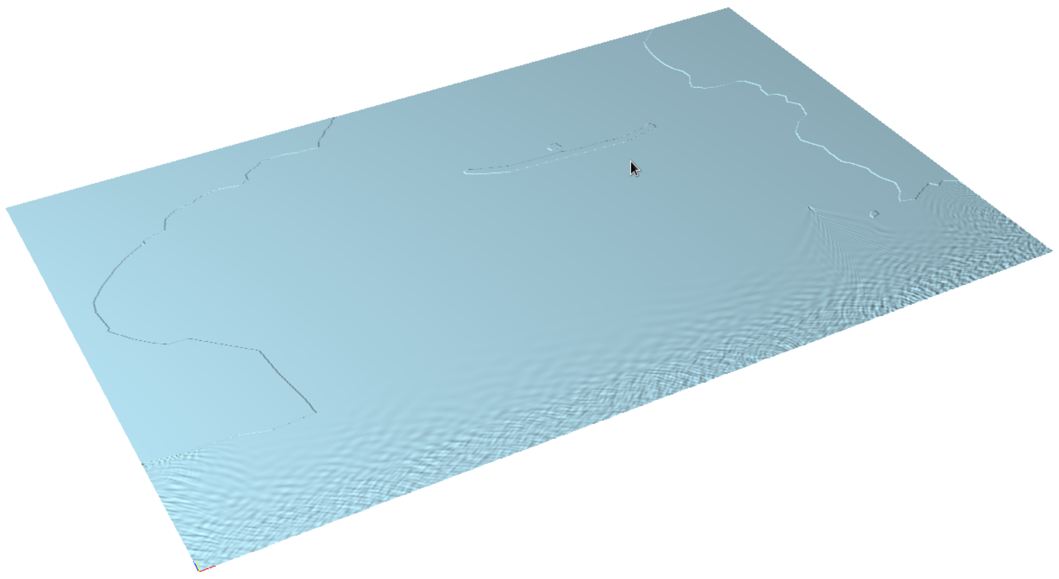


Figure 9.6: Snapshot of Plymouth Sound with 1.0 million nodes at t=100 seconds, $\Delta t = 0.2s$

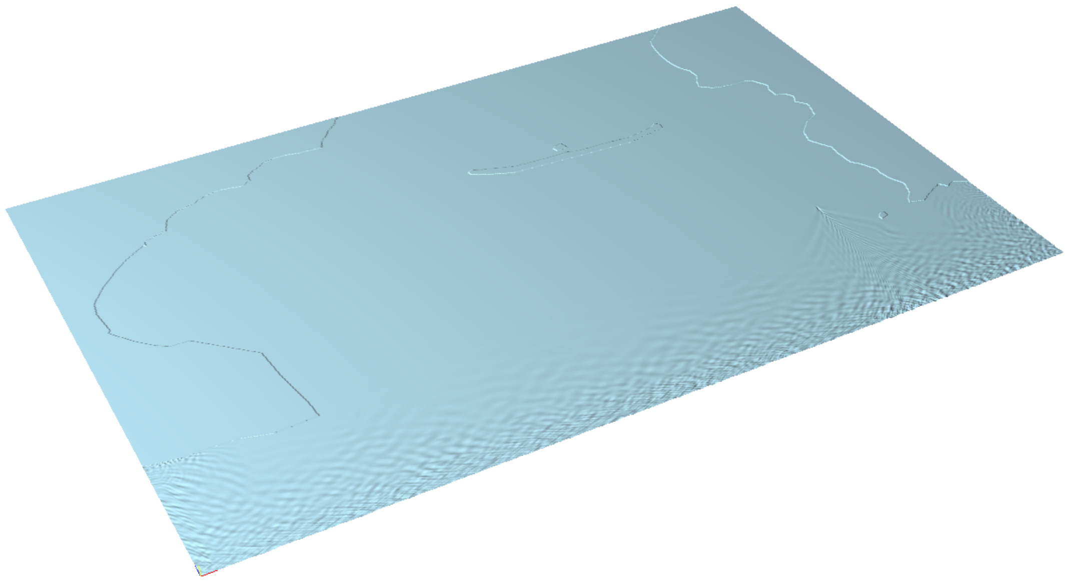


Figure 9.7: Snapshot of Plymouth Sound with 1.0 million nodes at t=100 seconds, $\Delta t = 0.4s$

There is a need to check the accuracy of the solution. In the case of river IJssel, the incoming waves have smaller frequency and thus larger time periods. Hence, we do not suffer from the inaccuracy of not capturing the incoming boundary waves. In the case of Plymouth Sound, minimum time period of the incoming waves is 4 seconds, and with a time step of 0.4 seconds we may not have enough temporal resolution. The variation of F_{ext} with time for different time steps for the test case of Plymouth Sound, 1.0 million nodes is plotted in Figure 9.8.

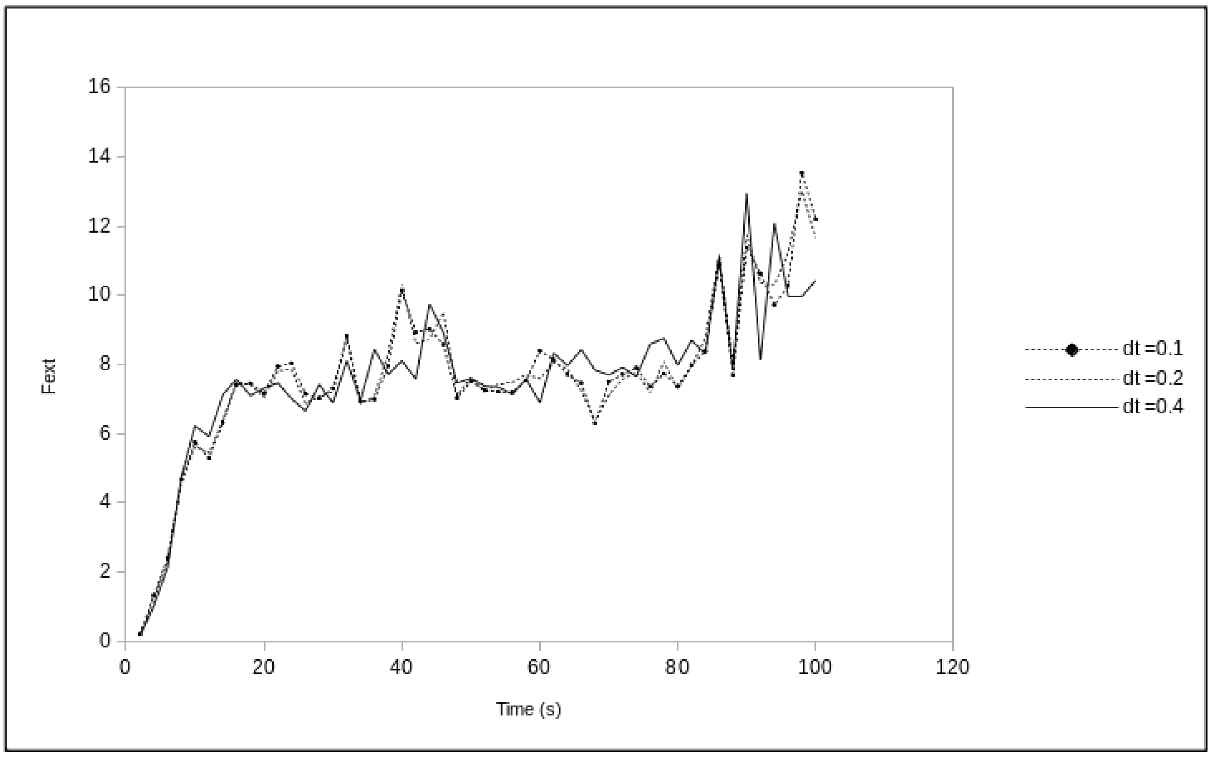


Figure 9.8: Froude force variation with time for various time steps : Port Plymouth

For $\Delta t = 0.1s$ and $\Delta t = 0.2s$, the results are relatively close. Later, the results of explicit and implicit methods are also compared for $\Delta t = 0.1s$. When we further increase the time step to 0.4s, deviation from the results at lower time steps is observed. Although this does not impact the stability of the solution. We ran the simulation for 1000 seconds of real-time and did not observe any divergence.

