

A Julia-Based Parallel FEM Solution to the Shallow Water Equations

by

Bingjie Xiong

A Thesis submitted to the Faculty of Graduate Studies of

The University of Manitoba

in partial fulfillment of the requirements of the degree of

Master of Science

Department of Electrical and Computer Engineering

University of Manitoba

Winnipeg

June 2025

Copyright ©Bingjie Xiong

Abstract

This thesis presents A two-dimensional finite element method (FEM) solver for the shallow water equations, implemented in the Julia programming language, is presented. The numerical design of the solver is inspired by the CRISSP2D model, which was originally written in Fortran decades ago. However, its original code-base is difficult to maintain and not well suited for modern parallel computing. To address these limitations, the solver presented in this work is constructed from the ground up with independent data structures, algorithms, and parallelization strategies. The implementation solves the depth-averaged shallow water equations using a Petrov-Galerkin testing method with linear triangular elements and leapfrog time integration, and it supports dynamic wetting and drying processes. Particular improvements are made in the treatment of boundary conditions through integration by parts applied to the weak form of the formulation.

The implementation supports both distributed-memory and shared-memory parallelism to improve computational performance. Julia’s flexible parallel computing capabilities provide a strong foundation for modern hydrodynamic modeling. The solver is then validated through a series of staged test cases involving subcritical river flow in a trapezoidal channel, designed to evaluate its ability to handle transitions between wet and dry states. Quantitative comparisons with results from the original Fortran code and HEC-RAS software show strong agreement in key hydrodynamic properties. This work demonstrates that Julia provides an efficient and expressive platform for hydrodynamic model development and offers a modern, extensible alternative for future research and application.

Acknowledgements

I would like to thank Prof. Ian and Prof. Shawn for their contributions to this master's thesis.

I also thank my parents for their patience, understanding, support, and care.

I gratefully acknowledge Manitoba Hydro for their funding support and assistance with CRISSP2D-related questions, with special thanks to Dr. Wang and Dr. Jarrod.

Contents

Acknowledgements	iii
List of Figures	viii
1. Introduction	1
1.1. Purpose	3
1.2. Scope	5
1.3. Outline	6
2. Mathematical background	7
2.1. Principles of Depth-Integrated 2D Hydrodynamic Model	7
2.1.1. Flow Variables	7
2.1.2. Hydrodynamic Equations	9
2.2. Boundary and Initial Conditions	11
2.2.1. Boundary Conditions	12
2.2.2. Initial Conditions	14
2.3. Summary	15
3. Simulation of Hydrodynamic Model using Finite Element Method	17
3.1. Introduction to the Finite Element Method	17
3.1.1. Steps in the Finite Element Method	19
3.2. Discretization of Flow Domain and Basis Function Selection	19
3.3. Finite Element Formulations with Petrov-Galerkin Weighted Residual Method	27

3.3.1.	Introduction of Galerkin Weighted Residual Method	27
3.3.2.	CRISSP2D with Galerkin Weighted Residual Method	28
3.4.	Boundary Conditions	49
3.4.1.	Choice of Equations for Boundary Conditions	50
3.4.2.	Flux Boundary Condition via the Continuity Equation	51
3.4.3.	Elevation Boundary Condition with Momentum Equations	55
3.5.	System of ODEs	55
3.6.	Solution Scheme	58
3.6.1.	Leap frog Explicit Time Integration Scheme	59
3.6.2.	Serial Algorithm	61
4.	Test Cases	63
4.1.	Subcritical Flow In a Rectangular River Channel	64
4.1.1.	Cold Start	65
4.1.2.	Hot Start	73
4.2.	Subcritical Flow in a Trapezoidal River Channel	79
4.2.1.	Boundary Conditions	82
4.2.2.	Simulation Result by Stage	83
4.3.	Test Case Summary	92
5.	FEM Parallelization	93
5.1.	Parallelization Prerequisites	93
5.2.	Share Memory Implementation	94
5.3.	Shared Memory Parallel Strategy	95

5.4. Shared Memory Performance Analysis	97
5.5. Distributed Parallel Implementation	97
5.6. Distributed Parallelize Strategy	98
5.6.1. Domain and Work Load Decomposition	100
5.6.2. Parallel Algorithm	106
5.6.3. Julia Implementation	108
5.6.4. Performance Analysis	127
5.7. Distributed Computing Parallel Speedup and Test Case Description	128
6. Conclusion	134
6.1. Summary of Accomplishments	134
6.2. Future Work	135
Appendix A: Rewriting the Hydrodynamic Equation in Conservative Form to Align with the Non-Conservative Structure	137
Appendix B: Rewriting the Hydrodynamic Equations to Explicitly Determine Com- ponents in the Non-Conservative Form	139
Appendix C: Detail Result for Matrix Multiplication in the Non-Conservative Form .	141
Appendix D: Applying the Divergence Theorem to the Continuity Equation with Galerkin Testing	143
Appendix E: Overview of METIS and SimplePartition Algorithms	145
E.1. SimplePartition Plugin	145
E.2. METIS	146
Appendix F: METIS Partitioning Techniques: GUI and GEO File Methods	148
F.1. Graphical User Interface (GUI) Method	148

F.2. geo File-based Method	148
Appendix G: Performance Analysis for alternative Distributed Parallel Approach . .	150
Appendix H: Comparative Analysis of HEC-RAS, Julia, and Fortran Results	151
H.1. Cold start case	151
H.2. Hot start case	152
Bibliography	154

List of Figures

<u>Figure</u>	<u>Page</u>
2.1. Plane and profile view of a generic river configuration	8
2.2. Boundary conditions in plan-view of the 2D river	12
3.1. Basis functions ϕ_j for linear triangle elements	21
3.2. Visualization of the river mesh using Gmsh.	24
3.3. Node information extracted from the SMS .geo file.	25
3.4. Element information extracted from the SMS .geo file.	25
3.5. Reach information extracted from the SMS .geo file.	25
3.6. Boundary information extracted from the SMS .geo file.	26
4.1. 3D rectangular cross geometric river model: structural details	64
4.2. Rectangular mesh of the computational river domain	65
4.3. Elevation boundary condition over 13 hours.	66
4.4. Flux boundary condition over 13 hours.	67
4.5. Water depth through the centerline of the river.	68
4.6. Relative error of water depth	69
4.7. Water depth difference between Julia and CRISSP2D	70
4.8. x -Direction velocity through the centerline of the river.	71
4.9. Relative error of velocity	72
4.10. Difference in Velocity between Julia and Fortran	73
4.11. Elevation boundary condition over 12 hours.	74

4.12. Flux boundary condition over 12 hours.	74
4.13. Water depth through the centerline of the river.	75
4.14. Relative error of water depth in hot start	76
4.15. Difference in water depth between Julia and Fortran	77
4.16. x -direction velocity through the centerline of the river.	77
4.17. Relative error of velocity	78
4.18. Difference in velocity between Julia and Fortran	79
4.19. 3D rectangular cross sectional river model: Geometric details.	80
4.20. Trapezoidal mesh of the river	81
4.21. Elevation boundary condition over 29 hours.	82
4.22. Flux boundary condition over 29 hours.	82
4.23. Expected downstream cross-sectional profiles	83
4.24. 2D water depth distribution at hour 14 during stage 1.	84
4.25. 2D velocity vector field at hour 14 during stage 1.	85
4.26. Total discharge along the river centerline at hour 14 during stage 1.	86
4.27. 2D water depth distribution at hour 23 during stage 2.	87
4.28. 2D velocity vector field at hour 23 during stage 2.	88
4.29. Total discharge along the river centerline at hour 23 during stage 2.	89
4.30. 2D water depth distribution at hour 29 during stage 3.	90
4.31. 2D velocity vector field at hour 29 during stage 3.	91
4.32. Total discharge along the river centerline at hour 29 during stage 3.	91
5.1. Partitioning of a 2D mesh using Gmsh.	100

5.2. Partitioning of the computational domain into three subdomains	101
5.3. Ghost mesh representation for each partition.	103
5.4. Data exchange process between partitions	105
5.5. Data exchange process using remote channels	117
5.6. Comparison between parallel and serial solutions of p	129
5.7. Comparison between parallel and serial solutions for h	129
5.8. Parallel speedup. An ideal scaling curve is included for reference.	130
5.9. Parallel speedup. An ideal scaling curve is included for reference.	132
5.10. Memory efficiency in parallel implementation	133
F.1. Gmsh GUI.	148
G.1. Parallel Speedup. An ideal scaling curve is included for reference.	150
H.1. Logarithmic relative error of water depth h , along the river centerline. Results from both the Julia and Fortran implementations are compared against HEC-RAS.	151
H.2. Logarithmic relative error of x -direction velocity, along the river streamline. Re- sults from both the Julia and Fortran implementations are compared against HEC-RAS.	152
H.3. Logarithmic relative error of water depth h , along the river centerline. Results from both the Julia and Fortran implementations are compared against HEC-RAS.	152
H.4. Logarithmic relative error of x -direction water velocity, along the river centerline. Results from both the Julia and Fortran implementations are compared against HEC-RAS.	153

1. Introduction

Open-channel flow problems play a critical role in hydraulic engineering and underpin a wide range of practical applications. For example, these problems are essential to understanding and predicting phenomena such as flood propagation, surge behavior in compound channels, and potential risks associated with dam failures [1]. Furthermore, understanding flow characteristics is important for analyzing processes influenced by flow dynamics, including water quality, sediment transport, aquatic habitat preservation, and the evolution and transport of ice formation [2]. Advancements in both computational hardware and numerical modeling techniques have made solving depth-averaged flow models more feasible, enabling engineers and researchers to address these challenges with greater efficiency and precision.

The dynamics of fluid flow phenomena and hydraulic problems are commonly described by the shallow water hydrodynamic equations, which are derived from the fundamental principles of conservation laws. The 2D shallow water equations are derived from the full Navier–Stokes equations under the assumptions of shallow flow, hydrostatic pressure distribution, and negligible velocity variation with water depth. Unlike the general Navier–Stokes equations, which are highly complex and generally impractical to solve analytically for arbitrary problem configurations, the shallow water equations provide a simplified framework. Analytical solutions for either system are only available in specific cases involving simple geometries and boundary conditions. As a result, numerical methods are essential for approximating solutions to these problems. Early developments in numerical approximation began with the finite difference method (FDM), which was widely applied during the 1960s and 1970s [3]. Later, advancements in numerical methods and computa-

tional power led to the widespread adoption of the finite volume method (FVM) and finite element method (FEM), which offer greater flexibility and robustness for solving complex flow problems [3]. The specific advantages of these methods, including their adaptability and accuracy, are explored in the following paragraphs.

The main challenges in solving two-dimensional shallow water equations numerically include accurately simulating shocks, minimizing or eliminating oscillations in the solution, handling domains with complex geometries, and addressing complex boundary conditions. Additionally, the wetting and drying regions during the simulation presents unique challenges that require specialized treatments [2]. When addressing these issues, both FDM and FVM have certain limitations compared to FEM.

FDM, with techniques like flux-split operators and the Alternating Direction Implicit (ADI) method [4], is less suited for scenarios involving rapid changes or highly nonlinear flow velocities [2]. its reliance on structured meshes limits its ability to handle irregular geometries. Furthermore, FDM often requires the introduction of imaginary nodes to apply boundary conditions, which can compromise solution precision at boundaries [2].

FVM, with flux-difference splitting schemes, benefits from its compatibility with unstructured meshes, which allows for greater flexibility in handling complex geometries. It also has some tolerance for discontinuities within the solution domain. However, it typically requires more nodes and longer simulation times to achieve the same level of accuracy as FEM [2].

In contrast to the limitations of FDM and FVM, FEM offers several advantages that make it a suitable choice for shallow water problem. FEM provides a unified computational framework that can handle both supercritical and subcritical flows using the same algorithm [2]. It is flexible and can be extended to higher-dimensional geometries and

higher-order solutions. The method is well-suited for irregular geometry domains by using unstructured meshes. Additionally, FEM naturally incorporates boundary conditions derived from the integral by parts formulation, allowing for accurate and straightforward treatment of irregular stream boundaries. Unwanted oscillations in the solution can be reduced by using upwind-weighted test functions, such as the characteristic dissipative Galerkin (CDG) scheme [1]. These advantages make FEM an efficient and reliable approach for solving the two-dimensional shallow water equations [2].

1.1. Purpose

While a number of numerical tools exist for shallow water modeling in one and two dimensions, such as HEC-RAS [5], River1D [6], and MIKE-11 [7] for 1D, and River2D [6], MIKE-21 [8], SMS [9], and Telemac [10] for 2D, most of them do not provide comprehensive support for two-dimensional river ice processes. This thesis is motivated by CRISSP2D [11], an existing numerical engine originally developed based on a mathematical model of river ice processes [12]. The original implementation, written in Fortran, has several limitations, including limited flexibility, inefficiencies in algorithmic performance, difficulties in extending to higher-order solutions and complex geometries, and inadequate support for parallelization. Many of these issues are due to the implementation being decades old with limited documentation, which makes modularization and modernization of the codebase difficult.

To address the challenges inherent in efficiently simulating two-dimensional shallow water equations, this thesis implements and evaluates the model in Julia. Julia is a modern programming language specifically designed to deliver high performance for technical and scientific computing. Compared to widely used general-purpose languages such as Python and C++, Julia offers several advantages:

1. High performance: Julia uses run time compilation to achieve execution speeds comparable to C and Fortran. According to benchmarks published on the Julia website, which compare identical algorithms across languages, Julia consistently outperforms Python and is competitive with C and Fortran [13].
2. Parallel Computing Support: Julia is designed for parallelism, with built-in modules for multi-threading (Threads) and distributed computing (Distributed) included in the standard library. GPU computing is supported via official packages such as CUDA.jl. For more advanced parallel computing models—such as MPI or OpenMP-style communication—Julia relies on external packages (e.g., MPI.jl), similar to C++ and Python. However, Julia’s standard ecosystem makes it straightforward to develop and scale parallel programs across different environments [13].
3. Focus on Scientific Computing: Unlike general-purpose languages, Julia’s core libraries and ecosystem are heavily oriented toward numerical analysis, simulation, and scientific computing. Its package ecosystem includes high-quality tools for linear algebra, optimization, differential equations, and data analysis. This focus, combined with Julia’s ease of use and performance, makes it especially suitable for research and development in computational science [13].

In general, further model development on the Fortran CRISSP2D code has been inhibited by the steep learning curve associated with the legacy Fortran code. Adopting a modern language like Julia is expected to accelerate the implementation of new functionality, enabling faster adaptation of the model as understanding of river ice processes advances.

The new implementation aims to improve flexibility, optimize algorithmic efficiency, and support parallel computing in both shared-memory and distributed computing environ-

ments. Additionally, it provides better extensibility for higher-order solutions and complex geometries through the integration of numerical related software packages such as Gmsh. Due to Julia’s high performance capabilities, the new model can handle river meshes with more elements than the original CRISSP2D program, which was limited to approximately 60,000 elements. Julia’s performance thus far indicates enhanced scalability for larger and more complex simulations.

1.2. Scope

Although the CRISSP2D software simulates hydrodynamic, thermal, and ice dynamics processes, the current Julia implementation is limited to the hydrodynamic model, which represents an important foundational step in the broader simulation of river ice processes. This work establishes the foundation for future extensions to include thermodynamic and ice-dynamics simulations. The main focus of this work is on implementing and evaluating parallel processing capabilities using the Julia framework. Unlike the existing CRISSP2D code, which lacks parallel processing, this improvement significantly accelerates the solution of hydrodynamic equations. Attempts by previous students to accelerate the CRISSP2D Fortran code directly lead to no computational improvement due to constraints arising from the implementation. The Julia solver’s parallel framework is a major contribution to two-dimensional river ice modeling with shallow-water equations, offering better computational efficiency and scalability.

This thesis does not focus on the detailed derivation of formulas for hydraulic behavior, but instead assumes them and focuses on their numerical solution. The numerical solver is limited to using a FEM-formulation with Petrov-Galerkin testing to solve two-dimensional hydrodynamic equations for depth-averaged shallow water problems in open channels.

The approximation functions employed in the FEM are first-order Lagrange interpolating polynomials. While higher-order polynomials could improve accuracy, they are not explored in this work, though the extension to high-order elements is enabled through the use of the Gmsh API. Similarly, linear triangular elements are used for mesh discretization, although higher-order elements could better fit curved geometries and yield more accurate results.

Parallel computing strategies implemented in this thesis are limited to shared-memory (using OpenMP) and distributed computing environments (using MPI). While GPU computing offers significant potential for improved performance, such co-processor acceleration remains a topic for future investigation.

1.3. Outline

The remainder of the thesis divided into three main parts.

1. Mathematical Foundations: Chapter 2 and 3 present the necessary mathematical theory to derive the final system of ordinary differential equations (ODEs), which are solved numerically using the leapfrog time integration scheme.
2. Validation and Test Results: Chapter 4 focuses on validating the FEM numerical engine by analyzing test results from different river models under various testing scenarios.
3. Parallel Strategies and Implementation: Chapter 5 compares different parallel computing strategies, evaluates their performance, and provides details on the software implementation.

Finally, Chapter 6 concludes the thesis and outlines possible continuations for the development and use of the solver.

2. Mathematical background

This Chapter introduces the shallow water hydrodynamic equations, which are used to simulate free surface shallow water flows under the influence of gravity, from upstream to downstream. The flow variables, which are unknown functions that vary with space and time, are governed by a system of partial differential equations (PDEs), commonly referred to as the shallow water hydrodynamic equations. In addition, boundary and initial conditions (BCs and ICs) are used to further constrain the flow variables. Together, the PDEs, the unknown flow variables, and the boundary and initial conditions form the essential components of the hydrodynamic initial-boundary value problem (IBVP).

2.1. Principles of Depth-Integrated 2D Hydrodynamic Model

This subsection provides the physical and mathematical background of the depth-integrated 2D hydrodynamic model, including key governing equations.

2.1.1. Flow Variables

In a 2D depth-integrated hydrodynamic problem, the variables that describe fluid flow include water depth, water velocity, and the scalar component of water discharge in the x and y directions. These flow variables are unknown functions of time t and position $\vec{r}$, where $\vec{r} = [x, y]$, defined over the domain Ω . Here $\Omega \subset \mathbb{R}^2$ represents the x, y plane, and $\vec{r} \in \Omega$. The boundary of the domain Ω is denoted as Γ , which will be referenced later in relation to boundary conditions.

Definitions of the Flow Variables

The flow variables in the hydrodynamic model are as follows:

- water surface elevation $\eta(\vec{r}, t)$ [m]: the vertical distance from the user-defined reference horizontal plane to the water surface, as depicted in Figure 2.1. Note that the value of η at a position is positive when the water surface is above the reference plane and negative when it is below. Additionally, Z_0 [m] represents the vertical distance from the reference horizontal plane to the riverbed, as depicted in Figure 2.1. Similar to η , Z_0 is positive when the channel bed is above the reference plane and negative when it is below.