9.3 Profiling Results

In this study, all the functions are implemented in CUDA. A custom profiler is built inside the code, which helps to analyze the time required by each function, whether it is implemented in C++ or CUDA; thus removing the restriction caused by the profilers like Valgrind which can be used only for a C++ implementation.

Following table shows the time required by each function inside the computewave(), which computes wave for a single time step. We have compiled the time run for 20 seconds in real-time. The other computations required are related to computing and moving the ship pulses.

Table 9.5: Profiling results for CUDA implementation: Run time= 20 seconds. IJssel 1 million nodes, $\Delta t = 0.1s$

Function	Time (s)	Time (%)
Initiate RHS()	0.071	0.83 %
Incoming Waves()	0.112	1.32%
Boundary Zetaphi()	0.169	1.99%
Compute RHS zeta phi()	0.509	6.00%
Implicit Solve()	5.215	61.49%
Compute RHS psi()	0.145	1.71%
CG Solver()	1.555	18.33%
Boundary psi()	0.004	0.04%
Correct zeta()	0.001	0.01%
Boundary zeta()	0.444	5.24%
Maximum waves()	0.257	3.03%

Table 9.6: Profiling results for CUDA implementation: Run time= 20 seconds. IJssel 1 million nodes, $\Delta t = 0.4s$

Function	Time(s)	Time(%)
Initiate RHS()	0.018	0.42%
Incoming Waves()	0.028	0.66%
Boundary Zetaphi()	0.043	1.03%
Compute RHS zeta phi()	0.127	3.03%
Implicit Solve()	3.374	80.48%
Compute RHS psi()	0.036	0.86%
CG Solver()	0.392	9.35%
Boundary psi()	0.001	0.02%
Correct zeta()	0.000	0.01%
Boundary zeta()	0.111	2.64%
Maximum waves()	0.063	1.50%

The above tables provides an insight into the time consuming steps for the CUDA solver. The function boundaryzeta() also includes the transfer of ζ from the device (GPU) to the host (CPU). This time could be reduced if we later decide to plot the waves directly from the GPU and not transfer it to the CPU at every time step. Also, the time required by maximum waves() can be reduced by switching on the OpenMP flag. Futher optimization of the implicit solver is possible by reducing the memory utlization and using single precision floating numbers instead of the double precision floating numbers used in this study.

9.4 Comparison: Explicit vs. Implicit

We consider the test case of Plymouth Sound to perform the comparison between the explicit CUDA Leap-Frog solver developed by Martijn and the Implicit solver developed in this thesis. In the explicit solver, further work was done after Martijn's thesis to move the functions like computing the time integral, right-hand side etc. from C++ to CUDA. We have used this solver in our study. In the code, this is referred to as CudaWavesComputer.

First we compare the accuracy of the two solvers. We plot the F_{ext} versus time with a time step of 0.1 seconds, which is below the CFL condition number. As the time step is the same, the incoming boundary waves and pressure pulse are also captured with the same temporal resolution. The difference between the two methods methods seen in Figure 9.9 can be attributed to the numerical dispersion. In the next Chapter, we will evaluate the numerical dispersion for two methods analytically.

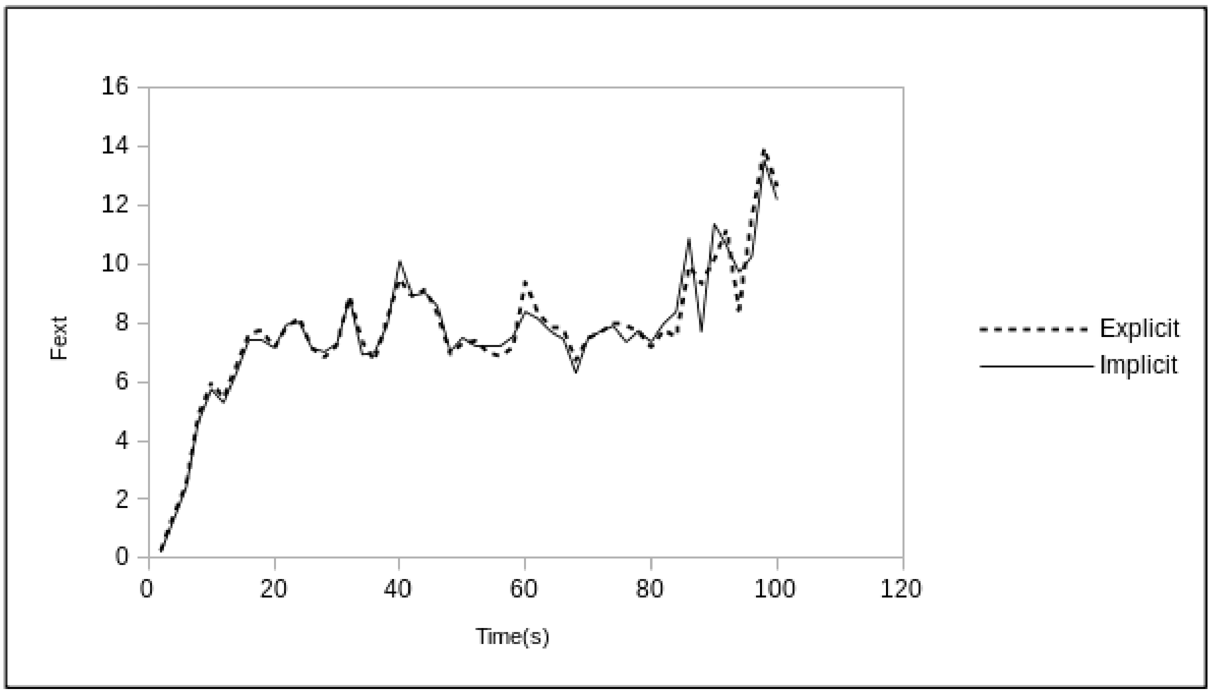


Figure 9.9: Froude force variation with time for the explicit and implicit solvers: Plymouth Sound with 1.5 Million Nodes, Δt=0.1 seconds

Next, we compare the timing results for the two methods. We conducted simulation for the explicit case with $\Delta t = 0.1$ seconds and for the implicit solver with $\Delta t = 0.2, 0.4$ seconds. Also, only the total time of the simulation is shown here for comparison.

Table 9.7: Comparison of explicit and implicit solvers: Plymouth Sound

N	Explicit $\Delta t = 0.1$	Implicit $\Delta t = 0.2$	Implicit $\Delta t = 0.4$
50000	5.99	4.83	3.28
100000	7.71	9.19	4.23
200000	6.95	11.63	6.20
500000	13.55	23.28	16.57
1000000	24.05	41.77	30.50
1500000	69.97	55.47	49.27

The above table is also shown in graph format below.

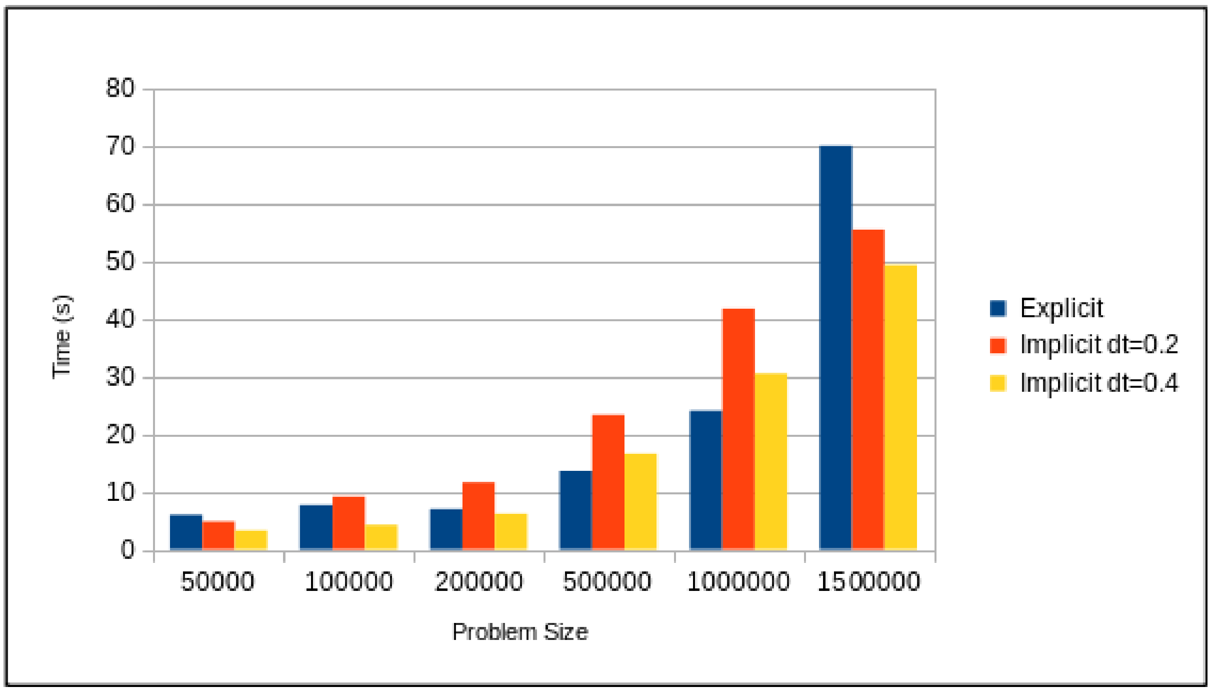


Figure 9.10: Variation of computational time for the explicit and implicit solvers with problem size: Plymouth Sound

Let us compare the explicit results at $\Delta t = 0.1$ seconds, and implicit results at $\Delta t = 0.4$ seconds. For the smaller problems, implicit solver is better than the explicit solver. This could be attributed to the memory bandwidth limitations for the small size problems. Also, number of iterations required for the Bi-CGSTAB solver for Plymouth Sound are less for the smaller problem size as discussed previously. As we increase the problem size, the number of iterations for the convergence of Bi-CGSTAB method increases thus increasing the overall time of the implicit solver. This we see for the problem sizes above 5000,000 nodes upto 1.0 million nodes. The explicit solver requires less time now.

For the problem size of 1.5 million nodes, we see a sudden increase in the overall time required for the explicit scheme. The reason is that for the case setup of 1.5 million nodes, the pulses are defined based on a mesh file. This makes the moving and computing ship pulses a compute intensive operation. Around 22.8% of the overall computation time is spent on pulses. In contrast, if the pulses are not defined through meshes, then only 3-4% of the computation time is spent on pulses.

When we move to the implicit method with larger time step of $\Delta t = 0.4$ seconds, we subsequently reduce the computation time of the pulse movement. Only 6.7% of the overall computation time is now spent on computing and moving the pulses as compared to the 22.8% of the time required by the explicit solver.

We will try to assess this impact of increase in problem size, for the River IJssel test case. The results are shown in Table 9.8.

Table 9.8: Comparison of explicit and implicit solvers: River IJssel

N	Explicit (s)	Implicit $\Delta t = 0.2$	Implicit $\Delta t = 0.4$
500000	21.17	18.65	14.63
1000000	22.96	32.73	25.77
1500000	56.01	55.53	42.02

Following are the observations:

- For the smaller problems, implicit solver is faster than the explicit solver for both test cases. The small number of iterations required for the convergence of the Bi-CGSTAB solver for smaller problems could be one of the reason.
- For the test case of 0.5 Million and 1.0 Million Nodes, the time required for simulation is almost the same. In fact, the explicit solver is faster.
- As we further increase the problem size, with $\Delta t = 0.4$ seconds which is four times larger than that of the explicit scheme Δt, again the implicit solver becomes faster than the explicit solver. Please note that the number of iterations required for convergence of the Bi-CGSTAB algorithm for the test case of River IJssel does not increase with the increase in problem size, as in the case of Plymouth Sound.

9.5 Concluding Remarks- Realistic Test Case

- The Chorin's projection scheme provides stable results for the realistic test cases.
- The implicit solver is faster than the explicit solver for certain test cases. Though in order to achieve this we need to use larger time steps which can result in large numerical dispersion. In the next Chapter, we will discuss more about the numerical dispersion for the implicit and the explicit schemes.
- The time step is governed by the required frames per second (fps) of the Ship simulator. It is intended to achieve 20 fps in real-time, which implies a time step of 0.05 seconds. With the current problem set-up, $\Delta t_{cfl} > 0.1s$ and the explicit solver can be used at 0.05 seconds. Using the implicit solver at 0.05 seconds won't result in improvement in performance.
  But this solver will scale linearly with the problem size and cannot solve large problems under 0.05 seconds. In that case, it would be required to move to the Non-Uniform grids, which results in less number of grid points.
  Also, it is intended to simulate more than one ships and capture the interaction between them. Such a setup will require a finer mesh near the interaction region. If we move towards a Non-Uniform grid set-up, it would be possible to have a finer grid near the ships and coarser grid away from the ship. With the explicit scheme, this might impose a time step limitation as the CFL number will be governed by the smallest spatial step size. Our implicit solver will become handy in such cases, as it is not restricted by the CFL number. In the next Chapter, we will provide a framework for setting up the Non-Uniform grid for the two-dimensional test case.

Chapter 10

Dispersion Analysis

Lets consider the one dimensional wave equation given by:

$$\frac{\partial \zeta}{\partial t} + c_0 \frac{\partial \zeta}{\partial x} = 0 \tag{10.1}$$

where ζ is the wave height, and c_0 is the wave velocity. In our system represented by the Variational Bousinessq approximation, the wave velocity is given by the surface gravity wave. The spatial derivative is discretized using central differences, and the time derivative is integrated with the fully implicit theta method and the explicit Leap-Frog method.

The theta-method discretization is given as

$$\zeta_i^{n+1} = \zeta_i^n - \frac{C\theta}{2}(\zeta_{i+1}^{n+1} - \zeta_{i-1}^{n+1}) - \frac{C(1-\theta)}{2}(\zeta_{i+1}^n - \zeta_{i-1}^n) \tag{10.2}$$

where C is the Courant number given by $C = c_0 \Delta t / \Delta x$. The Leap-Frog discretization is given by:

$$\zeta_i^{n+1} = \zeta_i^{n-1} - C(\zeta_{i+1}^n - \zeta_{i-1}^n) \tag{10.3}$$

The Taylor series expansion gives the modified equivalent partial differential equation for the theta method as [9]:

$$\frac{\partial \zeta}{\partial t} + c_0 \frac{\partial \zeta}{\partial x} = \frac{c_0^2 \Delta t}{2}(2\theta - 1)\frac{\partial^2 \zeta}{\partial x^2} - \frac{c_0 \Delta x^2}{6}(1 + C^2(3\theta - 1))\frac{\partial^3 \zeta}{\partial x^3} + O(\Delta x^4) \tag{10.4}$$

and the modified equivalent partial differential equation for the Leap-Frog is given as:

$$\frac{\partial \zeta}{\partial t} + c_0 \frac{\partial \zeta}{\partial x} = -\frac{c_0 \Delta x^2}{6}(1 - C^2)\frac{\partial^3 \zeta}{\partial x^3} + O(\Delta x^4) \tag{10.5}$$

As expected for the theta method, numerical diffusion arises with the magnitude:

$$\kappa_{num} = \frac{c_0^2 \Delta t}{2}(2\theta - 1) \tag{10.6}$$

and this numerical diffusion vanished with $\theta = 0.5$, which is the Implicit-Trapezoidal method.