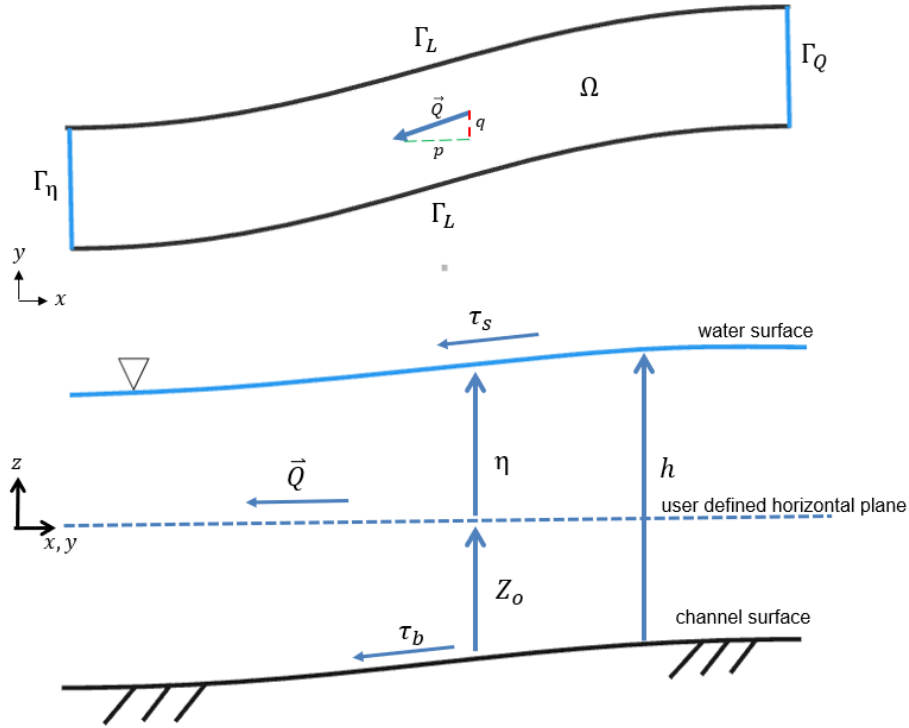


Figure 2.1. Plane and profile view of a generic river configuration. Plane view: The river domain Ω is bounded by land boundary Γ_L , elevation boundary Γ_η , and flux boundary Γ_Q . The total flow vector $\vec{Q}$ is decomposed into components along x and y directions. Profile view: Vertical variables are defined relative to a user-defined horizontal plane, with values increasing in the direction of the arrow. Shear stresses at the water surface (τ_s) and bed stress at the channel bed (τ_b) act in the horizontal direction.

- water depth $h(\vec{r}, t) = \eta(\vec{r}, t) - Z_0(\vec{r})$ [m]: h is the vertical distance from the river bed to the water surface.
- water discharge $p(\vec{r}, t)$ [m²/s]: volumetric flow rate per unit width in the x direction.
- water discharge $q(\vec{r}, t)$ [m²/s]: volumetric flow rate per unit width in the y direction.
- water velocity $u(\vec{r}, t) = p(\vec{r}, t)/h(\vec{r}, t)$ [m/s]: water velocity in the x direction.
- water velocity $v(\vec{r}, t) = q(\vec{r}, t)/h(\vec{r}, t)$ [m/s]: water velocity in the y direction.

The quantities η, p, q, h, u and v are functions of space and time used to describe the CRISSP2D hydrodynamic model. However, η, u , and v are derived from h, p and q , making h, p and q the primary key variables directly solved by the model.

2.1.2. Hydrodynamic Equations

The hydrodynamic equations governing h, p and q are foundational to solving two-dimensional, depth-integrated hydrodynamic problems¹. These equations include the continuity equation, which is based on the conservation of mass, and the momentum equations, which are derived from the conservation of momentum [14].

- Continuity equation

The continuity equation, based on the conservation of mass, states that the rate of change in water surface elevation over time is equal to the difference between the

¹Given that the hydrodynamic equations used in the CRISSP2D model are well-established with specific underlying assumptions, and since this thesis focuses on the software implementation of the shallow water equations rather than their development, the detailed derivation of these equations is not discussed here. The interested reader is referred to [14]

inflow rate and the outflow rate within a control volume [15].

$$\frac{\partial h}{\partial t} + \frac{\partial p}{\partial x} + \frac{\partial q}{\partial y} = 0 \quad (2.1)$$

- Momentum equations

The momentum equation, based on the principle of momentum conservation, expresses the equilibrium between the various forces acting on the fluid [15].

$$\begin{aligned} \frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(\frac{p^2}{h}\right) + \frac{\partial}{\partial y}\left(\frac{pq}{h}\right) &= \frac{\tau_{sx} - \tau_{bx}}{\rho} - gh \frac{\partial \eta}{\partial x} + \frac{1}{\rho} \left(\frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y} \right) \\ \frac{\partial q}{\partial t} + \frac{\partial}{\partial x}\left(\frac{pq}{h}\right) + \frac{\partial}{\partial y}\left(\frac{q^2}{h}\right) &= \frac{\tau_{sy} - \tau_{by}}{\rho} - gh \frac{\partial \eta}{\partial y} + \frac{1}{\rho} \left(\frac{\partial T_{xy}}{\partial x} + \frac{\partial T_{yy}}{\partial y} \right) \end{aligned} \quad (2.2)$$

in which ρ represents constant water density 1 [g/cm³] and g represents the gravitational constant [m/s²].

The momentum equations (2.2) can be divided into the following components:

– Convection Term:

$$\frac{\partial}{\partial x}\left(\frac{p^2}{h}\right) + \frac{\partial}{\partial y}\left(\frac{pq}{h}\right) \ \& \ \frac{\partial}{\partial x}\left(\frac{pq}{h}\right) + \frac{\partial}{\partial y}\left(\frac{q^2}{h}\right)$$

This term represents the transport of fluid by the flow, illustrating the influence of surrounding fluid particles on the movement of a given particle [3].

– Friction Term:

$$\frac{\tau_{sx} - \tau_{bx}}{\rho} \ \& \ \frac{\tau_{sy} - \tau_{by}}{\rho}$$

This term represents the shear stresses acting on the fluid, where τ_{sx} or τ_{sy} denote the surface shear stresses at the ice-covered water surface in the x or y direction and τ_{bx} or τ_{by} represent the bed shear stresses in the x or y direction.

– Gravity Term:

$$gh \frac{\partial \eta}{\partial x} \& gh \frac{\partial \eta}{\partial y}$$

This term describes the effect of the water surface slope on fluid motion.

– Diffusion Term:

$$\frac{1}{\rho} \left(\frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y} \right) \& \frac{1}{\rho} \left(\frac{\partial T_{xy}}{\partial x} + \frac{\partial T_{yy}}{\partial y} \right)$$

This term represents the damping effect on fluid motion, where the components of the stress tensor are defined as follows:

$$T_{xy} = T_{yx} = \epsilon_{xy} \left(\frac{\partial q}{\partial x} + \frac{\partial q}{\partial y} \right), T_{xx} = 2\epsilon_{xx} \left(\frac{\partial q}{\partial x} \right), T_{yy} = 2\epsilon_{yy} \left(\frac{\partial p}{\partial y} \right)$$

where ϵ_{xx} , ϵ_{xy} and ϵ_{yy} are all generalized eddy viscosity coefficients and are assumed to be 1.

2.2. Boundary and Initial Conditions

To obtain a unique and physically accurate solution to the hydrodynamic equations (2.15), it is necessary to define appropriate boundary and initial conditions. These conditions constrain the solution space and ensure the problem is well-defined, which is necessary before applying any numerical method. In this Section, boundary and initial conditions that support the hydrodynamic solver will be introduced.

2.2.1. Boundary Conditions

In the general context of boundary conditions for ordinary differential equations (ODEs) and partial differential equations (PDEs), three types are typically used: Dirichlet, Neumann, and Robin. Dirichlet conditions specify unknowns on the boundary, Neumann conditions specify derivatives of unknowns on the boundary, and Robin conditions specify a combination of value and derivative [16]. These abstract mathematical boundary conditions are often given specific names in flow problems, such as flux, land, or elevation boundaries. In the CRISSP2D hydrodynamic model, these boundaries are all treated as Dirichlet boundary conditions, where constant values of flow variables such as water surface elevation (η) or flow discharges (p and q) are specified at the boundary nodes.

Note that, the boundary conditions for η , p or q at Γ are homogeneous, indicating consistent values across the same boundary. Figure 2.2 shows a top-down view of our two-dimensional river domain, with each segment of the boundary (Γ_Q , Γ_L and Γ_η) defined by its prescribed Dirichlet condition; we will examine the precise treatment of these boundary types in the sections that follow.

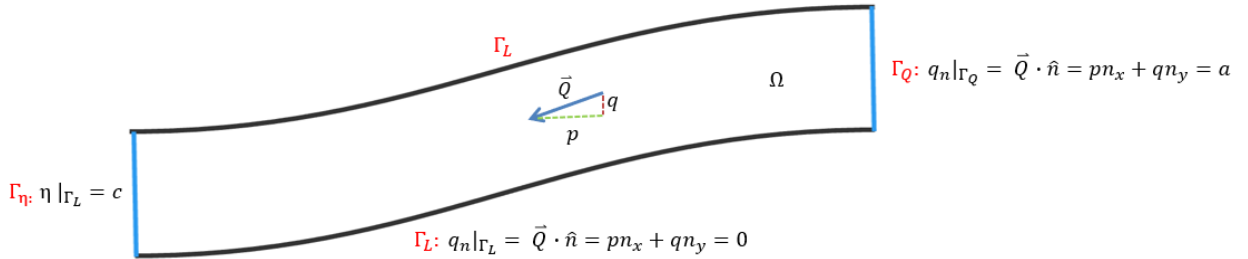


Figure 2.2. Plan-view of the 2D river domain Ω , showing three Dirichlet boundary segments: Γ_Q (flux boundary), Γ_L (land boundary), and Γ_η (elevation boundary)

2.2.1.1. Land Boundary

A land boundary is defined as a boundary where no flux occurs across it, typically set at the interface between land and river [11]. The mathematical representation of this boundary condition corresponds to setting the normal component of the total discharge vector $\vec{Q} = [p, q]$ to 0 along the land boundary Γ_L . This can be expressed as:

$$q_n|_{\Gamma_L} = \vec{Q} \cdot \hat{n} = \begin{bmatrix} p \\ q \end{bmatrix} \cdot \begin{bmatrix} n_x \\ n_y \end{bmatrix} = pn_x + qn_y = 0, t > 0, \quad (2.3)$$

where q_n denotes the scalar discharge in the outward normal direction from the river domain into the land.

2.2.1.2. Flux Boundary

A flux boundary is defined as a boundary across which a specified discharge occurs, typically applied at the upstream or downstream ends of the domain. At this boundary, the normal discharge q_n is prescribed as a constant a , and is given by:

$$q_n|_{\Gamma_Q} = \vec{Q} \cdot \hat{n} = \begin{bmatrix} p \\ q \end{bmatrix} \cdot \begin{bmatrix} n_x \\ n_y \end{bmatrix} = pn_x + qn_y = a, a \in \mathbb{R}, t > 0, \quad (2.4)$$

where a [m^2/s] is the unit width normal flow discharge distribution across the boundary Γ .

2.2.1.3. Elevation Boundary

An elevation boundary condition specifies the water surface elevation at the boundary and is typically applied at the downstream end of the river to constrain the outgoing flow.

It is expressed as:

$$\eta|_{\Gamma_\eta} = c, \quad c \in \mathbb{R}, t > 0, \quad (2.5)$$

where c [m] is the prescribed water surface elevation at the elevation boundary Γ_η .

2.2.2. Initial Conditions

The initial conditions represent the state of the flow at the start of the simulation. They are required for each of the flow variables in the solution domain. Ideally, if the initial conditions of p, q and h can be precisely defined at every position in the solution domain, the subsequent values of these variables can be accurately determined at any point during the simulation.

However, obtaining precise initial conditions (e.g., p, q , or h) for hydrodynamic simulation in real world scenarios, such as a river system, can be challenging. In the CRISSP2D model, the initial conditions are set using either a hot start or cold start approach.

2.2.2.1. Cold Start

The simulation starts from $t = 0$, and the initial conditions for p, q, u , and v are all set to zero, representing no initial flow. The water surface elevation η is initially set to a constant value (η_0) across the entire river domain, ensuring that the channel bed is fully covered with water. Mathematically, a Cold Start is initially imposed according to:

$$\begin{aligned} p, q, u, v|_{t=0} &= 0, \vec{r} \in \Omega \\ \eta(\vec{r}, t)|_{t=0} &= \eta_0, \eta_0 \in \mathbb{R} \\ h(\vec{r}, t)|_{t=0} &= -Z_0(\vec{r}) + \eta_0, \vec{r} \in \Omega, Z_0 \in \mathbb{R} \end{aligned} \quad (2.6)$$

This initial state resembles a stationary lake. Gradually, inflow is introduced at the upstream end of the model, and the downstream water level is dropped. This causes the water depth to decrease and the water velocity to increase throughout the model domain. The simulation is run for a sufficiently long duration for the entire system to reach a steady state.

2.2.2.2. Hot Start

The simulation begins from the final state of a previous simulation (which may or may not be the final time step of a Cold Start simulation), with p, q and h set to the values from that state, regardless of whether the system had reached a steady state. Typically, this initial state represents a river flowing from upstream to downstream with reasonable water velocity and depth at all nodes. This approach uses the previous simulation results to reduce computation time and improve stability. Mathematically, a Hot Start is represented as:

$$h, \eta, p, q, u, v(\vec{r}, t)|_{t=0} = \text{value}|_{t=t_{\max}}^{(\text{previous run})}, \vec{r} \in \Omega \quad (2.7)$$

It is important to note that boundary conditions differ between cold and hot start approaches. In a cold start, the inflow boundary conditions require a gradual increase in water discharge and a corresponding decrease in downstream water depth to achieve a physically meaningful state, allowing the water depth throughout the river to adjust accordingly.

2.3. Summary

In Chapter 2, we established the hydrodynamic equations, initial conditions, and boundary conditions for the 2D depth-integrated hydrodynamic model. With these foundational elements in place, we can now define the initial-boundary value problem (IBVP) for common hydrodynamic scenarios as follows.

Shallow water hydrodynamic equations:

$$\begin{aligned}
\frac{\partial h}{\partial t} + \frac{\partial p}{\partial x} + \frac{\partial q}{\partial y} &= 0 \\
\frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(\frac{p^2}{h}\right) + \frac{\partial}{\partial y}\left(\frac{pq}{h}\right) &= \frac{\tau_{sx} - \tau_{bx}}{\rho} - gh \frac{\partial \eta}{\partial x} + \frac{1}{\rho} \left(\frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y} \right) \\
\frac{\partial q}{\partial t} + \frac{\partial}{\partial x}\left(\frac{pq}{h}\right) + \frac{\partial}{\partial y}\left(\frac{q^2}{h}\right) &= \frac{\tau_{sy} - \tau_{by}}{\rho} - gh \frac{\partial \eta}{\partial y} + \frac{1}{\rho} \left(\frac{\partial T_{xy}}{\partial x} + \frac{\partial T_{yy}}{\partial y} \right)
\end{aligned} \tag{2.8}$$

Boundary Conditions:

$$\begin{aligned}
q_n|_{\Gamma_Q} &= a, t > 0 \\
q_t|_{\Gamma_Q} &= b, t > 0 \\
\eta|_{\Gamma_\eta} &= c, t > 0
\end{aligned} \tag{2.9}$$

Initial Condition:

$$\begin{aligned}
p, q, u, v|_{t=0} &= \text{value}|_{t=t_{\max}}^{(\text{previous run})} \\
\eta|_{t=0} &= \text{value}|_{t=t_{\max}}^{(\text{previous run})} \text{ or } \eta_0 \\
h|_{t=0} &= \text{value}|_{t=t_{\max}}^{(\text{previous run})} \text{ or } \eta_0 - Z_0
\end{aligned} \tag{2.10}$$

3. Simulation of Hydrodynamic Model using Finite Element Method

This Chapter outlines the process of numerically solving the shallow-water hydrodynamic model. The workflow begins with the discretization of the domain, followed by the formulation of the governing system of equations using the Petrov-Galerkin method, and concludes with solving the equations numerically. The software tools supporting this workflow are Julia and Gmsh. Julia is chosen to implement the numerical simulation, while Gmsh provides support for meshing, basis functions, and includes a flexible programming API compatible with Julia and other languages.

3.1. Introduction to the Finite Element Method

The Finite Element Method (FEM) is a numerical technique used to obtain approximate solutions for initial boundary value problems (IBVPs) in various fields [17]. In this work, FEM is applied to numerically solve the hydrodynamic model by approximating the functions h , p and q , such that the continuous equations are converted into a set of discrete equations suitable for computer solution. Generally, this approach begins by discretizing the problem domain Ω into small regular elements (e.g., triangles), making it easier to represent general unknown functions h , p and q . On each element, a simple polynomial expansion, expressed as a sum of coefficients multiplied by basis functions, provides partial flexibility in approximating the function. By combining these partial representations across all elements, the discretization enables the representation of complicated functions.

The CRISSP2D model employs the Lagrange interpolation method to perform the polynomial expansion, representing the unknown function in each element as:

$$\tilde{U}^e(x, y, t) = \sum_{j=1}^n \phi_j^e(x, y) U_j^e(t) \quad (3.1)$$

where

- $\tilde{U}^e(x, y, t)$ is the unknown function in the element
- $U_j^e(t)$ is the value of the unknown function at interpolation node j of the element, also referred to as the degree of freedom (with separate expansions for each of h , p or q in the CRISSP2D model)
- $\phi_j^e(x, y)$ is the Lagrange basis function, also known as the shape function, associated with the interpolation node j
- n is the number of basis functions

After approximating the unknown functions using polynomial expansion, the governing differential equations within each element Ω^e can be expressed in terms of these unknown coefficients. To solve for the coefficients, the Galerkin residual method [18] is employed in the CRISSP2D model, where the governing equations are recast by minimizing weighted residuals. This process results in a system of ordinary differential equations (ODEs) that can be solved numerically. Solving this system, with the appropriate boundary conditions applied, yields the nodal coefficients in the polynomial expansion for each element. By assembling the solutions from all elements, the complex behavior of the unknown functions across the entire domain can be represented, achieving the goal of the FEM approach.

3.1.1. Steps in the Finite Element Method

Based on the previous introduction, the FEM simulation process can be summarized in three key steps:

1. **Discretization and Basis Function Selection:** The continuous domain is divided into smaller elements, and a suitable basis (shape) function is chosen to approximate the unknowns within each element.
2. **Formulate the System of Ordinary Differential Equations(ODEs) with the Characteristic-Dissipative-Galerkin (CDG) Method:** The Galerkin testing procedure is applied to solve the governing differential equations in each element, using the Characteristic-Dissipative-Galerkin method to maintain stability in the non-linear fluid flow equations [11].
3. **Solve the System of ODEs:** Solve the resulting system of ODEs and assemble the solutions to approximate the unknowns over the entire domain.

Each of these steps will be discussed in detail in the following subsections.

3.2. Discretization of Flow Domain and Basis Function Selection

In the continuous domain, an unknown function can only be exactly represented by a basis function expansion with an infinite number of basis functions. To make the problem solvable by a computer, discretization is introduced to approximate the unknown function using a finite set of basis functions. As introduced earlier, the Finite Element Method (FEM) divides the continuous domain Ω into smaller elements Ω^e . Within each element, a finite-dimensional basis function expansion, specifically the Lagrange interpolation method

in CRISSP2D, is used to approximate the unknown function:

$$\tilde{U}^e(x, y, t) = \sum_{j=1}^n \phi_j^e(x, y) U_j^e(t).$$

This approach provides an approximate representation of the solution within each element, allowing the governing hydrodynamic equations to be discretized and computed efficiently.