For a given value of the Courant number, the numerical dispersion for the Trapezoidal method is always greater than the Leap-Frog scheme. In fact, if $C = 1$, the numerical dispersion vanishes for the Leap-Frog scheme. This is not the case for the Trapezoidal method, which gives a finite dispersion of $c_0\Delta x^2/4$ when $C = 1$.

10.1 Dispersion Relation

The exact dispersion relation is obtained via a substitution of $\zeta(x,t) = \hat{\zeta}exp[i(kx - \omega t)]$ which gives the relationship between the frequency and the wavenumber as :

$$\omega = c_0 k \tag{10.7}$$

The phase speed is then given by $c = \dfrac{\omega}{k} = c_0$ and hence the equation is not dispersive since all the waves propogte at the same speed c_0 regardless of the wavenumber k.

Deriving the dispersion relation for the Implicit-Trapezoidal method and the Leap-Frog method from the Taylor series expansion, we obtain:

$$\frac{c}{c_0} = \begin{cases} 1 - \frac{1}{6}(k\Delta x)^2(1 + \frac{C^2}{2}) & \text{Theta method} \\ 1 - \frac{1}{6}(k\Delta x)^2(1 - C^2) & \text{Leap-Frog method} \end{cases}$$

These show that $c = c_0$ only when the Leap-Frog method is used and when $C = 1$. This is not possible for the Implicit Trapezoidal method.

The above relation holds only in the case of small $k\Delta x$ as we have ignored the higher order terms in the Taylor series expansion. A general dispersion relation can be derived by substituting:

$$\begin{aligned} \zeta(x_{j\pm1}, t^{n\pm1}) &= \hat{\zeta}exp[i(kx_{j\pm1} - \omega t^{n\pm1})], \\ &= \hat{\zeta}exp[i(kx_j - \omega t^n)]exp(\pm ik\Delta x)exp(\mp i\omega\Delta t), \\ &= \zeta(x_j, t^n)exp(\pm ik\Delta x)exp(\mp i\omega\Delta t) \end{aligned}$$

If we substitute the above relation into the discretized equations, we get the following relations. The numerical dispersion relation for the Implicit Trapezoidal method is given by:

$$\frac{c}{c_0} = \frac{i}{Ck\Delta x}log\left[\frac{2 - iCsink\Delta x}{2 + iCsink\Delta x}\right] \tag{10.8}$$

and for the Leap Frog scheme is given by:

$$\frac{c}{c_0} = \frac{sin^{-1}(Csink\Delta x)}{Ck\Delta x} \tag{10.9}$$

Figure 10.1 shows the variation of c/c_0 with $k\Delta x$ for various Courant numbers.

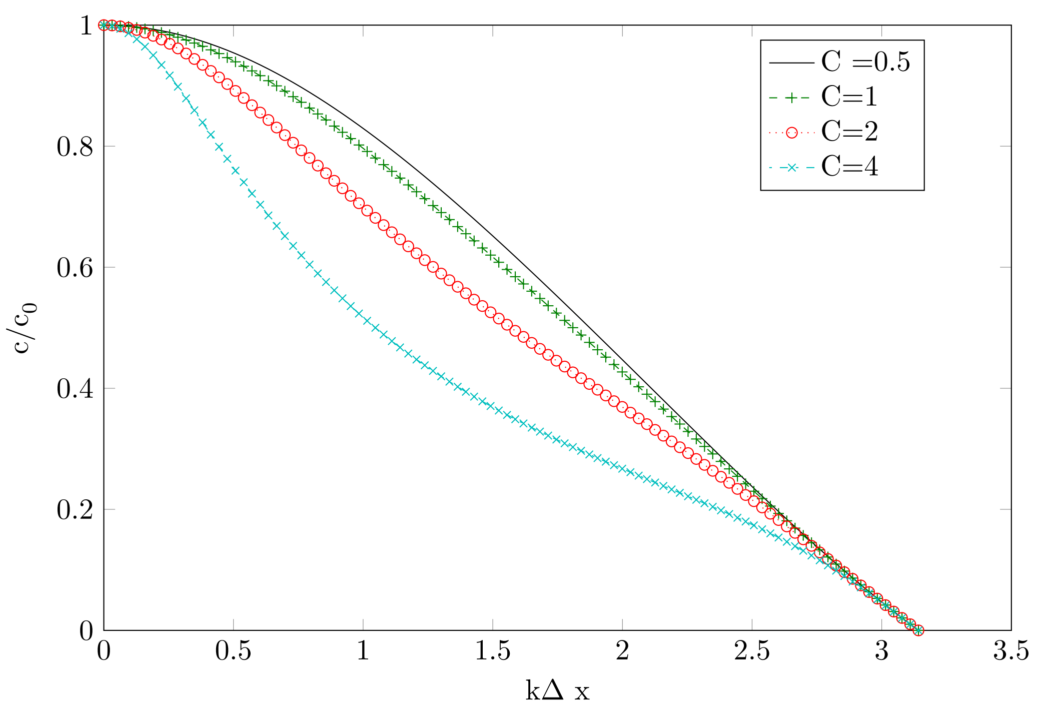


Figure 10.1: Dispersion analysis of Implicit Trapezoidal method

In order to limit the dispersion error, we need to formulate the problem such that we are near the top part of the curve. For example, to get under 10% dispersion error, and achieve a time step above the CFL limit, say $\Delta t = 2 \times \Delta t_{CFL}$, then $k\Delta x$ should be below 0.5.

10.2 Numerical Dispersion for One dimensional test case

For the one dimensional test case discussed in Chapter 5, we discuss the impact of numerical dispersion with various time steps for the Implicit Trapezoidal method. We consider a fixed location $x = 50m$, and observe the wave passing through for various time steps. The results are shown in Figure 10.2.

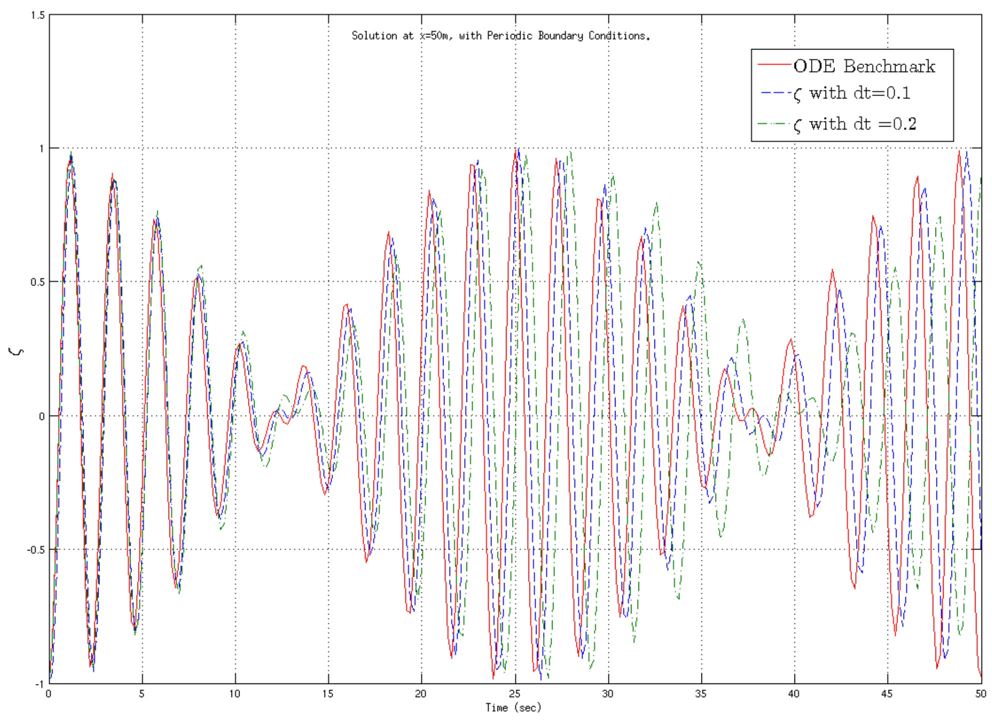


Figure 10.2: Phase lag for the Implicit Trapezoidal scheme. Solution at x=50m with periodic boundary conditions

We can see a clear phase lag, which increases for larger time steps. It is difficult to quantify this error, as we have a coupled set of equations. Although a general trend can be observed.