In CRISSP2D, the basis functions, also known as the shape functions, are chosen as first-order Lagrange interpolating polynomials, represented as:

$$\phi_j^e(x, y) = a_j^e x + b_j^e y + c_j^e \tag{3.2}$$

where a_j^e , b_j^e , and c_j^e are constant that determine or specify the shape function, and where j is the shape function index. Specifically, the shape functions are chosen to satisfy the condition that they take a value of 1 at their corresponding interpolation node and 0 at all other nodes and are complete to some desired polynomial order. While each basis function $\phi_j^e(x, y)$ is locally defined on an element e , they are constructed to be continuous across elements. This continuity is necessary for assembling a global solution. Figure 3.1 illustrates how the basis function associated with node j spans multiple elements that share that node.

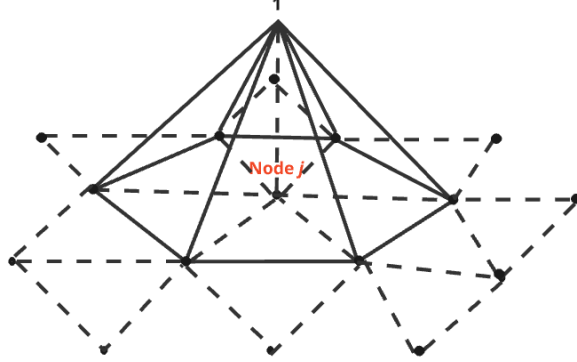


Figure 3.1. Basis functions ϕ_j associated with node j , defined over multiple linear triangle elements sharing that node [17].

The current implementation employs first-order basis functions, but it is possible to improve accuracy by employing higher-order Lagrange basis functions. Gmsh supports these higher-order functions, which could provide more accurate results in the solution. This improvement can be considered for future developments.

The unknown function in the entire domain can be then represented using partial basis function as:

$$\tilde{U}(x, y, t) = \sum_{e=1}^M \sum_{j=1}^n \phi_j^e(x, y) U_j^e(t),$$

where

- M is the total number of elements in the domain,
- n is the number of local basis functions per element,
- $\phi_j^e(x, y)$ is the j -th basis function on element e , which is nonzero only within Ω^e and zero outside.

During the space discretization process, also known as meshing, elements of different geometric orders can be used to capture the curvature of the geometry. While higher-order geometric

representation of elements may improve the accuracy of the simulation, they also increase storage requirements and computational cost. In the CRISSP2D model, two-dimensional, first-order triangular elements were chosen for their simplicity and flexibility in covering irregular domains, making them a suitable compromise between accuracy and computational efficiency

After determining the geometry order, mesh generation or discretization can be performed using various software techniques. This process starts by generating all the grid nodes and then connecting them to form triangular elements [3]. In the CRISSP2D model, grid generation is not performed directly within the model but is supported by externally generated grids, typically created using SMS¹ software. The procedure for meshing the river domain with the help of SMS and Gmsh software involves the following steps:

1. **Generating Grid Nodes:** All the grid nodes are generated within the river domain.
2. **Collecting Riverbed Depth Information:** The riverbed depth data (Z_0) at each node is gathered. This data is either obtained by interpolating polynomial formulas or by direct measurement at the specific grid locations. The Z_0 values are treated as the z -axis values for each grid node. Although the mesh is two-dimensional, with only the x and y coordinates of each node used to generate the triangular elements, the z information at each node is still required to solve the hydrodynamic equations.
3. **Forming Triangular Elements:** The coordinates of all the nodes, along with their Z_0 values, are input into the SMS software. The software then automatically connects the nodes to form triangular elements, covering the entire river domain.

¹Surface-water Modeling System [9]

4. **Generate mesh file:** The mesh information, including both node and element data, is then written into a .geo file. A converter program, written in Julia, was developed to convert the .geo file into a mesh file format compatible with Gmsh. An example of a river mesh generated by SMS software and visualized using Gmsh is shown in Figure 3.2.

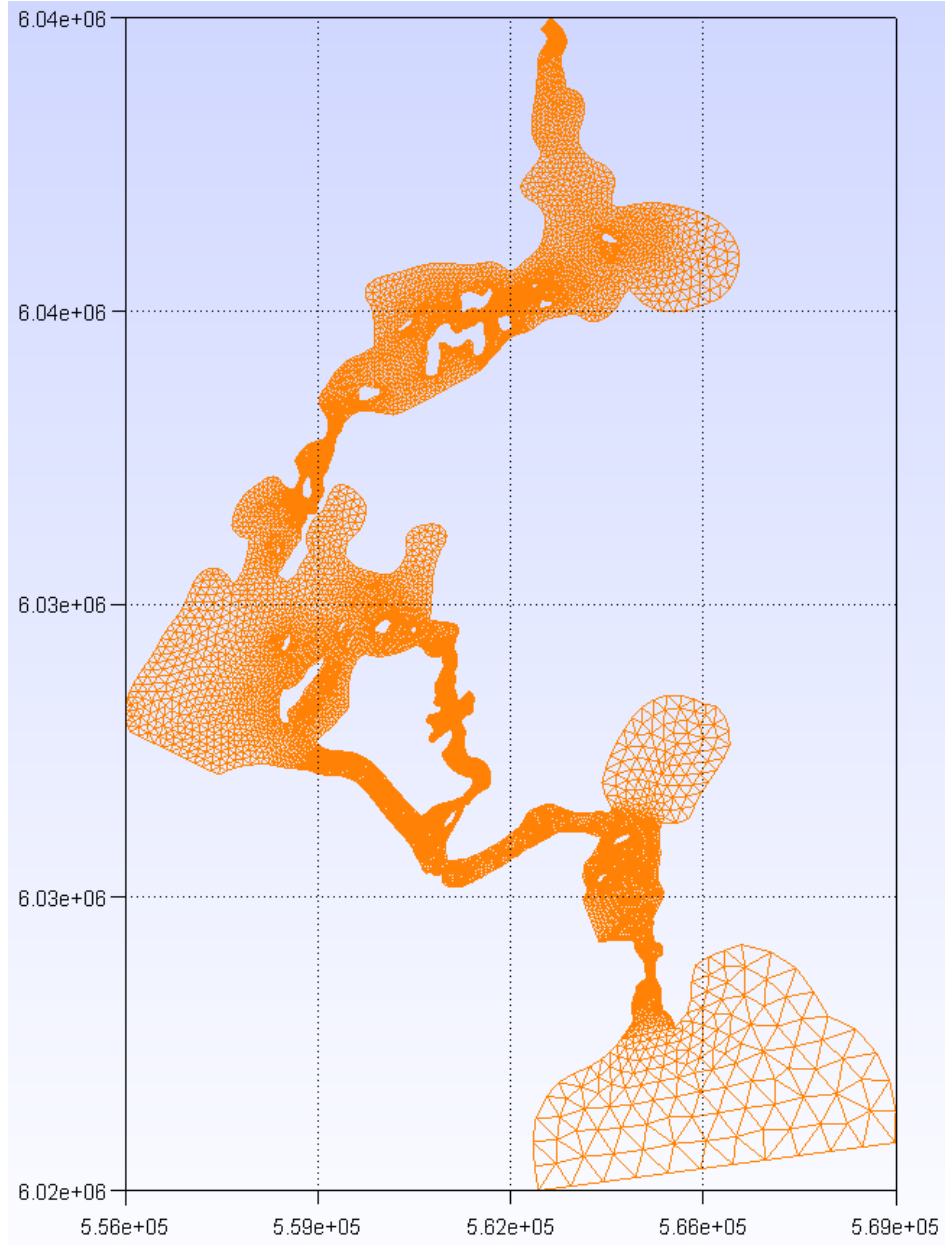


Figure 3.2. Visualization of the river mesh using Gmsh.

The .geo file produced by the SMS software in the final step contains three main components: node information, element information, and boundary information.

The main part of the node information, as shown in Figure 3.3, includes the x and y coordinates at each global node index i , with the channel bed depth (depth(i) in Figure 3.3) represented by the $-Z_0$ value. The global node index refers to the index of a node in the entire domain, rather than just within a specific element.

#	i	ncode	ncbdy	x(i)	y(i)	depth(i)
	1	3	3	0.0000	200.0000	10.000

Figure 3.3. Node information extracted from the SMS .geo file.

The element information, as shown in Figure 3.4, specifies the element index and the three global node indices of the current element in their partial vertex order. For example, “i 1: 1 12 13” indicates that element #1 has three nodes, where vertex 1 is node 1, vertex 2 is node 12, and vertex 3 is node 13. Additionally, the irch value represents the corresponding reach index for each element. By checking the reach index in Figure 3.5, one can retrieve all related coefficients or properties for the element, which will not be detailed here.

#	i	node1	node2	node3	irch
	1	1	12	13	2

Figure 3.4. Element information extracted from the SMS .geo file.

#	irch	cnn	cnnbdr	exx	eyy	exy	smn	wupwind	dsnb	cohb	theta0	theta1	beta1	vbb	gama	dni	dinice	dice
1	0.025	1.000	1.000	1.000	1.000	1.000	0.990	0.250	0.010	1	1	1	0	0.001	0	0	0.02	0.02
2	0.025	1.000	1.000	1.000	1.000	1.000	0.990	0.250	0.010	1	1	1	0	0.001	0	0	0.02	0.02

Figure 3.5. Reach information extracted from the SMS .geo file.

The boundary information lists all river boundary types and their associated boundary global node index as shown in Figure 3.6. Note that the ‘Open Boundary’ in the Figure is equivalent to a flux boundary.

```
# Land Boundary 1
101
11 32 53 74 95 116 137 158 179 200
221 242 263 284 305 326 347 368 389 410
431 452 473 494 515 536 557 578 599 620
641 662 683 704 725 746 767 788 809 830
851 872 893 914 935 956 977 998 1019 1040
1061 1082 1103 1124 1145 1166 1187 1208 1229 1250
1271 1292 1313 1334 1355 1376 1397 1418 1439 1460
1481 1502 1523 1544 1565 1586 1607 1628 1649 1670
1691 1712 1733 1754 1775 1796 1817 1838 1859 1880
1901 1922 1943 1964 1985 2006 2027 2048 2069 2090
2111

# Land Boundary 2
101
2092 2071 2050 2029 2008 1987 1966 1945 1924 1903
1882 1861 1840 1819 1798 1777 1756 1735 1714 1693
1672 1651 1630 1609 1588 1567 1546 1525 1504 1483
1462 1441 1420 1399 1378 1357 1336 1315 1294 1273
1252 1231 1210 1189 1168 1147 1126 1105 1084 1063
1042 1021 1000 979 958 937 916 895 874 853
832 811 790 769 748 727 706 685 664 643
622 601 580 559 538 517 496 475 454 433
412 391 370 349 328 307 286 265 244 223
202 181 160 139 118 97 76 55 34 13

# Open Boundary 1
11
1 2 3 4 5 6 7 8 9 10
11

# Open Boundary 2
11
2111 2109 2107 2105 2103 2101 2099 2097 2095 2093
2092
```

Figure 3.6. Boundary information extracted from the SMS .geo file.

Together, this file encapsulates all the geometric and boundary data required to solve a FEM problem for the discretized domain.

A potential future improvement for the space discretization process is to handle all the steps using the Gmsh graphical user interface, streamlining the workflow, eliminating the need for file format conversions, and simplifying the process.

3.3. Finite Element Formulations with Petrov-Galerkin Weighted Residual Method

This subsection focuses on formulating the discretized version of the governing system of equations using the Petrov-Galerkin weighted residual method. The hydrodynamic equations introduced in Chapter 2 are transformed into conservative and non-conservative forms and a Galerkin method is applied to each, with different test functions. These formulations combine to form a system of ODEs that is solved numerically. Details of how boundary conditions impact the resulting equations are also provided.

3.3.1. Introduction of Galerkin Weighted Residual Method

A general expression for representing a PDE for an initial-boundary value problem (IBVP) in a discretized domain Ω^e is [17]:

$$L\tilde{U}^e = f, \quad (3.3)$$

where f is the forcing function, L is a differential operator, and $\tilde{U}^e$ is the unknown approximate function sought to (approximately) satisfy the PDE. The goal of the Galerkin weighted residual method is to obtain the best approximate function by minimizing the weighted residual between the left and right-hand sides of the PDE. Based on Equation 3.3, the residual r is expressed as:

$$r = L\tilde{U}^e - f. \quad (3.4)$$

In order to minimize the residual in the weak formulation, a set of weighting functions (also called test functions) w_i is introduced. A general approach is to select the weighting functions to be the same as the basis functions ϕ_i used in the expansion of the unknowns ($\tilde{U}^e$) [17], although other choices are also possible. In CRISSP2D, the weighting function is selected as a combination of the basis functions and additional terms that incorporate upwind matrices to enhance numerical stability. The mathematical representation of the integral of the weighted residual R_i is given by:

$$R_i = \int_{\Omega^e} w_i r \, d\Omega = 0. \quad (3.5)$$

The collection of R_i for each test function forms a system of equations (ODEs in the case of the shallow water equation), where the residual r is expressed in terms of the unknown coefficients in $\tilde{U}$, allowing the system to be solved numerically

3.3.2. CRISSP2D with Galerkin Weighted Residual Method

The CRISSP2D model modifies the traditional Galerkin method to account for the non-linearity inherent in the hydrodynamic equations. Specifically, the model transforms the hydrodynamic equations into conservative and non-conservative forms and applies different weighting or testing functions to each form.

In the following sections, we will look at these two forms - the conservative and non-conservative separately, highlighting how the weighting functions differ and how each form is handled within CRISSP2D.

3.3.2.1. Conservative Form with Galerkin Testing

The general conservative form of a PDE for an unknown quantity U is written as:

$$\frac{\partial U}{\partial t} + \nabla \cdot \vec{E}(U) = S, \quad (3.6)$$

where:

- U is the unknown quantity (e.g. h , p or q in the hydrodynamic model)
- $\nabla \cdot \vec{E}(U)$ describes how the flux of the conserved quantity changes within the domain in both the x and y directions and is usually referred to as the **flux term**. This flux term can be further expanded as: $\vec{E}(U) = [F(U), G(U)]$. At this point, the divergence term is purely related to the interior of the element. The influence of the flux across element boundaries will only be introduced when integrated over a control volume (finite element) during the testing process, which will be discussed in a later section.
- S is the source term. In the hydrodynamic equations, the right-hand side of the equation is interpreted as the source term, which depends on the solution. It also acts as the forcing function f in Equation (3.3).

The residual function r for the general conservative form equation (3.6) is derived as:

$$r = \frac{\partial U}{\partial t} + \nabla \cdot \vec{E}(U) - S(U). \quad (3.7)$$

A Galerkin method is defined to use the same testing functions as basis functions. That is, by employing the test function ϕ_i associated with vertex i , the conservative form (Equation

(3.6)), leads to a Galerkin testing method, which can be written as:

$$\int_{\Omega^e} \phi_i \left(\frac{\partial U}{\partial t} + \nabla \cdot \vec{E}(U) - S(U) \right) d\Omega. \quad (3.8)$$

For the conservative form, we derive the discrete equation by first approximating the unknown quantity U using basis function expansion approximation $\tilde{U}$. The approximated unknowns are then substituted into the governing equations (3.8), and the flux and source terms are evaluated based on these approximations, denoted as $\vec{E}(\tilde{U})$ and $S(\tilde{U})$. The resulting system of equations can be expressed as follows:

$$\int_{\Omega^e} \phi_i \left(\frac{\partial \tilde{U}}{\partial t} + \nabla \cdot \vec{E}(\tilde{U}) - S(\tilde{U}) \right) d\Omega. \quad (3.9)$$

From this general form, which handles a single PDE with one unknown quantity U , obtaining the specific equations required for computational solutions to the hydrodynamic equations in CRISP2D involves several steps. While the hydrodynamic equations form a system of PDEs with a vector of unknowns, the same steps apply for both cases.

1. **Formulate the System of PDEs with Galerkin Testing:** The hydrodynamic equations introduced in Chapter 2 are rewritten to match the general structure of the conservative form. After that, the integral of the residual function is formulated by applying the Galerkin testing procedure.

Recall the hydrodynamic equations (Equation (2.8)) introduced in Chapter 2, which are repeated here for convenience:

$$\frac{\partial h}{\partial t} + \frac{\partial p}{\partial x} + \frac{\partial q}{\partial y} = 0$$

$$\begin{aligned}\frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(\frac{p^2}{h}\right) + \frac{\partial}{\partial y}\left(\frac{pq}{h}\right) &= \frac{\tau_{sx} - \tau_{bx}}{\rho} - gh\frac{\partial \eta}{\partial x} + \frac{1}{\rho}\left(\frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y}\right) \\ \frac{\partial q}{\partial t} + \frac{\partial}{\partial x}\left(\frac{pq}{h}\right) + \frac{\partial}{\partial y}\left(\frac{q^2}{h}\right) &= \frac{\tau_{sy} - \tau_{by}}{\rho} - gh\frac{\partial \eta}{\partial y} + \frac{1}{\rho}\left(\frac{\partial T_{xy}}{\partial x} + \frac{\partial T_{yy}}{\partial y}\right).\end{aligned}$$

These equations can be written into the general structure of the conservative form as follows:

$$\frac{\partial}{\partial t} \begin{bmatrix} h \\ p \\ q \end{bmatrix} + \frac{\partial}{\partial x} \begin{bmatrix} p \\ p^2/h \\ pq/h \end{bmatrix} + \frac{\partial}{\partial y} \begin{bmatrix} q \\ pq/h \\ q^2/h \end{bmatrix} = \begin{bmatrix} 0 \\ \rho^{-1}(\tau_{sx} - \tau_{bx} + \partial_x T_{xx} + \partial_y T_{yx}) - gh\partial_x \eta \\ \rho^{-1}(\tau_{sy} - \tau_{by} + \partial_x T_{xy} + \partial_y T_{yy}) - gh\partial_y \eta \end{bmatrix} \quad (3.10)$$

or

$$\frac{\partial}{\partial t} \underline{U} + \frac{\partial}{\partial x} \underline{F}(\underline{U}) + \frac{\partial}{\partial y} \underline{G}(\underline{U}) = \underline{S}(\underline{U}), \quad (3.11)$$

where the definitions of

$$\underline{U} = \begin{bmatrix} h \\ p \\ q \end{bmatrix},$$

$\underline{F}(\underline{U})$, $\underline{G}(\underline{U})$ and $\underline{S}(\underline{U})$ are apparent from Equation (3.10).

We can further simplify by combining the flux terms $\underline{F}(\underline{U})$ and $\underline{G}(\underline{U})$ as the general flux term $\vec{E}(\underline{U})$. This matches the general structure presented in Equation (3.6).

For notational convenience we introduce the collection of basis functions on element e as the vector $\underline{\Phi}^e$, testing a single equation (say the h equation) with all three basis

functions is equivalent to multiplying by the vector

$$\underline{\Phi}^e = \begin{bmatrix} \phi_1^e \\ \phi_2^e \\ \phi_3^e \end{bmatrix} \quad (3.12)$$

and integrating. Finally, testing all three equations is equivalent to introducing the matrix

$$\underline{\Phi}^e = \begin{bmatrix} \underline{\Phi}^e & \underline{0} & \underline{0} \\ \underline{0} & \underline{\Phi}^e & \underline{0} \\ \underline{0} & \underline{0} & \underline{\Phi}^e \end{bmatrix}, \quad (3.13)$$

where

$$\underline{0} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix},$$

and multiplying and integrating. The integral of the residual form is then formulated by applying the Galerkin test as:

$$\int_{\Omega^e} \underline{\Phi}^e \left(\frac{\partial \underline{U}}{\partial t} + \frac{\partial}{\partial x} \underline{F}(\underline{U}) + \frac{\partial}{\partial y} \underline{G}(\underline{U}) - \underline{S}(\underline{U}) \right) d\Omega = \underline{0}. \quad (3.14)$$

2. **Substituting the Approximated Functions:** The approximated functions, obtained using the Lagrange interpolation method, will be substituted into the integral of the residual form equation (3.14). In the CRISSP2D model, instead of simply substituting the approximate function of the unknown quantities $\underline{\tilde{U}}^e$ into the flux and source terms, the flux and source terms themselves are approximated as functions of

the approximate unknowns:

$$\underline{\vec{E}}(U) \approx \underline{\tilde{E}}^e(\underline{\tilde{U}}^e) = \begin{bmatrix} \underline{\tilde{F}}^e(\underline{\tilde{U}}^e) \\ \underline{\tilde{G}}^e(\underline{\tilde{U}}^e) \end{bmatrix}, \quad \underline{S}(U^e) \approx \underline{\tilde{S}}^e(\underline{\tilde{U}}^e).$$

Let $f_{i,j}^e(t)$ represent the flux components at the j^{th} node for the i^{th} component of the flux vector $\underline{F}(\underline{U})$ (denoted as $F_i^e(\underline{U})$) in element e . In terms of basis functions ϕ_j^e in element e , $\underline{\tilde{F}}^e(\underline{\tilde{U}}^e)$ can be written as:

$$\underline{\tilde{F}}^e(\underline{\tilde{U}}^e) = \begin{bmatrix} \sum_{j=1}^n \phi_j^e f_{1,j}^e(t) \\ \sum_{j=1}^n \phi_j^e f_{2,j}^e(t) \\ \sum_{j=1}^n \phi_j^e f_{3,j}^e(t) \end{bmatrix}. \quad (3.15)$$

Now, we define:

$$\underline{f}_i^e(t) = \begin{bmatrix} f_{i,1}^e(t) \\ f_{i,2}^e(t) \\ f_{i,3}^e(t) \end{bmatrix} \quad (3.16)$$

to be the vector of nodal values in element e for $F_i^e(\underline{U})$ and we define

$$\underline{f}^e(t) = \begin{bmatrix} \underline{f}_1^e(t) \\ \underline{f}_2^e(t) \\ \underline{f}_3^e(t) \end{bmatrix} \quad (3.17)$$

to be the vector containing the nodal values for all three components of the vector $\underline{F}$. Using the shape function vector (3.12) and the shape function matrix (3.13), $\underline{\tilde{F}}^e(\underline{\tilde{U}}^e)$ can finally be written in a compact form showing the dependence on all of the coefficients

for each component as:

$$\tilde{\underline{F}}^e(\tilde{\underline{U}}^e) = \begin{bmatrix} \underline{\Phi}^{eT} \underline{f}_1^e(t) \\ \underline{\Phi}^{eT} \underline{f}_2^e(t) \\ \underline{\Phi}^{eT} \underline{f}_3^e(t) \end{bmatrix} = \begin{bmatrix} \underline{\Phi}^{eT} & \underline{0}^T & \underline{0}^T \\ \underline{0}^T & \underline{\Phi}^{eT} & \underline{0}^T \\ \underline{0}^T & \underline{0}^T & \underline{\Phi}^{eT} \end{bmatrix} \begin{bmatrix} \underline{f}_1^e(t) \\ \underline{f}_2^e(t) \\ \underline{f}_3^e(t) \end{bmatrix} = \underline{\Phi}^{eT} \underline{f}^e(t). \quad (3.18)$$

Similarly, if we define $\underline{g}^e(t)$ and $\underline{s}^e(t)$ as the vector of nodal values in element e for the flux term $\underline{G}(\underline{U})$ and the source term $\underline{S}(\underline{U})$, respectively, we can obtain the approximate functions $\tilde{\underline{G}}^e(\tilde{\underline{U}}^e)$ and $\tilde{\underline{S}}^e(\tilde{\underline{U}}^e)$ in the same manner as for $\tilde{\underline{F}}^e(\tilde{\underline{U}}^e)$, where

$$\tilde{\underline{G}}^e(\tilde{\underline{U}}^e) = \underline{\Phi}^{eT} \underline{g}^e(t), \quad \tilde{\underline{S}}^e(\tilde{\underline{U}}^e) = \underline{\Phi}^{eT} \underline{s}^e(t). \quad (3.19)$$

A similar process can also be applied to the unknown quantities term $\tilde{\underline{U}}^e$, where

$$\tilde{\underline{U}}^e = \underline{\Phi}^{eT} \underline{u}^e(t). \quad (3.20)$$

Substituting all the components $\tilde{\underline{U}}^e$, $\tilde{\underline{F}}^e(\tilde{\underline{U}}^e)$, $\tilde{\underline{G}}^e(\tilde{\underline{U}}^e)$, and $\tilde{\underline{S}}^e(\tilde{\underline{U}}^e)$ back into the integral of the residual equation (3.14) leads to a system of equations

$$\int_{\Omega^e} \underline{\Phi}^e \left(\frac{\partial}{\partial t} \underline{\Phi}^{eT} \underline{u}^e(t) + \frac{\partial}{\partial x} \underline{\Phi}^{eT} \underline{f}^e(t) + \frac{\partial}{\partial y} \underline{\Phi}^{eT} \underline{g}^e(t) - \underline{\Phi}^{eT} \underline{s}^e(t) \right) d\Omega. \quad (3.21)$$

This expression contains spatial integrals of combinations of known basis functions that can be computed directly, leaving a system of ODEs as derived in the next step.

3. **Converting into a system of ODEs:** The system of PDEs (3.21) can be converted into a system of ordinary differential equations by evaluating the spatial integrals, allowing for efficient computation. Then, the remaining derivative d/dt can be ap-

proximated using a time-stepping scheme such as the Leapfrog method which will be explained in detail later in this Chapter. In Equation (3.21), the spatial derivatives $\partial/\partial x$ and $\partial/\partial y$ are applied to the basis functions within each element. The integrals, essential for the Galerkin method, are straightforward to compute. Evaluating the integrals converts the continuous partial differential equations into a discrete system of ODEs.

To proceed, we begin by further expanding the system so that the integral of each term can be handled separately. This allows us to apply numerical integration methods to each term in the equation, ensuring that the contributions from the time derivative, spatial derivatives, flux terms, and source terms are treated appropriately. To this end we rewrite Equation (3.21) as:

$$\int_{\Omega^e} \Phi^e \frac{\partial}{\partial t} \Phi^{eT} \underline{u}^e(t) + \Phi^e \frac{\partial}{\partial x} \Phi^{eT} \underline{f}^e(t) + \Phi^e \frac{\partial}{\partial y} \Phi^{eT} \underline{g}^e(t) - \Phi^e \Phi^{eT} \underline{s}^e(t) d\Omega. \quad (3.22)$$

Starting with the first term in the sum, i.e, the time derivative term, we observe that since $\partial/\partial t$ only applies to $\underline{u}^e(t)$ which is a function that varies with time, and the integral over the element domain applies only to the basis functions, which vary with space. Thus we can restructure this term as:

$$\int_{\Omega^e} \Phi^e \left(\frac{\partial}{\partial t} \Phi^{eT} \underline{u}^e(t) \right) d\Omega = \int_{\Omega^e} \Phi^e \Phi^{eT} d\Omega \frac{d}{dt} \underline{u}^e(t). \quad (3.23)$$

Substituting the full expression Φ^e results in:

$$\begin{aligned}
& \int_{\Omega^e} \begin{bmatrix} \underline{\Phi}^e & \underline{0} & \underline{0} \\ \underline{0} & \underline{\Phi}^e & \underline{0} \\ \underline{0} & \underline{0} & \underline{\Phi}^e \end{bmatrix} \begin{bmatrix} \underline{\Phi}^{eT} & \underline{0}^T & \underline{0}^T \\ \underline{0}^T & \underline{\Phi}^{eT} & \underline{0}^T \\ \underline{0}^T & \underline{0}^T & \underline{\Phi}^{eT} \end{bmatrix} d\Omega \frac{d}{dt} \underline{u}^e(t) \\
&= \begin{bmatrix} \int_{\Omega^e} \underline{\Phi}^e \underline{\Phi}^{eT} d\Omega & \underline{00}^T & \underline{00}^T \\ \underline{00}^T & \int_{\Omega^e} \underline{\Phi}^e \underline{\Phi}^{eT} d\Omega & \underline{00}^T \\ \underline{00}^T & \underline{00}^T & \int_{\Omega^e} \underline{\Phi}^e \underline{\Phi}^{eT} d\Omega \end{bmatrix} \frac{d}{dt} \underline{u}^e(t),
\end{aligned} \tag{3.24}$$

where:

$$\begin{aligned}
\int_{\Omega^e} \underline{\Phi}^e \underline{\Phi}^{eT} d\Omega &= \int_{\Omega^e} \begin{bmatrix} \phi_1^e \\ \phi_2^e \\ \phi_3^e \end{bmatrix} \begin{bmatrix} \phi_1^e & \phi_2^e & \phi_3^e \end{bmatrix} d\Omega \\
&= \int_{\Omega^e} \begin{bmatrix} \phi_1^e \phi_1^e & \phi_1^e \phi_2^e & \phi_1^e \phi_3^e \\ \phi_2^e \phi_1^e & \phi_2^e \phi_2^e & \phi_2^e \phi_3^e \\ \phi_3^e \phi_1^e & \phi_3^e \phi_2^e & \phi_3^e \phi_3^e \end{bmatrix} d\Omega = \mathcal{M}^e.
\end{aligned} \tag{3.25}$$

Here, $\mathcal{M}^e$ refers to the scalar mass matrix on element e . It captures the interactions (projections) between all basis functions and can be computed once for the element. In practice, closed-form expressions are available for computing the matrix entries for first-order basis functions on first-order triangles [19]. To improve computational efficiency and reduce memory usage, CRISSP2D employs a lumped mass matrix. In this approach, the mass matrix is replaced by a diagonal matrix, where each diagonal entry is the sum of the corresponding row of the original local mass matrix [11]. For

first-order triangular elements, the lumped mass matrix on element e becomes:

$$\mathcal{M}_{lumped}^e = \int_{\Omega^e} \begin{bmatrix} \sum_{j=1}^3 \phi_1^e \phi_j^e & 0 & 0 \\ 0 & \sum_{j=1}^3 \phi_2^e \phi_j^e & 0 \\ 0 & 0 & \sum_{j=1}^3 \phi_3^e \phi_j^e \end{bmatrix} d\Omega \quad (3.26)$$

Alternatively, numerical integration schemes can be used to pre-compute the mass matrix entries, an approach that extends easily to curved geometries and high-order expansions.

Substituting the partial mass matrix (3.25) back into Equation (3.24), and introducing the block diagonal mass matrix $\mathbf{M}^e$ as:

$$\mathbf{M}^e = \begin{bmatrix} \mathcal{M}^e & \underline{00}^T & \underline{00}^T \\ \underline{00}^T & \mathcal{M}^e & \underline{00}^T \\ \underline{00}^T & \underline{00}^T & \mathcal{M}^e \end{bmatrix},$$

we obtain a numerical form of the expression that involves only the unknown quantities $\underline{u}^e(t)$, represented as the nodal quantities vector as a function of time:

$$\int_{\Omega^e} \Phi^e \left(\frac{\partial}{\partial t} \Phi^{eT} \underline{u}^e(t) \right) d\Omega = \begin{bmatrix} \mathcal{M}^e & \underline{00}^T & \underline{00}^T \\ \underline{00}^T & \mathcal{M}^e & \underline{00}^T \\ \underline{00}^T & \underline{00}^T & \mathcal{M}^e \end{bmatrix} \frac{d}{dt} \underline{u}^e(t) = \mathbf{M}^e \frac{d}{dt} \underline{u}^e(t). \quad (3.27)$$

Note that the time derivative will be resolved later in this Chapter.

We can apply a similar approach to the last term in Equation (3.22), except that the last term does not involve a time derivative. This results in a similar numerical form

for the last term:

$$\int_{\Omega^e} \Phi^e \Phi^{eT} \underline{s}^e(t) d\Omega = \int_{\Omega^e} \Phi^e \Phi^{eT} d\Omega \underline{s}^e(t) = \mathbf{M}^e \underline{s}^e(t). \quad (3.28)$$

The procedure for handling the second and third terms in Equation (3.22) differs slightly from that of the unknown quantities and source terms, as these terms involve spatial derivatives. As mentioned earlier, these spatial derivatives are applied to the basis functions, which vary with space. Thus, we can restructure these two terms as:

$$\int_{\Omega^e} \Phi^e \frac{\partial}{\partial x} \Phi^{eT} \underline{f}^e(t) d\Omega = \int_{\Omega^e} \Phi^e \frac{\partial}{\partial x} \Phi^{eT} d\Omega \underline{f}^e(t) \quad (3.29)$$

and

$$\int_{\Omega^e} \Phi^e \frac{\partial}{\partial y} \Phi^{eT} \underline{g}^e(t) d\Omega = \int_{\Omega^e} \Phi^e \frac{\partial}{\partial x} \Phi^{eT} d\Omega \underline{g}^e(t). \quad (3.30)$$

Substituting the full expression of Φ^e as before in the second term results in:

$$\begin{aligned} & \int_{\Omega^e} \begin{bmatrix} \underline{\Phi}^e & \underline{0} & \underline{0} \\ \underline{0} & \underline{\Phi}^e & \underline{0} \\ \underline{0} & \underline{0} & \underline{\Phi}^e \end{bmatrix} \frac{\partial}{\partial x} \begin{bmatrix} \underline{\Phi}^{eT} & \underline{0}^T & \underline{0}^T \\ \underline{0}^T & \underline{\Phi}^{eT} & \underline{0}^T \\ \underline{0}^T & \underline{0}^T & \underline{\Phi}^{eT} \end{bmatrix} d\Omega \underline{f}^e(t) \\ &= \begin{bmatrix} \int_{\Omega^e} \underline{\Phi}^e \underline{\partial}_x \underline{\Phi}^{eT} d\Omega & \underline{00}^T & \underline{00}^T \\ \underline{00}^T & \int_{\Omega^e} \underline{\Phi}^e \underline{\partial}_x \underline{\Phi}^{eT} d\Omega & \underline{00}^T \\ \underline{00}^T & \underline{00}^T & \int_{\Omega^e} \underline{\Phi}^e \underline{\partial}_x \underline{\Phi}^{eT} d\Omega \end{bmatrix} \underline{f}^e(t). \end{aligned} \quad (3.31)$$

If we define:

$$\begin{aligned}
\int_{\Omega^e} \underline{\Phi^e} \underline{\partial_x \Phi^{eT}} d\Omega &= \int_{\Omega^e} \begin{bmatrix} \phi_1^e \\ \phi_2^e \\ \phi_3^e \end{bmatrix} \begin{bmatrix} \partial_x \phi_1^e & \partial_x \phi_2^e & \partial_x \phi_3^e \end{bmatrix} d\Omega \\
&= \int_{\Omega^e} \begin{bmatrix} \phi_1^e \partial_x \phi_1^e & \phi_1^e \partial_x \phi_2^e & \phi_1^e \partial_x \phi_3^e \\ \phi_2^e \partial_x \phi_1^e & \phi_2^e \partial_x \phi_2^e & \phi_2^e \partial_x \phi_3^e \\ \phi_3^e \partial_x \phi_1^e & \phi_3^e \partial_x \phi_2^e & \phi_3^e \partial_x \phi_3^e \end{bmatrix} d\Omega = \mathcal{S}_x^e,
\end{aligned} \tag{3.32}$$

where $\mathcal{S}_x^e$ is the x -derivative partial stiffness matrix on element e whose entries involve interactions of basis functions with x -derivatives of basis functions. Like the mass matrix, the stiffness matrix is also straightforward to pre-compute [19]. By introducing the block diagonal stiffness matrix $\mathbf{S}_x^e$, the numerical form for the second term involving $\partial/\partial x$ can be written as:

$$\int_{\Omega^e} \underline{\Phi^e} \frac{\partial}{\partial x} \underline{\Phi^{eT}} \underline{f^e}(t) d\Omega = \begin{bmatrix} \mathcal{S}_x^e & \underline{00^T} & \underline{00^T} \\ \underline{00^T} & \mathcal{S}_x^e & \underline{00^T} \\ \underline{00^T} & \underline{00^T} & \mathcal{S}_x^e \end{bmatrix} \underline{f^e}(t) = \mathbf{S}_x^e \underline{f^e}(t). \tag{3.33}$$

Similarly, if we define $\underline{\Phi_i} \underline{\partial_y \Phi_i^T} = \mathcal{S}_y^e$, where $\mathcal{S}_y^e$ is the y -derivative partial stiffness matrix on element e , and apply the same procedure to the third term with $\partial/\partial y$, we can obtain the numerically solvable form:

$$\int_{\Omega^e} \underline{\Phi^e} \frac{\partial}{\partial y} \underline{\Phi^{eT}} \underline{g^e}(t) d\Omega = \mathbf{S}_y^e \underline{g^e}(t). \tag{3.34}$$

Having derived suitable numerical forms of all terms in Equation (3.21) where all spatial integrals are evaluated, we now combine these individual terms to arrive at a system of ODEs in time, that can be efficiently solved by a computer. The final system of ODEs is:

$$\mathbf{M}^e \partial_t \underline{u}^e(t) + \mathbf{S}_x^e \underline{f}^e(t) + \mathbf{S}_y^e \underline{g}^e(t) - \mathbf{M}^e \underline{s}^e(t) \quad (3.35)$$

We will discretize this system in time and solve for the unknown quantities $u_i(t)$ in the following sections.

3.3.2.2. Non-Conservative Form with Galerkin Testing:

In standard Galerkin finite element methods using the conservative form of hydrodynamic equations, where the testing functions are equal to the basis functions, can lead to undesirable oscillations in hydrodynamic problems due to the hyperbolic nature of their governing equations. To address this, the Streamline-Upwind Petrov-Galerkin (SUPG) [20] formulation is employed in CRISSP2D. SUPG modifies the standard Galerkin method by introducing an additional stabilization term in the matrix representation of the testing functions, denoted as Φ' . The modified testing functions incorporates upwind weighting, aligning the test functions with the dominant flow direction. By considering the flow velocity direction, SUPG effectively suppresses numerical oscillations, resulting in a significant improvement in numerical stability compared to the standard Galerkin method [21]. The non-conservative form of hydrodynamic equations was chosen specifically to implement the SUPG method.