10.3 Non-Uniform Grid

In this section we develop the discretization scheme for the non-uniform grid and explore the possiblity of employing the implicit solver for the non-uniform grid.

We focus on a rectilinear grid in two dimensions as shown in Figure 10.3.

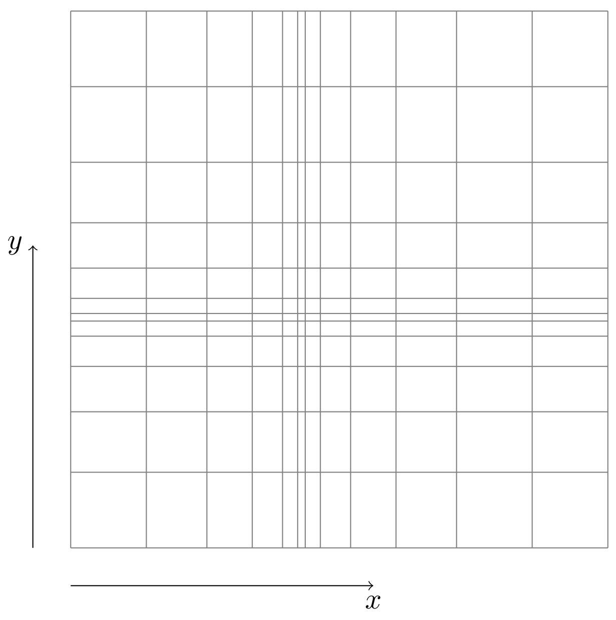


Figure 10.3: Rectilinear grid in two dimensions

The grid size in x and y direction varies smoothly. The grid size distribution in the x-direction is shown in Figure 10.4. Near the boundaries, the grid size is constant. In the central area, it varies quadratically and is continuous in nature. Though the first derivative of such grid size ditribution has a discontinuity. Based on the parameters of the quadratic curve, the ratio of the maximum grid size to the minimum grid size can be varied.

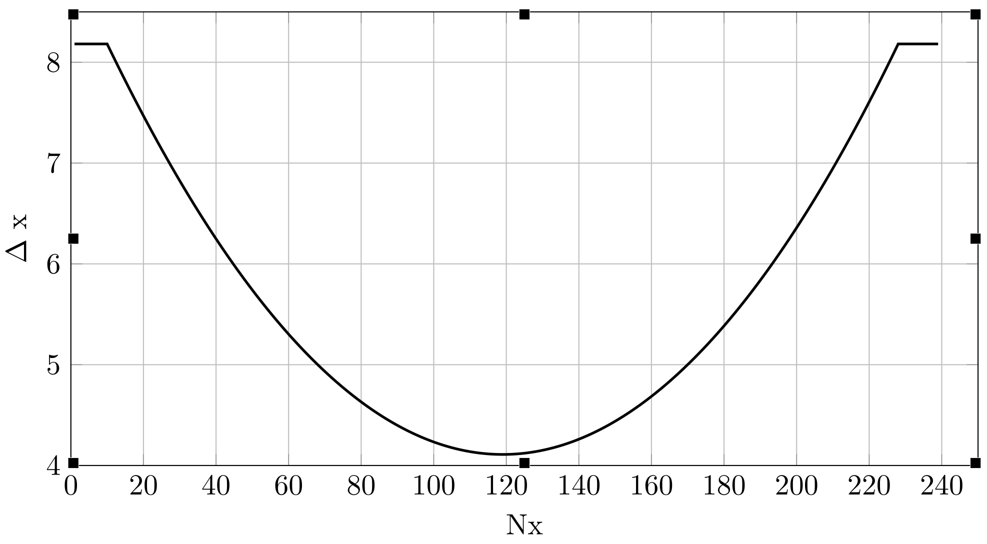


Figure 10.4: Grid size distribution in one dimension

For the two-dimensional rectilinear grid, the y-spacing can also vary with similar distribution.

In order to access elements, we do not need to store the location of all the grid points. As the grid is rectilinear, we store the x-coordinates and y-coordinates in two separate arrays x and y. When we have to access any grid element (i,j), its location can be determined as $x(i), y(j)$.

10.3.1 Spatial discretization

The governing equations are discretized based on the grid shown in Figure 10.5.

The variables are evaluated at the grid points (i, j) and the finite volume as shown as the dotted rectangle in Figure 10.5 is centered around the grid point.

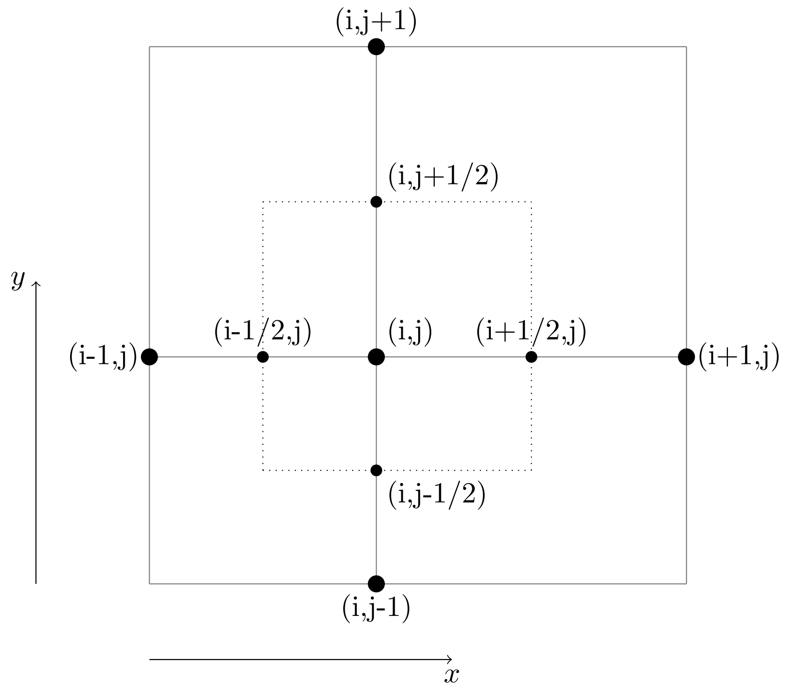


Figure 10.5: Finite Volume grid

We define the following:

- $\Delta x_i = \dfrac{x_{i+1} - x_{i-1}}{2}$
- $\Delta y_j = \dfrac{y_{j+1} - y_{j-1}}{2}$

The governing equation for variable ζ is given by:

$$\frac{\partial \zeta}{\partial t} + \mathbf{U}_x \frac{\partial \zeta}{\partial x} + \mathbf{U}_y \frac{\partial \zeta}{\partial y} + h(\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2}) - h\mathcal{D}(\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2}) = 0, \tag{10.10}$$

Lets consider discretization of each term separately. For the time derivative,

$$\iint \frac{\partial \zeta}{\partial t} dxdy = \frac{\partial \zeta_{i,j}}{\partial t} \Delta x_i \Delta y_j \tag{10.11}$$

For the single spatial derivative in x-direction,

$$\begin{aligned}\iint U_x \frac{\partial \zeta}{\partial x} dxdy &= U_x(\zeta_{i+1/2,j} - \zeta_{i-1/2,j})\Delta y_j \\ &= U_x(\frac{\zeta_{i+1,j} + \zeta_{i,j}}{2} - \frac{\zeta_{i,j} + \zeta_{i-1,j}}{2})\Delta y_j \\ &= U_x(\frac{\zeta_{i+1,j} - \zeta_{i-1,j}}{2})\Delta y_j\end{aligned}$$