The non-conservative form is written by applying derivatives directly to the primitive unknowns for the flux terms. The general structure of a PDE for an unknown quantity U in

non-conservative form is written as:

$$\frac{\partial U}{\partial t} + \vec{\mathbf{A}}(U) \cdot \nabla U = S(U), \quad (3.36)$$

where:

- U represents the unknown quantity (e.g. h , p or q)
- $\vec{\mathbf{A}}(U) = [\mathbf{A}_x(U), \mathbf{A}_y(U)]$ represents the convection matrices with respect to the flux term $\vec{E}(U)$ in the conservative form, in the x and y directions. These matrices are derived from the flux terms $\vec{E}(U)$ as $\vec{\mathbf{A}}(U) = [\partial F / \partial U, \partial G / \partial U]$. Details regarding $\mathbf{A}_x(U)$ and $\mathbf{A}_y(U)$ are provided later in this section.
- S is the source term. In the hydrodynamic equations, the right-hand side of the equations is interpreted as the source term, which depends on the solution, also act as the forcing function.

In CRISP2D, the SUPG formulation includes two terms from the non-conservative form: tested versions of $\mathbf{A}(U) \cdot \nabla U$ and $S(U)$ [11]. Following the standard Galerkin testing procedure, by employing the test function ϕ_i , we arrive at a Galerkin testing method that can be written as:

$$\int_{\Omega^e} \phi_i (\mathbf{A}(U) \cdot \nabla U - S(U)) d\Omega. \quad (3.37)$$

By further expanding the dot product $\mathbf{A}(U) \cdot \nabla U$ in the Equation (3.37), we obtain:

$$\int_{\Omega^e} \phi_i (\mathbf{A}_x \partial_x U + \mathbf{A}_y \partial_y U - S(U)) d\Omega. \quad (3.38)$$

The process for testing the non-conservative form follows the same Galerkin procedure as the previously-described conservative testing.

In order to rewrite the hydrodynamic equations introduced in Chapter 2 in the general structure of the non-conservative form, we must first reformulate the hydrodynamic equations from Equation (2.8) into a form that is compatible with the general non-conservative structure. The transformed hydrodynamic equations, which will be used to further restructure into the non-conservative form, are as follows:

$$\begin{aligned}\frac{\partial h}{\partial t} + \frac{\partial p}{\partial x} + \frac{\partial q}{\partial y} &= 0 \\ \frac{\partial q}{\partial t} + \frac{\partial}{\partial x}\left(\frac{pq}{h}\right) + \frac{\partial}{\partial y}\left(\frac{q^2}{h} + \frac{gh^2}{2}\right) &= \frac{\tau_{sy} - \tau_{by}}{\rho} - gh\frac{\partial Z_0}{\partial y} + \frac{1}{\rho}\left(\frac{\partial T_{xy}}{\partial x} + \frac{\partial T_{yy}}{\partial y}\right) \\ \frac{\partial q}{\partial t} + \frac{\partial}{\partial x}\left(\frac{pq}{h}\right) + \frac{\partial}{\partial y}\left(\frac{q^2}{h} + \frac{gh^2}{2}\right) &= \frac{\tau_{sy} - \tau_{by}}{\rho} - gh\frac{\partial Z_0}{\partial y} + \frac{1}{\rho}\left(\frac{\partial T_{xy}}{\partial x} + \frac{\partial T_{yy}}{\partial y}\right).\end{aligned}\tag{3.39}$$

A detailed derivation of the reformulation process is covered in Appendix A.

Next, we seek to explicitly determine the values of $\mathbf{A}_x \partial_x U + \mathbf{A}_y \partial_y U - S(U)$ from the governing equation (3.39). Letting $c = \sqrt{gh}$, we can further rewrite the equation (3.39) as the following:

$$\begin{aligned}\frac{\partial h}{\partial t} + \frac{\partial p}{\partial x} + \frac{\partial q}{\partial y} &= 0 \\ \frac{\partial p}{\partial t} + (c^2 - u^2)\frac{\partial h}{\partial x} + 2u\frac{\partial p}{\partial x} + u\frac{\partial q}{\partial y} + v\frac{\partial p}{\partial y} - uv\frac{\partial h}{\partial y} &= \rho^{-1}(\tau_{sx} - \tau_{bx} + \partial_x T_{xx} + \partial_y T_{yx}) - gh\partial_x Z_0 \\ \frac{\partial q}{\partial t} + u\frac{\partial q}{\partial x} + v\frac{\partial p}{\partial x} - uv\frac{\partial h}{\partial x} + (c^2 - v^2)\frac{\partial h}{\partial y} + 2v\frac{\partial q}{\partial y} &= \rho^{-1}(\tau_{sy} - \tau_{by} + \partial_x T_{xy} + \partial_y T_{yy}) - gh\partial_y Z_0.\end{aligned}\tag{3.40}$$

A detailed derivation can be found in Appendix B. After regrouping Equation (3.40) into the following matrix form

$$\begin{aligned}
\frac{\partial}{\partial t} \begin{bmatrix} h \\ p \\ q \end{bmatrix} + \begin{bmatrix} 0 & 1 & 0 \\ c^2 - u^2 & 2u & 0 \\ -uv & v & u \end{bmatrix} \frac{\partial}{\partial x} \begin{bmatrix} h \\ p \\ q \end{bmatrix} + \begin{bmatrix} 0 & 0 & 1 \\ -uv & v & u \\ c^2 - v^2 & 0 & 2v \end{bmatrix} \frac{\partial}{\partial y} \begin{bmatrix} h \\ p \\ q \end{bmatrix} \\
= \begin{bmatrix} 0 \\ \rho^{-1}(\tau_{sx} - \tau_{bx} + \partial_x T_{xx} + \partial_y T_{yx}) - gh\partial_x Z_0 \\ \rho^{-1}(\tau_{sy} - \tau_{by} + \partial_x T_{xy} + \partial_y T_{yy}) - gh\partial_y Z_0 \end{bmatrix}, \tag{3.41}
\end{aligned}$$

we can explicitly determine

$$U = \begin{bmatrix} h \\ p \\ q \end{bmatrix}, \quad \mathbf{A}_x = \begin{bmatrix} 0 & 1 & 0 \\ c^2 - u^2 & 2u & 0 \\ -uv & v & u \end{bmatrix}, \quad \mathbf{A}_y = \begin{bmatrix} 0 & 0 & 1 \\ -uv & v & u \\ c^2 - v^2 & 0 & 2v \end{bmatrix} \tag{3.42}$$

and

$$\underline{S}(U) = \begin{bmatrix} 0 \\ \rho^{-1}(\tau_{sx} - \tau_{bx} + \partial_x T_{xx} + \partial_y T_{yx}) - gh\partial_x Z_0 \\ \rho^{-1}(\tau_{sy} - \tau_{by} + \partial_x T_{xy} + \partial_y T_{yy}) - gh\partial_y Z_0 \end{bmatrix},$$

such that:

$$\frac{\partial}{\partial t} \underline{U} + \mathbf{A}_x \frac{\partial}{\partial x} \underline{U} + \mathbf{A}_y \frac{\partial}{\partial y} \underline{U} = \underline{S}(\underline{U}). \tag{3.43}$$

After explicitly determining the values of components in the non-conservative form of the governing equation (3.43), we now need to determine the testing functions, which are introduced through the SUPG method.

The SUPG stabilization weighting function matrix Φ' used to test in CRISSP2D for testing, can be expressed as follows:

$$\Phi' = \omega \frac{\Delta x}{2} \frac{\partial \Phi}{\partial x} \mathbf{W}_x + \omega \frac{\Delta y}{2} \frac{\partial \Phi}{\partial y} \mathbf{W}_y, \quad (3.44)$$

where

- ω represents the stability parameter, which can be adjusted by the user to determine how much of these additional terms should be considered in the weighting
- $\mathbf{W}_x$ and $\mathbf{W}_y$ are the upwind matrices, in the x and y directions, respectively. These terms help adjust the weighting differently based on the flow properties in the x and y directions.
- Δx and Δy represent the element length in x and y directions, respectively, which improves the overall stability by accounting for mesh size. In CRISSP2D, they are defined as $\Delta x = \Delta y = \sqrt{2 \cdot \text{Area}}$ where “Area” is the area of Ω^e .
- Φ represents the standard shape function matrix (3.13)

The Characteristic Dissipative Galerkin (CDG) method is applied in CRISSP2D to set up the upwind matrices, which are defined as [11]:

$$\mathbf{W}_x = \mathbf{R}_{A_x} \cdot |\mathbf{\Lambda}_{A_x}| \cdot \mathbf{L}_{A_x}, \quad \mathbf{W}_y = \mathbf{R}_{A_y} \cdot |\mathbf{\Lambda}_{A_y}| \cdot \mathbf{L}_{A_y}. \quad (3.45)$$

To provide more detail, we now examine the specific components of the upwind matrices.

For $\mathbf{W}_x$,

$$\mathbf{R}_{A_x} = \begin{bmatrix} 1 & 1 & 0 \\ u+c & u-c & 0 \\ v & v & 1 \end{bmatrix}$$

and

$$\mathbf{L}_{A_x} = \mathbf{R}_{A_x}^{-1} = \frac{1}{2c} \begin{bmatrix} -(u-c) & 1 & 0 \\ u+c & -1 & 0 \\ -2vc & 0 & 2c \end{bmatrix}$$

are the right and left eigenvector matrices of $\mathbf{A}_x$ (3.42), and

$$|\mathbf{\Lambda}_{A_x}| = \begin{bmatrix} \frac{|u+c|}{\sqrt{\lambda_1^2+v^2}} & & \\ & \frac{|u-c|}{\sqrt{\lambda_2^2+v^2}} & \\ & & \frac{|u|}{\sqrt{\lambda_3^2+v^2}} \end{bmatrix}$$

is the diagonal matrix of the absolute value of the eigenvalues ($\lambda_1 = u + c$, $\lambda_2 = u - c$ and $\lambda_3 = u$) of $\mathbf{A}_x$ with some modifications to account for specific stability requirements or flow conditions.

Similarly for $\mathbf{W}_y$,

$$\mathbf{R}_{A_y} = \begin{bmatrix} 1 & 1 & 0 \\ u & u & 1 \\ c+v & v-c & 0 \end{bmatrix}, \quad \mathbf{L}_{A_y} = \mathbf{R}_{A_y}^{-1} = \frac{1}{2c} \begin{bmatrix} -(v-c) & 0 & 1 \\ v+c & 0 & -1 \\ -2uc & 2c & 0 \end{bmatrix}$$

and

$$|\mathbf{\Lambda}_{A_y}| = \begin{bmatrix} \frac{|v+c|}{\sqrt{\gamma_1^2+u^2}} & & \\ & \frac{|v-c|}{\sqrt{\gamma_2^2+u^2}} & \\ & & \frac{|v|}{\sqrt{\gamma_3^2+u^2}} \end{bmatrix}$$

where $\gamma_1 = v + c$, $\gamma_2 = v - c$ and $\gamma_3 = v$ are the eigenvalues of $\mathbf{A}_y$.

With the testing functions Φ' (3.44) now well-defined, we can proceed by multiplying it with the additional terms from the non-conservative form and integrating, resulting in the integral of the residual form:

$$\int_{\Omega^e} (\omega \frac{\Delta x}{2} \frac{\partial \Phi^e}{\partial x} \mathbf{W}_x + \omega \frac{\Delta y}{2} \frac{\partial \Phi^e}{\partial y} \mathbf{W}_y) (\mathbf{A}_x \frac{\partial \underline{U}}{\partial x} + \mathbf{A}_y \frac{\partial \underline{U}}{\partial y} - \underline{S}(\underline{U})) d\Omega = 0. \quad (3.46)$$

Begin by approximate the unknown quantities $\underline{U} \approx \tilde{\underline{U}}^e = \Phi^{eT} \underline{u}^e(t)$ and source term $\underline{S}(\underline{U}) \approx \tilde{\underline{S}}^e(\tilde{\underline{U}}) = \Phi^{eT} \underline{s}^e(t)$. Substituting these approximations into the integral of the residual (3.46) gives us:

$$\int_{\Omega^e} (\omega \frac{\Delta x}{2} \frac{\partial \Phi^e}{\partial x} \mathbf{W}_x + \omega \frac{\Delta y}{2} \frac{\partial \Phi^e}{\partial y} \mathbf{W}_y) (\mathbf{A}_x \frac{\partial \tilde{\underline{U}}^e}{\partial x} + \mathbf{A}_y \frac{\partial \tilde{\underline{U}}^e}{\partial y} - \tilde{\underline{S}}^e(\tilde{\underline{U}}^e)) d\Omega. \quad (3.47)$$

Further expanding the system allows us to move to the next step, which involves numerically integrating each term using appropriate numerical schemes.

We define the resulting governing equation as $\mathcal{F}$:

$$\begin{aligned} \mathcal{F} &= \int_{\Omega^e} (\omega \frac{\Delta x}{2} \frac{\partial \Phi^e}{\partial x} \mathbf{W}_x + \omega \frac{\Delta y}{2} \frac{\partial \Phi^e}{\partial y} \mathbf{W}_y) (\mathbf{A}_x \frac{\partial \tilde{\underline{U}}^e}{\partial x} + \mathbf{A}_y \frac{\partial \tilde{\underline{U}}^e}{\partial y} - \tilde{\underline{S}}^e(\underline{U})) d\Omega \\ &= \int_{\Omega^e} \omega \frac{\Delta x}{2} \frac{\partial \Phi^e}{\partial x} \mathbf{W}_x \mathbf{A}_x \frac{\partial \tilde{\underline{U}}^e}{\partial x} + \omega \frac{\Delta y}{2} \frac{\partial \Phi^e}{\partial y} \mathbf{W}_y \mathbf{A}_x \frac{\partial \tilde{\underline{U}}^e}{\partial x} + \omega \frac{\Delta x}{2} \frac{\partial \Phi^e}{\partial x} \mathbf{W}_x \mathbf{A}_y \frac{\partial \tilde{\underline{U}}^e}{\partial y} \\ &\quad + \omega \frac{\Delta y}{2} \frac{\partial \Phi^e}{\partial y} \mathbf{W}_y \mathbf{A}_y \frac{\partial \tilde{\underline{U}}^e}{\partial y} - \omega \frac{\Delta x}{2} \frac{\partial \Phi^e}{\partial x} \mathbf{W}_x \tilde{\underline{S}}^e(\underline{U}) - \omega \frac{\Delta y}{2} \frac{\partial \Phi^e}{\partial y} \mathbf{W}_y \tilde{\underline{S}}^e(\underline{U}) d\Omega. \end{aligned} \quad (3.48)$$

$\mathbf{W}_x \mathbf{A}_x$, $\mathbf{W}_x \mathbf{A}_y$, $\mathbf{W}_y \mathbf{A}_y$, $\mathbf{W}_x$ and $\mathbf{W}_y$ are calculated as the average of the values across all vertices based on the algorithm used in CRISSP2D:

$$\mathbf{W}_x \mathbf{A}_x = \frac{1}{n} \sum_{j=1}^n (\mathbf{W}_{x_j} \mathbf{A}_{x_j}), \quad \mathbf{W}_x = \frac{1}{n} \sum_{j=1}^n \mathbf{W}_{x_j}, \quad (3.49)$$

where $\mathbf{W}_{x_j}$ and $\mathbf{W}_{x_j} \mathbf{A}_{x_j}$ represents the $\mathbf{W}_x$ and $\mathbf{W}_x \mathbf{A}_x$ matrix evaluated using the solution at vertex j , respectively. This approach simplifies the evaluation of the integrals by treating the values at each vertex uniformly across the element, allowing them to be factored out of the integral:

$$\begin{aligned} \mathcal{F} = & \omega \frac{\Delta x}{2} \mathbf{W}_x \mathbf{A}_x \int_{\Omega^e} \frac{\partial \Phi^e}{\partial x} \frac{\partial \tilde{U}^e}{\partial x} d\Omega + \omega \frac{\Delta y}{2} \mathbf{W}_y \mathbf{A}_x \int_{\Omega^e} \frac{\partial \Phi^e}{\partial y} \frac{\partial \tilde{U}^e}{\partial x} d\Omega \\ & + \omega \frac{\Delta x}{2} \mathbf{W}_x \mathbf{A}_y \int_{\Omega^e} \frac{\partial \Phi^e}{\partial x} \frac{\partial \tilde{U}^e}{\partial y} d\Omega + \omega \frac{\Delta y}{2} \mathbf{W}_y \mathbf{A}_y \int_{\Omega^e} \frac{\partial \Phi^e}{\partial y} \frac{\partial \tilde{U}^e}{\partial y} d\Omega \\ & - \omega \frac{\Delta x}{2} \mathbf{W}_x \int_{\Omega^e} \frac{\partial \Phi^e}{\partial x} \tilde{S}^e(\tilde{U}^e) d\Omega - \omega \frac{\Delta y}{2} \mathbf{W}_y \int_{\Omega^e} \frac{\partial \Phi^e}{\partial y} \tilde{S}^e(\tilde{U}^e) d\Omega. \end{aligned} \quad (3.50)$$

Next, we substitute the expansions of the unknown functions $\tilde{U}^e = \Phi^{eT} \underline{u}^e(t)$ and source term $\tilde{S}^e(\tilde{U}) = \Phi^{eT} \underline{s}^e(t)$ into the governing equations $\mathcal{F}$. Since $\underline{u}^e(t)$ and $\underline{s}^e(t)$ are constant in the spatial domain, they can be moved outside of the integral, which gives us:

$$\begin{aligned} \mathcal{F} = & \omega \frac{\Delta x}{2} \mathbf{W}_x \mathbf{A}_x \left(\int_{\Omega^e} \frac{\partial \Phi^e}{\partial x} \frac{\partial \Phi^{eT}}{\partial x} d\Omega \underline{u}^e(t) \right) + \omega \frac{\Delta y}{2} \mathbf{W}_y \mathbf{A}_x \left(\int_{\Omega^e} \frac{\partial \Phi^e}{\partial y} \frac{\partial \Phi^{eT}}{\partial x} d\Omega \underline{u}^e(t) \right) \\ & + \omega \frac{\Delta x}{2} \mathbf{W}_x \mathbf{A}_y \left(\int_{\Omega^e} \frac{\partial \Phi^e}{\partial x} \frac{\partial \Phi^{eT}}{\partial y} d\Omega \underline{u}^e(t) \right) + \omega \frac{\Delta y}{2} \mathbf{W}_y \mathbf{A}_y \left(\int_{\Omega^e} \frac{\partial \Phi^e}{\partial y} \frac{\partial \Phi^{eT}}{\partial y} d\Omega \underline{u}^e(t) \right) \\ & - \omega \frac{\Delta x}{2} \mathbf{W}_x \left(\int_{\Omega^e} \frac{\partial \Phi^e}{\partial x} \Phi^{eT} d\Omega \underline{s}^e(t) \right) - \omega \frac{\Delta y}{2} \mathbf{W}_y \left(\int_{\Omega^e} \frac{\partial \Phi^e}{\partial y} \Phi^{eT} d\Omega \underline{s}^e(t) \right). \end{aligned} \quad (3.51)$$