Similarly,

$$\iint U_y \frac{\partial \zeta}{\partial y} dxdy = U_y (\frac{\zeta_{i,j+1} - \zeta_{i,j-1}}{2}) \Delta x_i \tag{10.12}$$

$$\begin{aligned}
\iint h \frac{\partial^2 \varphi}{\partial x^2} dxdy &= h \left(\frac{\partial \varphi}{\partial x}|_{i+1/2,j} - \frac{\partial \varphi}{\partial x}|_{i-1/2,j} \right) \Delta y_j \\
&= h \left(\frac{\varphi_{i+1,j} - \varphi_{i,j}}{x_{i+1} - x_i} - \frac{\varphi_{i,j} - \varphi_{i-1,j}}{x_i - x_{i-1}} \right) \Delta y_j \\
&= \varphi_{i+1,j} \frac{h\Delta y_j}{x_{i+1} - x_i} + \varphi_{i-1,j} \frac{h\Delta y_j}{x_i - x_{i-1}} - \varphi_{i,j} \left(\frac{h\Delta y_j}{x_{i+1} - x_i} + \frac{h\Delta y_j}{x_i - x_{i-1}} \right)
\end{aligned}$$

Similarly,

$$\iint h \frac{\partial^2 \varphi}{\partial y^2} dxdy = \varphi_{i,j+1} \frac{h\Delta x_i}{y_{j+1} - y_j} + \varphi_{i,j-1} \frac{h\Delta x_i}{y_j - y_{j-1}} - \varphi_{i,j} \left(\frac{h\Delta x_i}{y_{j+1} - y_j} + \frac{h\Delta x_i}{y_j - y_{j-1}} \right)$$

The above discretization results in formation of pentadiagonal matrices, represented by 5-point stencils. In order to use the RRB-Solver, we need to show that these matrices are either skew-symmetric or symmetric in nature.

As done previously with the Uniform grid, we do not divide the whole equation by $\Delta x \Delta y$ term, which comes with the time integral. This is done because $\Delta x \Delta y$ now varies with the grid, and we would like to keep this non-uniform part with the diagonal term.

The stencil for the convective term is give by:

$$S_{\zeta\zeta} : \begin{bmatrix} 0 & U_y \dfrac{\Delta x_i}{2} & 0 \\ -U_x \dfrac{\Delta y_j}{2} & 0 & U_x \dfrac{\Delta y_j}{2} \\ 0 & -U_y \dfrac{\Delta x_i}{2} & 0 \end{bmatrix} \tag{10.13}$$

To verify the symmetry (or skew-symmetry) of the matrix, we need to compare the outer diagonals with each other as was done for the uniform grid case. In this case, the absolute contribution from the West and the East term should be same, i.e. , $-U_x \frac{\Delta y}{2}|_{i,j} = -\left[U_x \frac{\Delta y}{2}|_{i-1,j} \right]$. This is true as Δy remains constant if we traverse in the x-direction. Similarly we can prove that the diffusive term is also symmetric in nature.

Please note that the above derivation is applicable only for the Rectilinear grids. Also, if we were to divide the equation by $\Delta x_i \Delta y_j$, then we would need $-U_x \frac{1}{2\Delta x}|_{i,j} = U_x \frac{1}{2\Delta x}|_{i-1,j}$, which is true only when $\Delta x_i = \Delta x_{i-1}$, but that is the uniform grid case.

Based on the above formulation, we can now formulate the stencils for the two-dimensional case and use the RRB-solver (both implicit and symmetric).

10.3.2 Accuracy of the Non-Uniform Grid

Accuracy of the spatial discretization described above can be analyzed by expanding the derivatives with the Taylor series and comparing the results. The single derivative is accurate upto $O((x_{i+1}-x_i)((x_i-x_{i-1}))$ i.e. to the second order in the space increments. The second derivative is accurate upto $O((x_{i+1}-x_i)-((x_i-x_{i-1}))$ i.e. to the first order in the difference of the grid intervals. In the case of uniform grid, this would be just $O(\Delta x^2)$, but is higher for non-uniform grid and depends on the type of stretching in the grid.

Another impact of the Non-Uniform grid could be seen on the numerical dispersion. As we saw before that the numerical dispersion depends on the CFL number and $k\Delta x$. Also, that was the case in only one dimension. In the case of non isotropic grid, that is, if the number of points per wavelength are not the same in every direction, it adds to more dispersion. Also if the grid size is varying, then the numerical wave speed would also be different, and this causes effects like reflection of the wave etc.

We will try to observe these effects in the results given in the next section.

10.3.3 Non-Uniform Grid Results

The grid used in the simulations is shown in Figure 10.6.

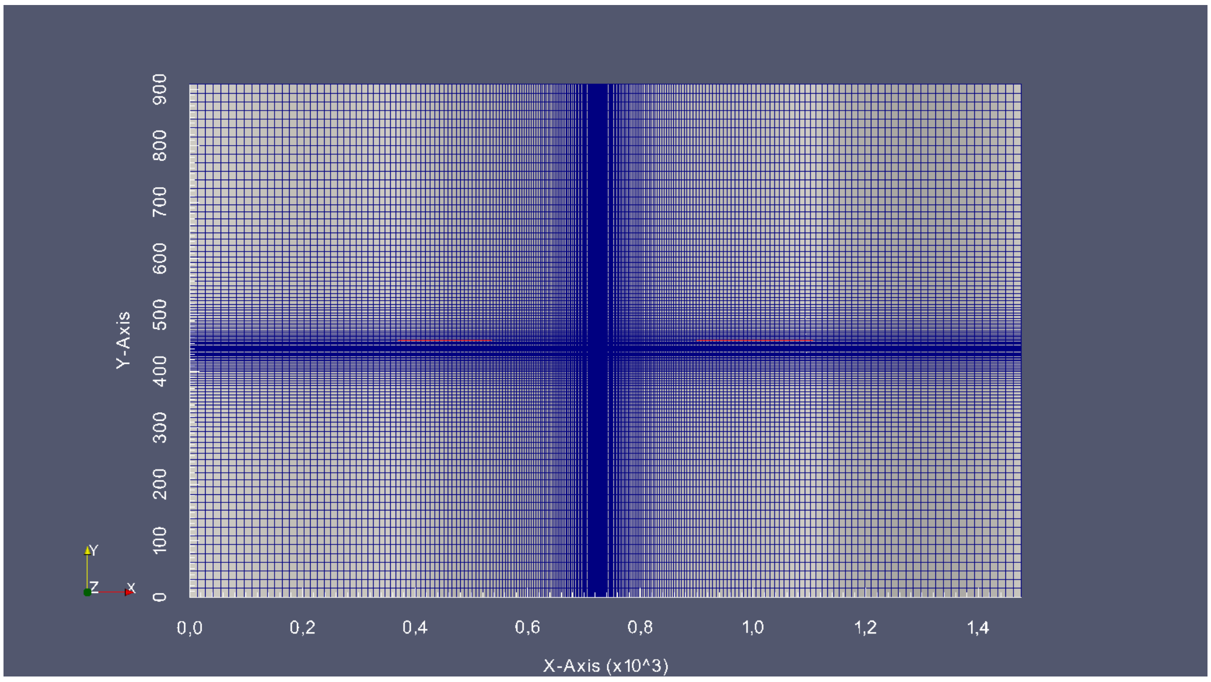


Figure 10.6: The Non-Uniform grid

Following are the parameters used in the simulation

- Lx = 1476 m, Ly = 908 m
- Nx = 240, Ny =120
- Ux = Uy = 1m/s

- h=50m
- g=9.81 m/s^2
- double precision for computation

The maximum grid spacing in the x-direction is 13.8m, and the mimimum grid spacing is 1.37m resulting in $\frac{\Delta x_{max}}{\Delta x_{min}} = 10$. Same ratio is applied on the grid spacing in the y-direction. Δt_{CFL} based on the minimum grid spacing is 0.031s. Our aim is to carry out the simulations for various Δt's and observe the required computation time, and the number of iterations of the Bi-CGSTAB algorithm. Also important is to reflect upon the accuracy and stability of the results obtained.