Similar to the stiffness matrices defined in Section 3.3.2.1 for the conservative form of the equations, here we define stiffness matrices that now involve integrals of derivatives of basis

functions multiplied by derivatives of basis functions:

$$\begin{aligned}
\int_{\Omega^e} \frac{\partial_\xi \Phi^e \partial_\zeta \Phi^{eT}}{d\Omega} d\Omega &= \int_{\Omega^e} \begin{bmatrix} \partial_\xi \phi_1^e \\ \partial_\xi \phi_2^e \\ \partial_\xi \phi_3^e \end{bmatrix} \begin{bmatrix} \partial_\zeta \phi_1^e & \partial_\zeta \phi_2^e & \partial_\zeta \phi_3^e \end{bmatrix} d\Omega \\
&= \int_{\Omega^e} \begin{bmatrix} \partial_\xi \phi_1^e \partial_\zeta \phi_1^e & \partial_\xi \phi_1^e \partial_\zeta \phi_2^e & \partial_\xi \phi_1^e \partial_\zeta \phi_3^e \\ \partial_\xi \phi_2^e \partial_\zeta \phi_1^e & \partial_\xi \phi_2^e \partial_\zeta \phi_2^e & \partial_\xi \phi_2^e \partial_\zeta \phi_3^e \\ \partial_\xi \phi_3^e \partial_\zeta \phi_1^e & \partial_\xi \phi_3^e \partial_\zeta \phi_2^e & \partial_\xi \phi_3^e \partial_\zeta \phi_3^e \end{bmatrix} d\Omega = \mathcal{S}_{\xi\zeta}^e,
\end{aligned} \tag{3.52}$$

where ξ and ζ are each either x or y , depending on the particular term we're dealing with (e.g., $\xi = x$ and $\zeta = y$ for $\mathcal{S}_{xy}$)

Now, as we focus on the integral terms in equation $\mathcal{F}$ (3.51), it is important to convert the continuous formulation into a numerical form. With all the elementary FEM matrices we have introduced, we can rewrite each integral term in $\mathcal{F}$ as:

$$\begin{aligned}
\int_{\Omega^e} \frac{\partial \Phi^e}{\partial \xi} \frac{\partial \Phi^{eT}}{\partial \zeta} d\Omega &= \int_{\Omega^e} \begin{bmatrix} \frac{\partial_\xi \Phi^e}{\partial \xi} & \underline{0} & \underline{0} \\ \underline{0} & \frac{\partial_\xi \Phi^e}{\partial \xi} & \underline{0} \\ \underline{0} & \underline{0} & \frac{\partial_\xi \Phi^e}{\partial \xi} \end{bmatrix} \begin{bmatrix} \frac{\partial_\zeta \Phi^{eT}}{\partial \zeta} & \underline{0}^T & \underline{0}^T \\ \underline{0}^T & \frac{\partial_\zeta \Phi^{eT}}{\partial \zeta} & \underline{0}^T \\ \underline{0}^T & \underline{0}^T & \frac{\partial_\zeta \Phi^{eT}}{\partial \zeta} \end{bmatrix} d\Omega \\
&= \begin{bmatrix} \int_{\Omega^e} \frac{\partial_\xi \Phi^e \partial_\zeta \Phi^{eT}}{d\Omega} & \underline{00}^T & \underline{00}^T \\ \underline{00}^T & \int_{\Omega^e} \frac{\partial_\xi \Phi^e \partial_\zeta \Phi^{eT}}{d\Omega} & \underline{00}^T \\ \underline{00}^T & \underline{00}^T & \int_{\Omega^e} \frac{\partial_\xi \Phi^e \partial_\zeta \Phi^{eT}}{d\Omega} \end{bmatrix} \\
&= \begin{bmatrix} \mathcal{S}_{\xi\zeta} & \underline{00}^T & \underline{00}^T \\ \underline{00}^T & \mathcal{S}_{\xi\zeta} & \underline{00}^T \\ \underline{00}^T & \underline{00}^T & \mathcal{S}_{\xi\zeta} \end{bmatrix} = \mathbf{S}_{\xi\zeta}^e.
\end{aligned} \tag{3.53}$$

Here, $\mathbf{S}_{\xi\xi}^e$ represents the block diagonal stiffness matrix of $\mathcal{S}_{\xi\xi}^e$.

Now, we can substitute the numerical form of all the integral terms back into equation $\mathcal{F}$ to obtain the final expression of the Galerkin test on the non-conservative form:

$$\begin{aligned} & \omega \frac{\Delta x}{2} \mathbf{W}_x \mathbf{A}_x (\mathbf{S}_{xx}^e \underline{u}^e(t)) + \omega \frac{\Delta y}{2} \mathbf{W}_y \mathbf{A}_x (\mathbf{S}_{yx} \underline{u}^e(t)) + \omega \frac{\Delta x}{2} \mathbf{W}_x \mathbf{A}_y (\mathbf{S}_{xy}^e \underline{u}^e(t)) \\ & + \omega \frac{\Delta y}{2} \mathbf{W}_y \mathbf{A}_y (\mathbf{S}_{yy}^e \underline{u}^e(t)) - \omega \frac{\Delta x}{2} \mathbf{W}_x (\mathbf{S}_x^{Te} \underline{s}^e(t)) - \omega \frac{\Delta y}{2} \mathbf{W}_y (\mathbf{S}_y^{Te} \underline{s}^e(t)). \end{aligned} \quad (3.54)$$

Where $\mathbf{S}_x^{Te}$ and $\mathbf{S}_y^{Te}$ are the block diagonal matrices of the transformed stiffness matrices $\mathcal{S}_x^e$ and $\mathcal{S}_y^e$, respectively. These transformations arise due to the fact that we shift the derivative operator from the basis function to the test function for the last two integral terms on the left-hand side of the equation (3.51). Detailed results after matrix multiplications can be found in Appendix C.

Before combining equations (3.54) with equations (3.35) to form the complete system of ODEs implemented in CRISSP2D, we need to address how boundary conditions are applied within the model. This step will involve modifications to the final ODEs, ensuring that the boundary constraints are integrated correctly.

3.4. Boundary Conditions

Having discretized the governing shallow water PDEs and obtained a system of ODEs, we are now left with two essential steps: (a) incorporating initial and boundary conditions, and (b) numerically solving the resulting ODEs.

In this section, we describe how boundary conditions for both water surface elevation and discharge (or velocity) can be enforced within the FEM discretization framework. Starting with the choice of equations for each boundary condition, we will then detail how to

apply these boundary conditions using the corresponding equations, ensuring that the solution aligns with the physical properties of the model and the numerical methods employed.

3.4.1. Choice of Equations for Boundary Conditions

We have some flexibility regarding which equations to use for enforcing specific boundary conditions. In our approach, we will use the continuity equation (2.1) in Chapter 2:

$$\frac{\partial h}{\partial t} + \frac{\partial p}{\partial x} + \frac{\partial q}{\partial y} = 0$$

to enforce the flux boundary condition and the momentum equations (2.2)

$$\frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(\frac{p^2}{h}\right) + \frac{\partial}{\partial y}\left(\frac{pq}{h}\right) = \frac{\tau_{sx} - \tau_{bx}}{\rho} - gh\frac{\partial \eta}{\partial x} + \frac{1}{\rho}\left(\frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y}\right)$$

$$\frac{\partial q}{\partial t} + \frac{\partial}{\partial x}\left(\frac{pq}{h}\right) + \frac{\partial}{\partial y}\left(\frac{q^2}{h}\right) = \frac{\tau_{sy} - \tau_{by}}{\rho} - gh\frac{\partial \eta}{\partial y} + \frac{1}{\rho}\left(\frac{\partial T_{xy}}{\partial x} + \frac{\partial T_{yy}}{\partial y}\right)$$

to enforce the elevation boundary condition.

While both flux and elevation boundary conditions are Dirichlet boundary conditions, they differ in how we apply them in the numerical framework. In Julia implementation, the flux boundary condition, which specifies the normal flux ($\tilde{q}_n$) at the boundary, is applied through the continuity equation. Using the divergence theorem, we can manage the normal flux without explicitly calculating the normal direction at each boundary node or decomposing the user-defined boundary condition $\tilde{q}_n$ into p and q (the flux components in the x and y directions) at each boundary node. This approach contrasts with the original CRISSP2D Fortran code, which requires decomposing the normal direction for boundary flux at each

boundary node. By avoiding this decomposition, this method not only simplifies the process but also improves computational efficiency.

For the elevation boundary condition, which is applied through the momentum equations, no directional computation is required. Therefore, in each simulation step, before updating p and q , we can simply set the user defined elevation value at the boundary nodes. This approach simplifies the process of applying boundary conditions, reducing the need for complex direction-based calculations typically associated with the flux boundary condition and allows us to handle boundary conditions efficiently by leveraging the specific characteristics of different boundary conditions.

3.4.2. Flux Boundary Condition via the Continuity Equation

We will use the divergence theorem to apply the boundary conditions to the continuity equation, as this approach ensures mass conservation. This is achieved by relating the net discharge through the boundary to the rate of change in water depth within the domain.

Recall the continuity equation:

$$\frac{\partial h}{\partial t} + \frac{\partial p}{\partial x} + \frac{\partial q}{\partial y} = 0,$$

where h represents the water depth, and p and q are the components of the water flux in the x and y directions, respectively.

Let us define the flux vector $\vec{Q}$ as $\vec{Q} = [p , q]$, then the continuity equation can be written in vector form as:

$$\frac{\partial h}{\partial t} + \nabla \cdot \vec{Q} = 0.$$

To derive the weak form, we multiply the conservative form by a Galerkin testing function $\underline{\Phi}^e$ and integral over the element:

$$\int_{\Omega^e} \underline{\Phi}^e \left(\frac{\partial h}{\partial t} + \nabla \cdot \vec{Q} \right) d\Omega = \int_{\Omega^e} \underline{\Phi}^e \frac{\partial h}{\partial t} d\Omega + \int_{\Omega^e} \underline{\Phi}^e \nabla \cdot \vec{Q} d\Omega.$$

By applying the divergence theorem to the second term:

$$\int_{\Omega^e} \underline{\Phi}^e \nabla \cdot \vec{Q} d\Omega = \oint_{\Gamma_Q^e} \underline{\Phi}^e \vec{Q} \cdot \hat{n} d\Gamma - \int_{\Omega^e} \vec{Q} \cdot \nabla \underline{\Phi}^e d\Omega.$$

Substitute this back in, we obtain:

$$\int_{\Omega^e} \underline{\Phi}^e \frac{\partial h}{\partial t} + \underline{\Phi}^e \nabla \cdot \vec{Q} d\Omega = \int_{\Omega^e} \underline{\Phi}^e \frac{\partial h}{\partial t} d\Omega + \oint_{\Gamma_Q^e} \underline{\Phi}^e \vec{Q} \cdot \hat{n} d\Gamma - \int_{\Omega^e} \vec{Q} \cdot \nabla \underline{\Phi}^e d\Omega \quad (3.55)$$

Here, $\oint_{\Gamma_Q^e} \vec{Q} \cdot \hat{n} d\Gamma$ represents the total flux along the outward normal of the boundary l , also known as the **boundary flux**. This boundary flux term can be used to apply the boundary condition.

- if the boundary flux term is negative, there is a net water flow into the element through the boundary
- if the boundary flux term is positive, there is a net water flow out of the element through the boundary

Note that as long as the solution is continuous across the interface between adjacent elements, this boundary term will cancel for all interior elements. This cancellation occurs because a continuous solution results in continuous flux, Where the flux into one element matches the flux leaving the adjacent element, effectively canceling out across the shared interface,

because the normal direction flips between adjacent elements. Therefore, as long as our solution guarantees continuity along element edges (as our choice of FEM basis function does), these surface integral terms are only necessary at the domain boundary Γ_Q . At this outer boundary, no adjacent element exists to balance the flux, so the boundary conditions must be enforced.

The term $\partial_t h$ represents the rate of change of water depth with respect to time at a point within the domain. This shows that the rate of change in water depth corresponds directly to the net inflow or outflow of water across the boundaries, which reflects the conservation of mass.

In Equation (3.55), the boundary flux term $\oint_{\Gamma_Q^e} \vec{Q} \cdot \hat{n} d\Gamma$ can be treated using one of the following two approaches.

3.4.2.1. Basis Function Expansion of Nodal Normal Discharge $\tilde{q}_n$

If the normal discharge across the element edge q_n^e is represented using the basis functions and nodal values as:

$$\vec{Q} \cdot \hat{n} = q_n^e(\vec{r}, t) = \underline{\Phi}^{eT} \underline{\tilde{q}}_n(t),$$

where $\underline{\tilde{q}}_n(t)$ denotes the nodal normal flux values prescribed along the boundary. This leads to the following weak form:

$$\int_{\Omega^e} \underline{\Phi}^e \left(\frac{\partial h}{\partial t} + \nabla \cdot \vec{Q} \right) d\Omega = \mathcal{M}^e \frac{d}{dt} \underline{h}(t) - \mathcal{S}_x^{eT} \underline{p}(t) - \mathcal{S}_y^{eT} \underline{q}(t) + \oint_{\Gamma_Q^e} \underline{\Phi}^e \underline{\Phi}^{eT} d\Gamma \underline{\tilde{q}}_n(t). \quad (3.56)$$

The term $\oint_{\Gamma_Q^e} \underline{\Phi}^e \underline{\Phi}^{eT} d\Gamma \underline{\tilde{q}}_n(t)$ is referred to as the “weighted boundary flux”. By introducing $\oint_{\Gamma_Q^e} \underline{\Phi}^e \underline{\Phi}^{eT} d\Gamma$ as the mass matrix with respect to the boundary integral, denoted as $\mathcal{M}_F^e$, we

simplify the ODE of the integral for the residual with the boundary condition term for the continuity equation as:

$$\mathcal{M}^e \frac{d}{dt} \underline{h}(t) - \mathcal{S}_x^{eT} \underline{p}(t) - \mathcal{S}_y^{eT} \underline{q}(t) + \mathcal{M}_\Gamma^e \underline{\tilde{q}}_n(t). \quad (3.57)$$

A detailed derivation of Equation (3.57), including the application of the divergence theorem and Galerkin testing for the continuity equation, is provided in Appendix D.

3.4.2.2. Constant Edge Flux q_n^e

If the normal discharge across the element edge $q_n^e(t)$ is treated as a constant value along the boundary (i.e., not expanded using basis functions), then after applying Galerkin testing to Equation 3.55, the weak form becomes:

$$\int_{\Omega^e} \underline{\Phi}^e \left(\frac{\partial h}{\partial t} + \nabla \cdot \vec{Q} \right), d\Omega = \mathcal{M}^e \frac{d}{dt} \underline{h}(t) - \mathcal{S}_x^{eT} \underline{p}(t) - \mathcal{S}_y^{eT} \underline{q}(t) + q_n^e(t) \oint_{\Gamma_Q^e} \underline{\Phi}^e d\Gamma. \quad (3.58)$$

Here, $q_n^e(t)$ represents the constant normal discharge applied uniformly along the boundary edge Γ^e , and $\oint_{\Gamma_Q^e} \underline{\Phi}^e d\Gamma$ acts as the weighting vector associated with the test functions over the boundary. This constant-flux approach is particularly convenient when the total normal discharge q_n^e across the element edge is known, such as along a flux boundary where inflow or outflow is explicitly specified.

Although this term is not included directly in the final ODE system in this thesis, it provides a practical way to test and validate boundary behavior when edge-wise flux is available and the current implementation in the Julia code base supports both approaches.

3.4.3. Elevation Boundary Condition with Momentum Equations

While the above equation allows us to impose the flux boundary condition naturally through the mathematical process of integration by parts, the approach for setting the elevation boundary condition is more straightforward. Specifically, we apply the elevation boundary conditions directly to the elevation boundary nodes. Recall the elevation boundary update equation introduced in Chapter 2:

$$\eta|_{\Gamma_L} = c, t > 0,$$

where c represents the user-defined elevation boundary condition. By prescribing the boundary elevation, the system automatically adjust flow rates based on the defined boundary elevations.

The overall simulation process starts by updating the water surface elevation η using the continuity equation, incorporating the user-defined flux boundary condition. Following this update, the elevation boundary condition is set before proceeding to update the flux components. Finally, update the flux components p and q using the momentum equations. This sequencing aligns with the requirements of the leap-frog time-stepping scheme employed in the model. A complete description of the program flow is coming in Section 3.6.2.

3.5. System of ODEs

As mentioned at the end of Section 3.3.2, after applying the Galerkin testing procedure to both the conservative and non-conservative forms of the hydrodynamic equations, we arrive at the point where these formulations can be combined into a final system of ordinary differential equations (ODEs). This sets the stage for addressing the last step (b): numerically solving the ODEs, which will be covered in the following section.

The full expression for the integral of the residual, using the SUPG testing method, is given as follows:

$$\int_{\Omega^e} \Phi^e \left(\frac{\partial \tilde{\underline{U}}^e}{\partial t} + \nabla \cdot \tilde{\underline{E}}^e(\tilde{\underline{U}}^e) - \tilde{\underline{S}}^e(\tilde{\underline{U}}^e) \right) d\Omega + \int_{\Omega^e} \Phi'^e(A(\tilde{\underline{U}}^e) \cdot \nabla \tilde{\underline{U}}^e - \underline{S}(\tilde{\underline{U}}^e)) d\Omega = \underline{0}. \quad (3.59)$$

In the full expression for the integral of the residual, the first integral represents the application of the Galerkin test to the conservative form of the hydrodynamic equations. This term can be transformed into its numerical form as:

$$\int_{\Omega^e} \Phi^e \left(\frac{\partial \tilde{\underline{U}}^e}{\partial t} + \nabla \cdot \underline{E}(\tilde{\underline{U}}^e) - \tilde{\underline{S}}(\tilde{\underline{U}}^e) \right) d\Omega = \mathbf{M}^e \partial_t \underline{u}^e(t) + \mathbf{S}_x^e \underline{f}^e(t) + \mathbf{S}_y^e \underline{g}^e(t) - \mathbf{M}^e \underline{s}^e(t).$$

The first integral can be further aligned with the boundary condition applied to the continuity equation. Specifically, the Galerkin test in the conservative form integrates with the flux boundary condition set on the continuity equation, defined as:

$$\int_{\Omega^e} \Phi^e \left(\frac{\partial \tilde{u}_1^e}{\partial t} + \frac{\partial \tilde{f}_1^e}{\partial x} + \frac{\partial \tilde{g}_1^e}{\partial y} \right) d\Omega = \mathcal{M}^e \frac{d}{dt} \underline{u}_1(t) + \mathcal{M}_{\Gamma}^e \tilde{\underline{q}}_n - \mathcal{S}_x^{eT} \underline{f}_1^e(t) - \mathcal{S}_y^{eT} \underline{g}_1^e(t).$$

The second integral corresponds to the application of the Galerkin test to the non-conservative form of the hydrodynamic equations, represented in its numerical form as follows:

$$\begin{aligned} & \int_{\Omega^e} \Phi'(A(\tilde{\underline{U}}^e) \cdot \nabla \tilde{\underline{U}}^e - \tilde{\underline{S}}(\tilde{\underline{U}}^e)) d\Omega \\ &= \omega \frac{\Delta x}{2} \mathbf{W}_x \mathbf{A}_x (\mathbf{S}_{xx}^e \underline{u}^e(t)) + \omega \frac{\Delta y}{2} \mathbf{W}_y \mathbf{A}_x (\mathbf{S}_{yx}^e \underline{u}^e(t)) + \omega \frac{\Delta x}{2} \mathbf{W}_x \mathbf{A}_y (\mathbf{S}_{xy}^e \underline{u}^e(t)) \\ & \quad + \omega \frac{\Delta y}{2} \mathbf{W}_y \mathbf{A}_y (\mathbf{S}_{yy}^e \underline{u}^e(t)) - \omega \frac{\Delta x}{2} \mathbf{W}_x (\mathbf{S}_x^{\mathbf{T}^e} \underline{s}^e(t)) - \omega \frac{\Delta y}{2} \mathbf{W}_y (\mathbf{S}_y^{\mathbf{T}^e} \underline{s}^e(t)). \end{aligned}$$

By combining these two integrals (Equation (3.35) and (3.54)) with the flux boundary condition (Equation (3.57)) and isolating only the unknown terms with the time derivative ($du^e(t)/dt$) on the left side, we obtain the final system of ODEs:

$$\begin{aligned}
\mathbf{M}^e \frac{d}{dt} \underline{u}^e(t) = & - \begin{bmatrix} -\mathcal{S}_x^{Te} & 0 & 0 \\ 0 & \mathcal{S}_x^e & 0 \\ 0 & 0 & \mathcal{S}_x^e \end{bmatrix} \underline{f}^e(t) - \begin{bmatrix} -\mathcal{S}_y^{Te} & 0 & 0 \\ 0 & \mathcal{S}_y^e & 0 \\ 0 & 0 & \mathcal{S}_y^e \end{bmatrix} \underline{g}^e(t) \\
& + \begin{bmatrix} -\mathcal{M}_\Gamma^e & 0 & 0 \\ 0 & \mathcal{M}^e & 0 \\ 0 & 0 & \mathcal{M}^e \end{bmatrix} \begin{bmatrix} \tilde{q}_n^e(t) \\ \underline{s}_2^e(t) \\ \underline{s}_3^e(t) \end{bmatrix} \\
& - \omega \frac{\Delta x}{2} \mathbf{W}_x \mathbf{A}_x(\mathbf{S}_{xx}^e \underline{u}^e(t)) - \omega \frac{\Delta y}{2} \mathbf{W}_y \mathbf{A}_x(\mathbf{S}_{yx}^e \underline{u}^e(t)) \\
& - \omega \frac{\Delta x}{2} \mathbf{W}_x \mathbf{A}_y(\mathbf{S}_{xy}^e \underline{u}^e(t)) - \omega \frac{\Delta y}{2} \mathbf{W}_y \mathbf{A}_y(\mathbf{S}_{yy}^e \underline{u}^e(t)) \\
& + \omega \frac{\Delta x}{2} \mathbf{W}_x(\mathbf{S}_x^{Te} \underline{s}^e(t)) + \omega \frac{\Delta y}{2} \mathbf{W}_y(\mathbf{S}_y^{Te} \underline{s}^e(t)). \tag{3.60}
\end{aligned}$$

This system can be solved marching on time at each time step to update the unknown quantities (η , p and q) as the simulation progresses.

If we define the right-hand side of Equation (3.60) as

$$\underline{\mathcal{F}}^e = \begin{bmatrix} \underline{\mathcal{F}}_e^h \\ \underline{\mathcal{F}}_e^p \\ \underline{\mathcal{F}}_e^q \end{bmatrix},$$

The final ODEs for a single-element mesh can be written as:

$$\frac{d}{dt} \underline{u}^e(t) = \mathbf{M}^{e-1} \underline{\mathcal{F}}^e(\underline{u}^e(t)). \tag{3.61}$$

However, to model the entire domain with multiple elements, we must solve for all nodes in the mesh simultaneously using a global mass matrix, $\mathbf{M}_g$ and global $\underline{\mathcal{F}}$ term, $\underline{\mathcal{F}}_g$. In FEM, each node is influenced by its adjacent nodes, so constructing the ODE system for the full domain requires assembling partial contributions into global matrices and vectors, where each node is coupled with its neighbors to ensure continuity across elements. The global $\underline{\mathcal{F}}$ is assembled by summing contributions at shared vertices across all elements. Similarly, the global mass matrix ($\mathbf{M}_g$) is constructed by mapping each partial element's mass matrix to the corresponding global node indices. This is achieved by adding each element's mass matrix contribution to the appropriate entries in the global matrix. Since a lumped mass matrix is used, in which only the diagonal entries are retained, $\mathbf{M}_g$ is a diagonal matrix. With the global form of the mass matrix $\mathbf{M}_g$ and $\underline{\mathcal{F}}_g$ and the unknowns $\underline{u}(t)$ at each global vertex, we can formulate the final system of ODEs representing the entire domain as:

$$\frac{d}{dt}\underline{u}(t) = \mathbf{M}_g^{-1}\underline{\mathcal{F}}_g(\underline{u}^e(t)), \quad (3.62)$$

where we can solve for the unknown quantities at each global node index simultaneously. The detail of how to approximate the time derivative d/dt will be covered in the next section.

3.6. Solution Scheme

When we look at the ODEs obtained from the Galerkin testing:

$$\frac{d}{dt}\underline{u}(t) = \mathbf{M}_g^{-1}\underline{\mathcal{F}}_g(\underline{u}^e(t))$$

it is important to note that

$$d\eta/dt = dh/dt = \mathbf{M}_g^{-1} \mathcal{F}_g(h),$$

since $\eta(x, y, t) = h(x, y, t) + Z_0(x, y)$. Therefore, we can express the time derivative of the unknowns as:

$$\frac{d}{dt} \underline{u}(t) = \frac{d}{dt} \begin{bmatrix} \eta \\ p \\ q \end{bmatrix} = \mathbf{M}_g^{-1} \underline{\mathcal{F}}_g \left(\begin{bmatrix} h \\ p \\ q \end{bmatrix} \right).$$

In the following section, we will explore how to numerically solve the ODEs using the leap-frog explicit time integration scheme. Afterward, we will conclude this Chapter with a pseudo code outline that demonstrates the sequential flow of updating these unknowns in a serial code. This serial approach will set the stage for a parallelized implementation in Julia, which will be covered in the following chapters.

3.6.1. Leap frog Explicit Time Integration Scheme

In order to perform time integration, one approach is to discretize continuous time t into a series of discrete time points $\{0, \Delta t, 2\Delta t, \dots, n\Delta t\}$, where n represents the n^{th} time step or time index. The solution at the n^{th} time step can be expressed as $\underline{u}(t) = \underline{u}(n\Delta t) = \underline{u}^n$, where u^0 represents the known initial conditions prescribed by the problem, as introduced in Section 2.2.2. This allows us to integrate the Equation (3.62) forward in time.

CRISSP2D uses a modified explicit time-integration technique (Euler forward integration) to approximate the time derivative:

$$\frac{d}{dt} \underline{u}(t) \approx \frac{\underline{u}^{n+1} - \underline{u}^n}{\Delta t} = \mathbf{M}_g^{-1} \underline{\mathcal{F}}_g(\underline{u}^n).$$

However, in CRISSP2D, the unknowns η (or h) and the flux components p and q are updated asynchronously using a leapfrog method.

- Flux component p and q : These are discretized at integer time steps $n = 0, 1, 2, \dots$
- Water Surface Elevation η : This is updated at half-integer time steps $n = 0.5, 1.5, 2, \dots$

This leads to the half-step leapfrog time integration scheme, where flux components and water surface elevation are updated alternately at staggered time intervals:

$$\begin{aligned}\frac{p^{n+1} - p^n}{\Delta t} &= \mathbf{M}_g^{-1} \underline{\mathcal{F}}_g^p(p^n, q^n, h^{n+0.5}) \\ \frac{q^{n+1} - q^n}{\Delta t} &= \mathbf{M}_g^{-1} \underline{\mathcal{F}}_g^q(p^n, q^n, h^{n+0.5}) \\ \frac{\eta^{n+1.5} - \eta^{n+0.5}}{\Delta t} &= \mathbf{M}_g^{-1} \underline{\mathcal{F}}_g^h(p^{n+1}, q^{n+1}, h^{n+0.5}).\end{aligned}\tag{3.63}$$

By leaving the unknowns we want to solve for in ODEs (i.e. p^{n+1}, q^{n+1} and $\eta^{n+0.5}$) on the left-hand side of the above equations, we arrive at the explicit time integration formulas for updating p , q and η :

$$\begin{aligned}p^{n+1} &= \mathbf{M}_g^{-1} \underline{\mathcal{F}}_g^p(p^n, q^n, h^{n+0.5}) \Delta t + p^n \\ q^{n+1} &= \mathbf{M}_g^{-1} \underline{\mathcal{F}}_g^q(p^n, q^n, h^{n+0.5}) \Delta t + q^n \\ \eta^{n+1.5} &= \mathbf{M}_g^{-1} \underline{\mathcal{F}}_g^h(p^{n+1}, q^{n+1}, h^{n+0.5}) \Delta t + \eta^{n+0.5}.\end{aligned}\tag{3.64}$$

To ensure the stability of the leapfrog method, the selected time step must be smaller than the critical time step [11]

$$\Delta t_{\text{critical}} = 1.5 \frac{\Delta L}{\sqrt{2 g h_{\max}}},\tag{3.65}$$

where

- ΔL : represents the minimum edge length among all triangular elements.
- h_{max} : represents the maximum water depth among the entire domain.

3.6.2. Serial Algorithm

After gathering all the necessary components to implement the serial algorithm for updating the unknown quantities of the flow, we conclude this Chapter with a serial pseudo code. This code can be implemented in a programming language like Julia and provides a foundation for further optimization and parallelization, which will be discussed in the following chapters.

Note that, For computational efficiency, we choose to store the nodal variables u and v explicitly. However, these variables are not necessary, as they can be directly calculated from the nodal values of p and q whenever needed.

Algorithm 1 Hydrodynamic FEM flow.

- 1: **procedure** HYDRODYNAMIC FEM
 - 2: **Preprocessing**
 - 3: Load msh file with Gmsh
 - 4: Create mesh object(including nodes, and elements, and connectivity)
 - 5: Construct elementary FEM matrices (Stiffness and Mass Matrices), boundary volumetric matrices($\mathcal{M}_\Gamma$) and global Mass Matrix($\mathbf{M}_g$)
 - 6: Initialize nodal properties: p, q, h, u, v
 - 7: **Processing**
 - 8: call Hydro(args)
 - 9: **end procedure**
-

Algorithm 2 HYDRO

```
1: function HYDRO(args)
2:   for each time step  $n$  do
3:     1. Assemble  $\underline{\mathcal{F}}_g^{p,q}(p^n, q^n, h^{n+0.5})$ 
4:     2. Update  $p, q$  by  $p^{n+1}, q^{n+1} = \mathbf{M}_g^{-1} \underline{\mathcal{F}}_g^{p,q}(p^n, q^n, h^{n+0.5}) \Delta t + p^n, q^n$ 
5:     3. Update  $\eta^{n+1.5}$  in the similar way with boundary conditions for  $p$  and  $q$  auto-
       matically applied
6:     4. Apply boundary conditions to boundary nodes of  $\eta^{n+1.5}$ 
7:   end for
8: end function
```

4. Test Cases

This Chapter presents two numerical test cases to validate the performance and behavior of the Julia implementation, specifically focusing on the integral by parts approach for applying boundary conditions. These tests are designed to evaluate the accuracy, stability, and robustness of the hydrodynamic model under various scenarios. The hydraulic parameters and constants used for testing, including the Manning’s roughness coefficient (set to 0.025), are specified in a user-defined HDF file and kept consistent across all software involved in the comparisons.

A grid independence test was not conducted in this thesis, as the primary objective was to validate the Julia implementation itself. However, it should be noted that grid independence testing is an essential step in practical model development for real-world applications to ensure the reliability and accuracy of simulation results. Finally, computational efficiency and simulation time are not discussed in this chapter, but are addressed separately in the following chapter.

The first test examines a river with a simple rectangular cross section, simulated using both hot-start and cold-start initialization options. This test serves as a baseline for verifying the accuracy of the model’s implementation and initialization processes. Steady-state hydraulic results for both cold and hot start cases are compared against outputs from the CRISSP2D Fortran implementation or HEC-RAS [5], depending on the specific test scenario. HEC-RAS is a one-dimensional river modeling tool that uses the finite-difference method (FDM) to simulate steady and unsteady flow conditions. Although its results provide a useful reference for evaluating the Julia implementation, differences are expected due to

the use of different numerical methods and model dimensionality. Therefore, HEC-RAS results serve only as practical benchmarks for comparison, and any differences between the HEC-RAS and Julia implementations are not considered errors, as the HEC-RAS results are numerical rather than analytical solutions.

The second test evaluates a river with a trapezoidal cross section, introducing the additional challenge of handling wet and dry nodes. This test is designed to validate the model's capability to accurately capture transitions between wet and dry states.

4.1. Subcritical Flow In a Rectangular River Channel

This test simulates a simple scenario: subcritical flow through a rectangular cross-section river. It avoids complexities such as transitions between wet and dry states or other complicated hydraulic situations. The primary goal of this test is to validate the accuracy and consistency of the core FEM algorithm for solving the shallow water equations.

The initial river model is designed as 2000 m in length and 500 m in width. The riverbed elevation decreases linearly from 0 m at the upstream boundary to -10 m at the downstream boundary, with a slope of -0.005.

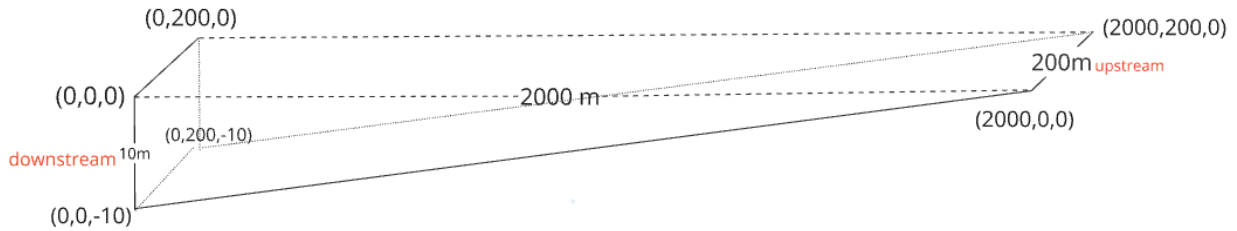


Figure 4.1. 3D rectangular cross geometric river model: structural details

Based on this river model, a two-dimensional finite element mesh was generated using SMS. With an element edge length of 1 m, the mesh consists of 4000 uniform elements and 2111 nodes, resulting in 2111×3 primary unknowns: h, p and q at each node.

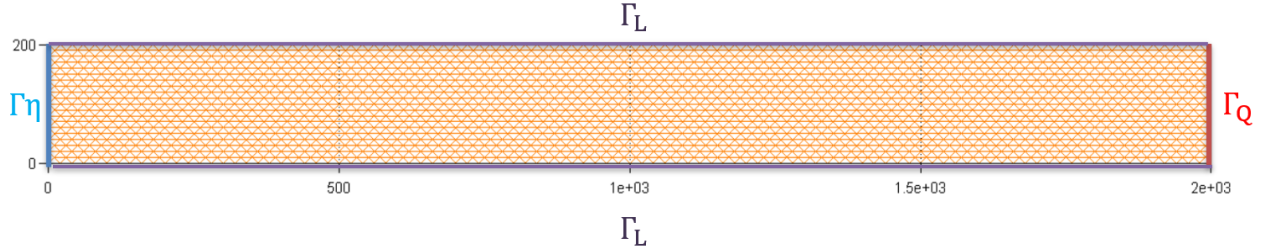


Figure 4.2. Rectangular mesh of the computational river domain. The left boundary (Γ_η) in blue denotes the elevation boundary, and the right boundary (Γ_Q) in red denotes the flux boundary. The top and bottom boundaries are treated as land boundaries (Γ_L), shown in purple. The element distribution is uniform throughout the domain.

4.1.1. Cold Start

The initial conditions for the cold start are as follows: the discharge components p and q are set to $0.0 \text{ m}^2/\text{s}$, and the water surface elevation η equals to 0.0 m throughout the entire river domain, as described in Section 2.2.2.1. The time step (Δt) that is chosen for the cold start situation is 0.2 s , which is smaller than the critical time step ($C_r \approx 1.516 \text{ s}$).

The elevation BCs at the downstream end of the river are assumed to be uniform across all elevation nodes ($\eta|_{\Gamma_\eta} = c, t > 0$). Its value decreases from 0 m to -1.5 m over 8 hours and then held constant for the next 5 hours, meaning the water surface at the downstream boundary will be 1.5 m lower than its initial state by the end of the simulation. The detailed boundary settings for each hour are depicted in Figure 4.3.

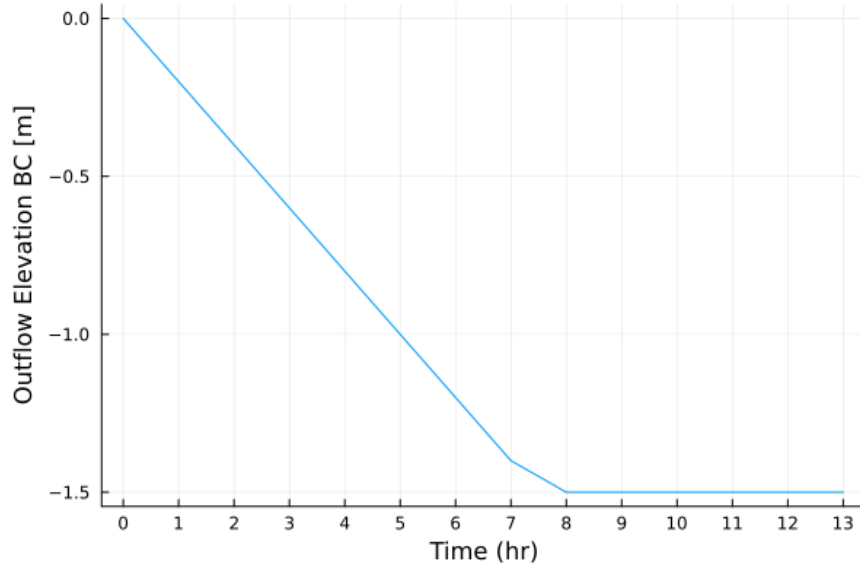


Figure 4.3. Elevation boundary condition over 13 hours.

The total inflow flux, denoted as Q_b (in units of $[\text{m}^3/\text{s}]$), is used to define the flux boundary condition at the upstream end. In the implementation, this total discharge is distributed across the inflow boundary nodes as nodal normal discharge values, q_n (in units of $[\text{m}^2/\text{s}]$), weighted according to the local water depth at each node. The absolute value of the inflow discharge increases from $20 \text{ m}^3/\text{s}$ to $160 \text{ m}^3/\text{s}$ over the same time period during which the elevation boundary condition is applied. Detailed boundary settings for each hour are presented in Figure 4.4.