First we show the results for $\Delta t = 0.5s$, which gives a $\frac{\Delta t}{\Delta t_{CFL}} = 15$.[1]

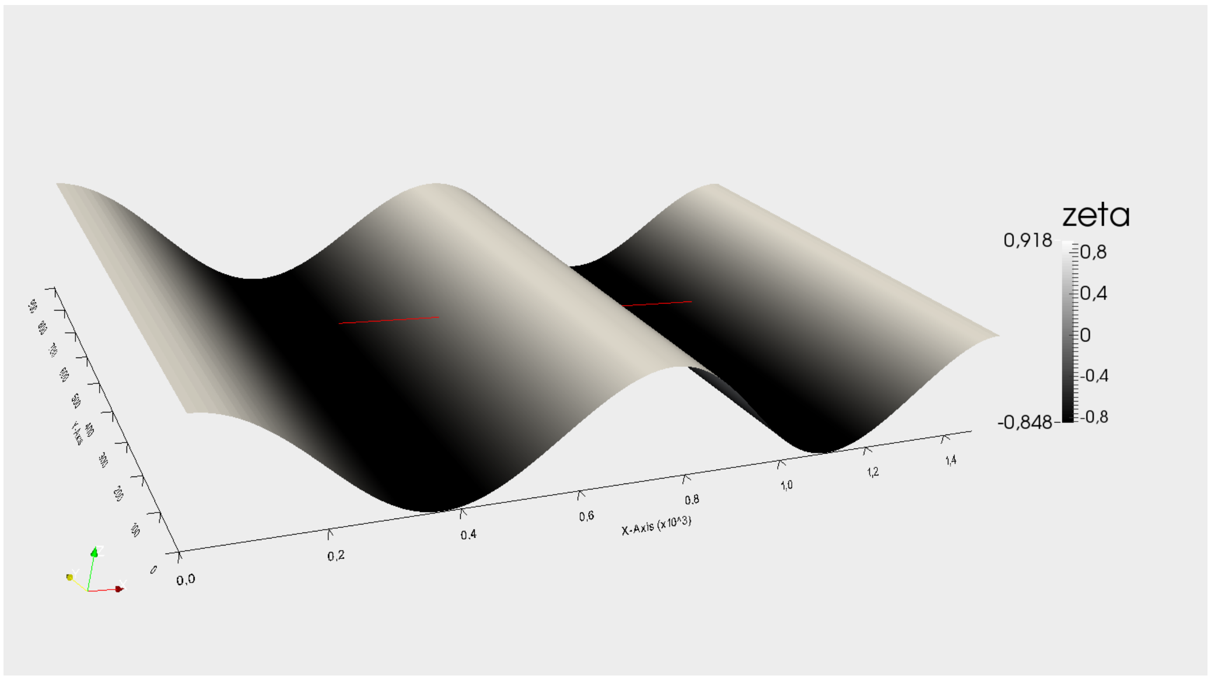


Figure 10.7: Initial conditions for ζ for the Non-Uniform grid

[1] The snapshots of the results are generated at various time steps with the help of Paraview which is an open-source visualization software.

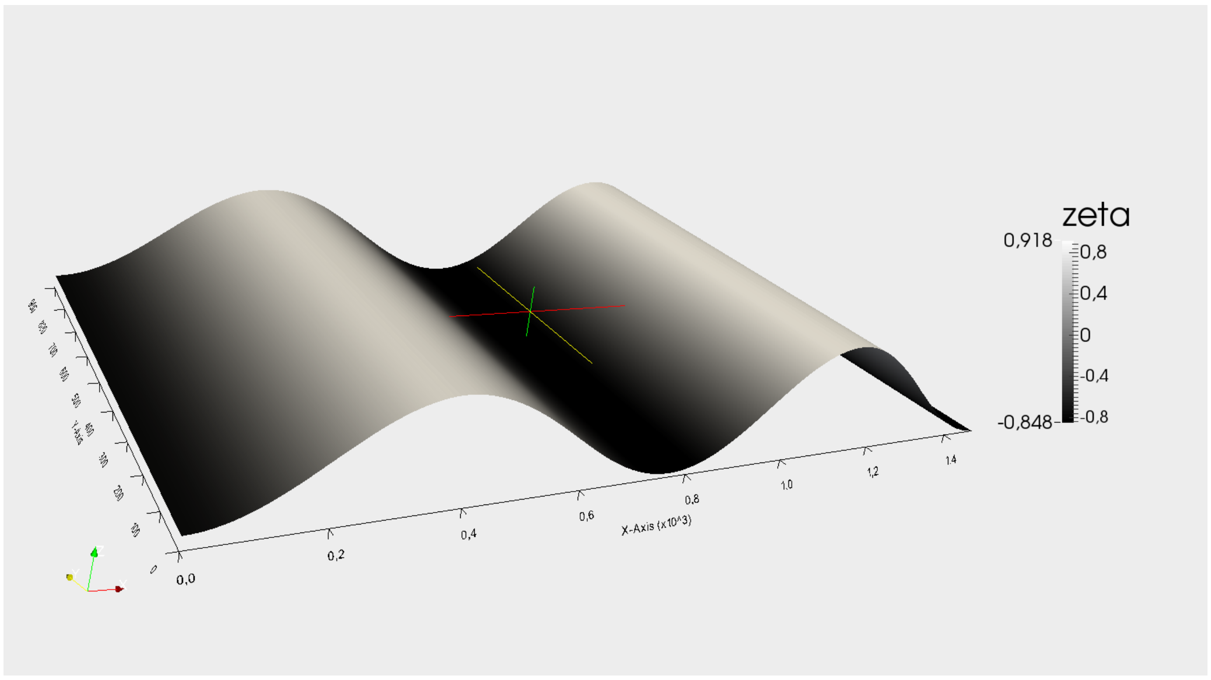


Figure 10.8: ζ at t=50s for the Non-Uniform grid

From Figure 10.7 and Figure 10.8, we can see that the algorithm developed is stable for the Non-Uniform grid, even at large CFL ratios.

We will now compare the results with that of CFL number corresponding to the $\Delta t_{CFL} = 0.031s$. We obtain the result by cutting a slice normal to the y-direction, thus allowing us to compare the results in an one-dimensional plot. The results are shown in Figure 10.9. We can see that the numerical dispersion is very small! This could be attributed to the fact that we are operating at very low $k\Delta x$ ratios. For the smallest grid spacing of 1.37m, the ratio $k\Delta x = \frac{4\pi}{L}\Delta x = 0.01$. As CFL number is inversely propotional to the spatial step size, this also correspond to the region in the grid with highest CFL ratio. For the largest grid spacing, $k\Delta x = 0.1$, but CFL ratio for such spacing is 10 times lower than the CFL ratio for the smallest grid size. Thus we are always near $c/c_0 = 1$, i.e. the top of the curve as shown in Figure 10.1.

This is in contrast to the previous case of a Uniform grid, when with larger CFL number we encountered large dispersion errors.

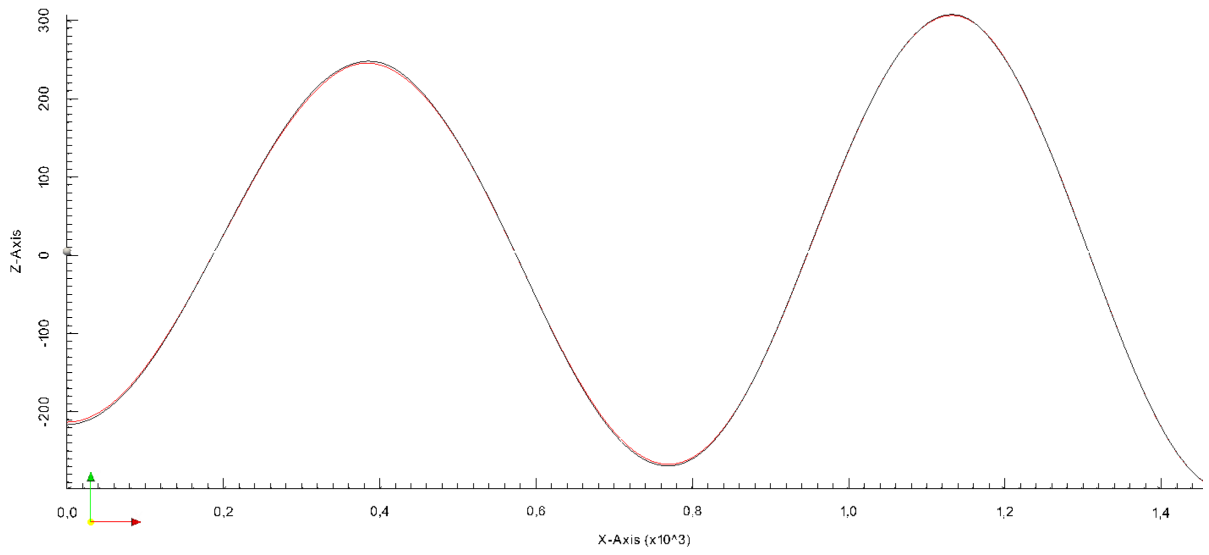


Figure 10.9: ζ at t=50s with $\Delta t = 0.031s$ and $\Delta t = 0.5s$

We now show the timing results for the solver. The simulations are run for 50 seconds in the real time. The implicit Bi-CGSTAB solver is implemented both in C++ and CUDA, whereas the CG solver is implemeted in CUDA.

Table 10.1: Timing results for the Non-Uniform Grid case with C++ Bi-CGSTAB solver, $\Delta t_{CFL} = 0.062s$, $\Delta x_{max}/\Delta x_{min} = 10$

Δt	Avg. No. of Iterations per time step	Run time of simulation
0.031	2	37.56
0.062	2	19.13
0.124	4	11.41
0.248	6	6.08
0.496	14	5.09
0.992	29	4.11

Table 10.2: Timing results for the Non-Uniform Grid case with CUDA implicit Solver, $\Delta t_{CFL} = 0.062s$, $\Delta x_{max}/\Delta x_{min} = 10$

Δt	Avg. No. of Iterations per time step	Run time of simulation
0.031	2	34.36
0.062	2	18.39
0.124	4	10.43
0.248	8	6.2
0.496	20	4.57
0.992	36	3.18

Following are the observations from above results:

- An explicit method here would require $\Delta t = 0.031s$ for the simulation due to CFL number limitation. With the implicit solver, this simulation could be carried out at larger time steps. The required frame per second is 20, which results in a Δt of 0.05 seconds. Performing the simiulation at 0.05 seconds would only require 2 iterations for the convergence of the Bi-CGSTAB algorithm and will save time required in double computation of the CG solver, computing the pulses etc (if the explicit solver is used). Also, in the case of multiple ships where the non-uniform grids are required, the pressure pulse computation can become a time consuming step.
- As observed in the Uniform grid case, CUDA implemtation of the Bi-CGSTAB solver requires more number of iterations to converge than the C++ solver. Part of the problem lies in the r1r2b1b2 storage structure in CUDA and its impact on the sparse matrix vector multiplication and the preconditioning solve function. Both of them need to be studied carefully, and constitute part of the future work.

This concludes our discussion on the non-uniform test case. We have been able to use the solvers developed for the uniform grid with suitable spatial discretization and stencil formulation. Much work still need to be carried out to study the accuracy and speed-up of these methods and their adaptation for the real case. We will now present our conclusions and propose the future work.

Chapter 11

Conclusion

We started with the efficient implementation of the explicit CUDA RRB solver, and intended to develop a robust implicit solver which is based on the similar RRB framework and can overcome the numerical limitation of the explicit schemes. We were able to achieve the following:

- Analyzed various time integration schemes, studied their stability and accuracy behaviour with the simplified test-problems in MATLAB. The Implicit Trapezoidal method came out as the front runner with suitable Symplectic and stability properties for the VBM model.
- Chorin's projection scheme has been developed to solve the coupled system of equations arising from the VBM model. By splitting the operations carefully, stability of the method has been ensured.
- Suitable physics based preconditioners which can take care of the weak-coupling of the system have been formulated and their performance studied . The preconditioners were studied analytically through the eigenvalue analysis and then by performing numerical experiments. The Gauss-Seidel block preconditioner with the gravitational constant as the coupling term provides the optimum number of iterations amongst various preconditioners.
- In order to develop the CUDA and C++ solver in an object oriented way, and to perform benchmarking, a two-dimensional test problem has been formulated. Object-oriented approach allows to plug-in various solvers and make use of the efficient solver developed by Martijn. Various profilers were used to optimize the code, which can be then directly incorporated in the real solver.
- A careful implementation of the Incoming boundary waves allowed a stable solution for the real test case. The Implicit solver for the real test cases outperforms the Explicit solver in certain scenarios. A speed-up factor upto 7/5 was observed.
- A frame-work for the further development of the non-uniform grid solver has been provided. A suitable spatial discretization has been evaluated, resulting in system matrices which can be handled by the RRB solvers.

Chapter 12

Future Work

We would like to recommend MARIN the following regarding the Interactive Waves project. The recommendations also indicate topics for the future research.

- Optimize the CUDA implementation of the implicit solver.
  - Investigate the role of initial guess on the convergence of the Bi-CGSTAB method. It is possible to get a better estimate of the initial guess by linear interpolation of results from previous time steps.
  - Investigate the difference in CUDA and C++ implementation of the Bi-CGSTAB solver. It is possible to reduce the number of iterations required for convergence of the CUDA implicit solver further.
  - Get the CUDA implicit sovler working with single precision floating numbers.
- Implement the weakly-reflective boundary conditions for the implicit solver routine. In current thesis, only the reflective (Neumann) boundary conditions have been implemented.
- Build up-on the non-uniform two-dimensional test case. It is also possible to develop a non-rectilinear grid and reduce the number of grid points further. In order to track the ship movement, a moving grid would be required. This would also require recomputing system matrices and reconstructing the preconditioner at subsequent timesteps. The implicit solver preconditioner construction step should be optimized further through efficient memory utilizations and using C++ standard library.
- Move the pulse computation on GPU. As we observed for the Plymouth Sound test case of 1.5 million nodes, this becomes a time consuming step. This will also become more important in the case of multiple ships.
- The ship simulator supplied by MARIN runs on a distributed system. A strategy needs to be developed which could either implement the current solver on distributed system or does the data transfer to the different compute nodes efficiently.

References

[1] Jong, M. de, Developing a CUDA solver for large sparse matrices for MARIN, MSc Thesis Applied Mathematics, Delft University of Technology, Delft, February 2012.

[2] Klopman, G., Variational Boussinesq Modelling of Surface Gravity Waves over Bathymetry, Phd Thesis, University of Twente, Twente, May 2010.

[3] Wout, E. van 't, Improving the Linear Solver used in the Interactive Wave Model of a Real-time Ship Simulator, MSc Thesis Applied Mathematics, Delft University of Technology, Delft, August 2009.

[4] Keppens R., Tòth G., Botchev M. A, and Ploeg A. van der, Implicit and semi-implicit schemes: algorithms, International Journal for Numerical Methods in Fluids, 30, 335-352, 1999.

[5] Botchev M. A., Sleijpen G. L. G. and Vorst H. A. van der, Low-dimensional Krylov subspace iterations for enhancing stability of time-step integration schemes, Preprint 1004, Dept. of Mathematics, Utrecht University, the Netherlands, 1997.

[6] Trefethen L.N, Bau D. III , Numerical Linear Algebra , Society for Industrial and Applied, Philadelhpia, 1997

[7] Hundsdorfer W. , Verwer J.G., Numerical Solution of Time-Dependent Advection-Diffusion Reaction Equations, Springer, Berlin, 2003

[8] Chorin, A. J., Numerical Solution of the Navier-Stokes Equations, Math. Comp. 22, 745â762, 1968

[9] Fringer O. [PDF Document].Retrieved from Lecture Notes Online Web site: http://web.stanford.edu/ fringer/teaching/cee262c/solution4.pdf