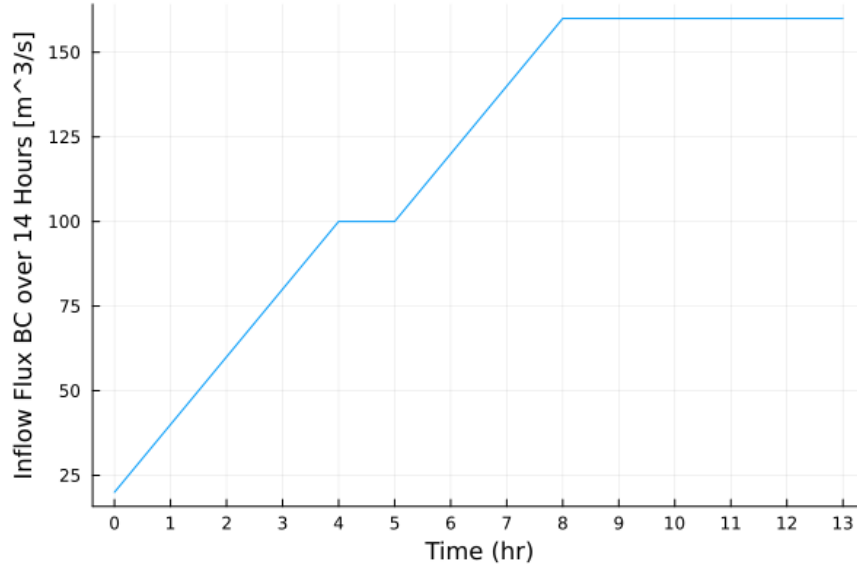


Figure 4.4. Flux boundary condition over 13 hours.

4.1.1.1. Cold Start Simulation Results

The Julia implementation achieved steady state after 149,601 steps, based on a convergence criterion ($\epsilon_{\text{tol}} = 10^{-4}$)¹. Under the selected time step and current model configuration, simulating 1 hour of shallow water behavior took 8.9 minutes of computational time. Conservation of mass can be verified by comparing the total inflow and outflow discharges, which should ideally be equal. In this case, the total discharge at the inflow boundary is 160.00 m³/s, while the total discharge at the outflow boundary is 160.13 m³/s, resulting in a difference of 0.08%. A computational hardware comparison between the Fortran and Julia implementations, using 500 iterations, demonstrates that Julia achieves comparable speed to Fortran. Specifically, Julia required 18.42 seconds for 500 iterations, while Fortran required

¹The simulation is considered to have reached steady state when $\epsilon_{r,h} = \frac{\|h^n - h^{n-1}\|}{\|h^n\|} < \epsilon_{\text{tol}}$

19.01 seconds. These results verify that the Julia implementation can achieve computational performance similar to that of Fortran.

Figure 4.5 compares the water depth along the x -axis centerline of the river with the results from the 1-D HEC-RAS model and the original Fortran implementation, using the same boundary conditions.

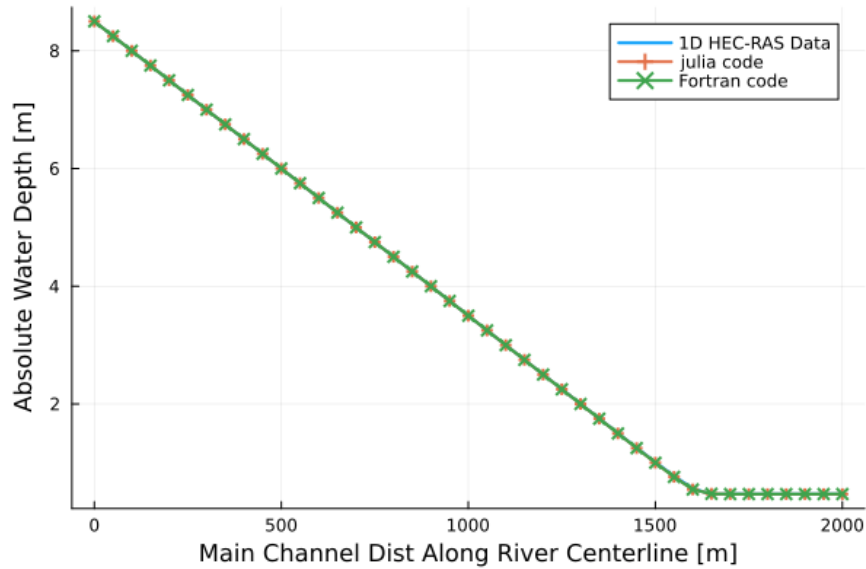


Figure 4.5. Water depth through the centerline of the river.

The x -axis represents the distance [m] from the downstream boundary along the centerline, while the y -axis shows the corresponding water depth [m]. Figure 4.6 presents the relative error in water depth, plotted on a logarithmic scale, where the logarithmic relative error is computed as $\log_{10}(|h_{\text{Julia}} - h_{\text{CRISSP2D}}|/|h_{\text{CRISSP2D}}|)$. Here, h_{Julia} and h_{CRISSP2D} denote the simulated water depth from the Julia and CRISSP2D implementations, respectively. This comparison highlights the agreement between the two models along the river centerline.

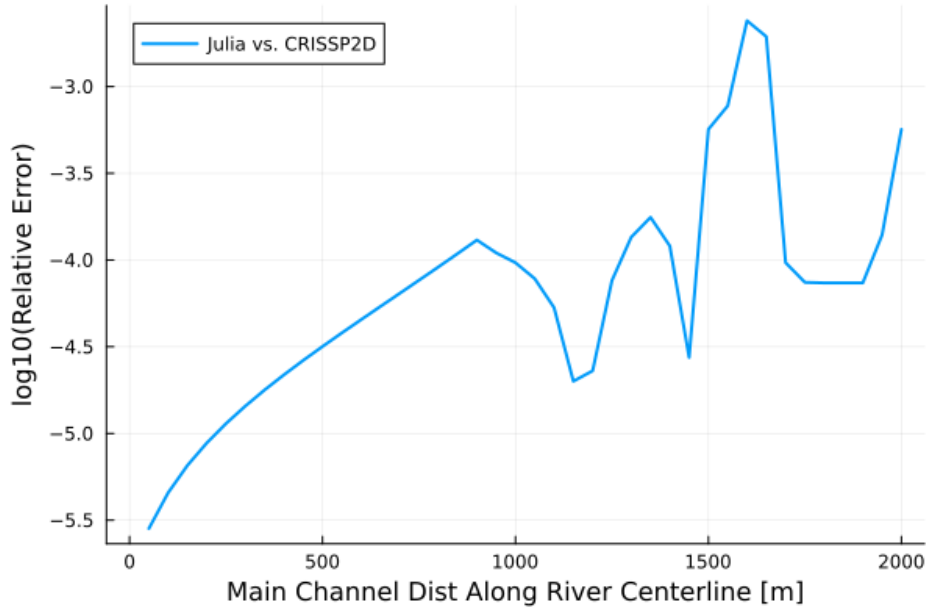


Figure 4.6. Logarithmic relative error of water depth h , along the river centerline. Results from the Julia implementations compared against CRISSP2D.

The majority of the relative error values lie below (10^{-3}) indicating that most of the domain maintains an accuracy better than 0.01%. The largest observed error is approximately $10^{-2.62}$, or about 0.24%, occurring near the downstream section around 1600 m. Notably, this location corresponds to a turning point in the water depth profile, as shown in Figure 4.5, which may contribute to a localized amplification of the relative error. Overall, the relative error plot demonstrates that the Julia results closely match the performance of the original Fortran code. Additionally, comparisons were performed between Julia and HEC-RAS, as well as between Fortran and HEC-RAS, for x-direction water velocity and water depth along the river centerline in the cold start case. The logarithmic error results for these comparisons are provided in Appendix H.

Figure 4.7 shows the absolute difference in water depth between the Julia implementation and the CRISSP2D Fortran results, evaluated across the two-dimensional river domain.

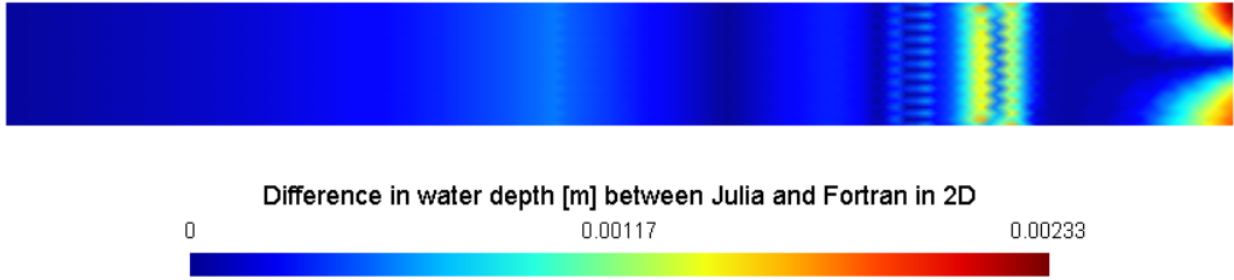


Figure 4.7. Absolute difference in water depth between the Julia implementation and the CRISSP2D Fortran results, evaluated across the 2D river domain.

Note that the maximum differences in water depth appear near the inflow boundary around $x = 2000$ m, likely due to differences in how boundary conditions are applied in the two implementations. In the Fortran code, both the flux and elevation boundaries are explicitly prescribed, whereas in the Julia implementation, the flux boundary is applied dynamically using the integral-by-parts method. As a result, the distribution of flux values across boundary nodes is not uniform, especially at the corner nodes, which are more likely to be influenced by the adjacent land boundary nodal values under the continuous FEM formulation.

A comparison of the profile of x -direction velocity along the river centerline is done for all three software implementations. As the river model represents a typical rectangular channel flowing from the positive to the negative x -direction, the vertical velocity component (v) remains close to zero throughout the domain. Since the river flows in x direction from right to left, and the y direction velocity component is approximately zero both along the

river centerline and throughout the domain, only the x direction velocity component (u) is included in the comparison.

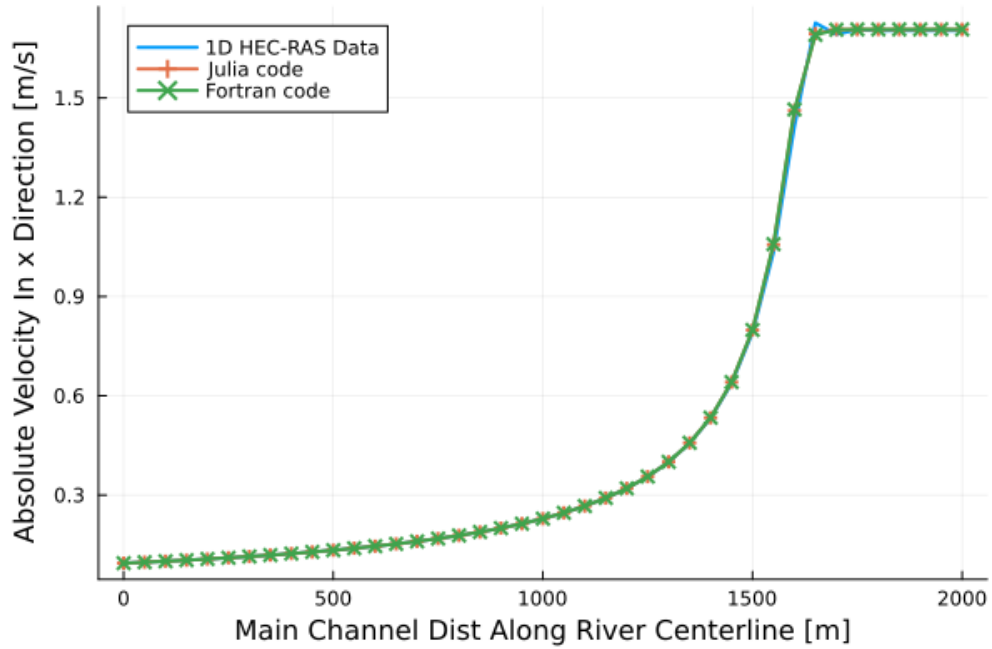


Figure 4.8. x -Direction velocity through the centerline of the river.

As shown, the Julia implementation closely matches the HEC-RAS and original Fortran implementation results at every profile position.

Similar to the water depth analysis, the relative error (in log scale) of the x -direction velocity component (u) between the three models is presented along the channel centerline in Figure 4.9.

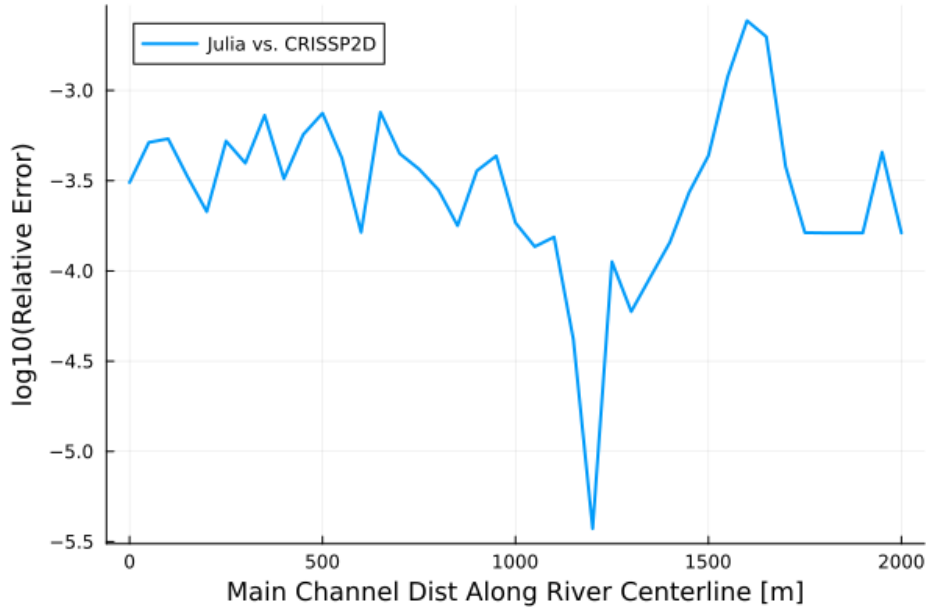


Figure 4.9. Logarithmic relative error of x -direction velocity, along the river streamline, comparing results from the Julia implementation and CRISSP2D.

As in the case of water velocity, the majority of the domain maintains a relative error below 10^{-3} , indicating accuracy better than 0.1% across most of the profile. The largest deviation occurs around $x = 1600$ m, where a turning point in the velocity profile is present as shown in Figure 4.8. At this location, the velocity rapidly increases, forming a steep gradient. The Julia implementation demonstrates strong overall agreement with CRISSP2D.

In addition, Figure 4.10 shows a two-dimensional map of the absolute velocity difference between the Julia and CRISSP2D (Fortran) results, illustrating the spatial variation across the full domain.

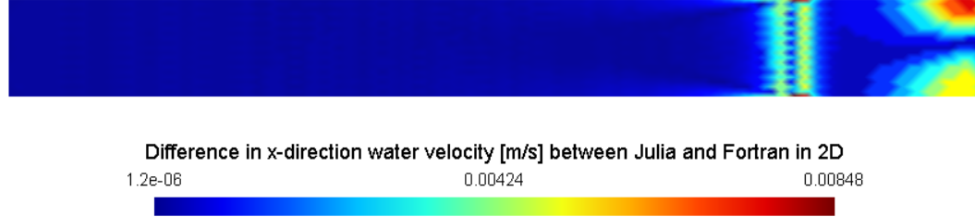


Figure 4.10. Absolute difference in x direction velocity between the Julia implementation and the CRISSP2D Fortran results, evaluated across the 2D river domain.

As shown in Figure 4.10, the spatial distribution of velocity differences between the Julia and Fortran implementations is similar to the water depth error pattern. The largest difference appear at the corner nodes near the inflow boundary, likely due to the same boundary treatment differences discussed previously. An additional band of elevated error occurs around $x = 1600$ m, which corresponds to the turning point in the velocity profile, as noted in the 1D velocity comparison. These localized differences do not significantly impact the overall solution accuracy, which remains consistent throughout the domain.

4.1.2. Hot Start

As described in Section 2.2.2.2, the initial conditions for h, p and q in the hot start are set to the final state of the last iteration from the cold start simulation. For the hot start, the same time step (0.2 s) is chosen, which is smaller than the critical time step ($C_r \approx 1.6$ s).

Similar to the initial conditions, the boundary conditions for the hot start also carry over from the final state of the cold start simulation. At the outflow boundary, the elevation value decreases from -1.5 m to -2.2 m over a period of 12 hours, as shown in Figure 4.11.

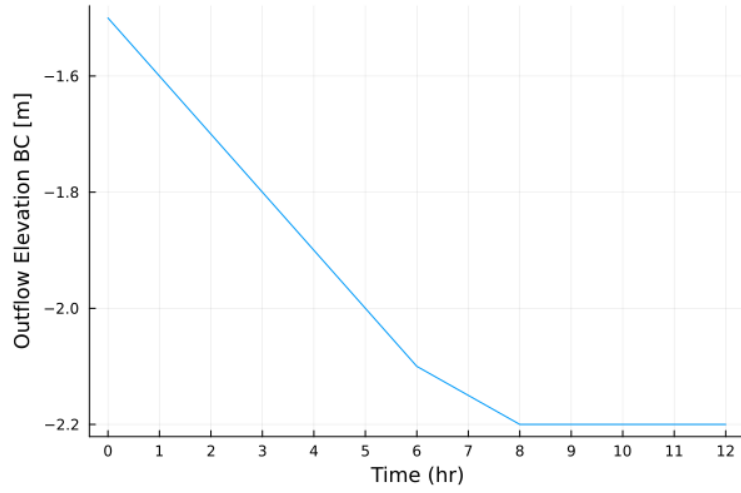


Figure 4.11. Elevation boundary condition over 12 hours.

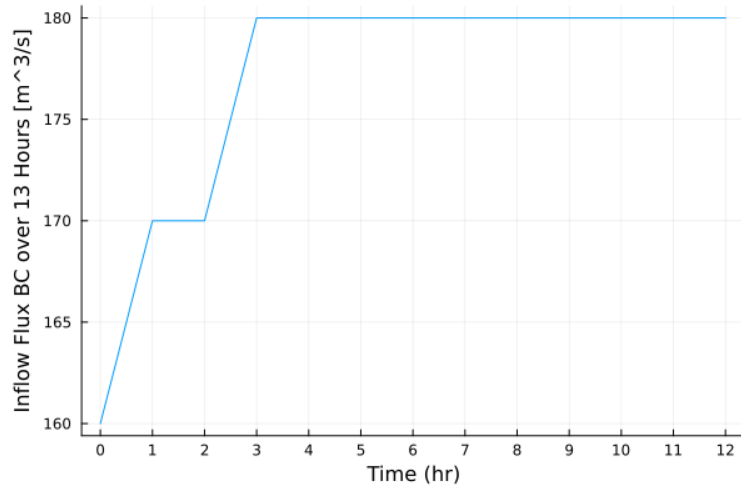


Figure 4.12. Flux boundary condition over 12 hours.

The total flux discharge at the inflow boundary increases hourly from 160 m³/s to 180 m³/s, as shown in Figure 4.12

4.1.2.1. Hot Start Simulation Result

The Julia implementation achieved steady state after 146,855 iterations, based on a convergence criterion of ($\epsilon_{\text{tol}} = 10^{-4}$). In the hot start case, the total discharge measured at the inflow boundary is approximately $180.00 \text{ m}^3/\text{s}$, while the total discharge at the outflow boundary is approximately $179.83 \text{ m}^3/\text{s}$, resulting in a difference of $0.17 \text{ m}^3/\text{s}$. Figure 4.13 compares the water depth along the x direction streamline of the river with the results from the 1-D HEC-RAS model and the original Fortran implementation, using the same boundary conditions.

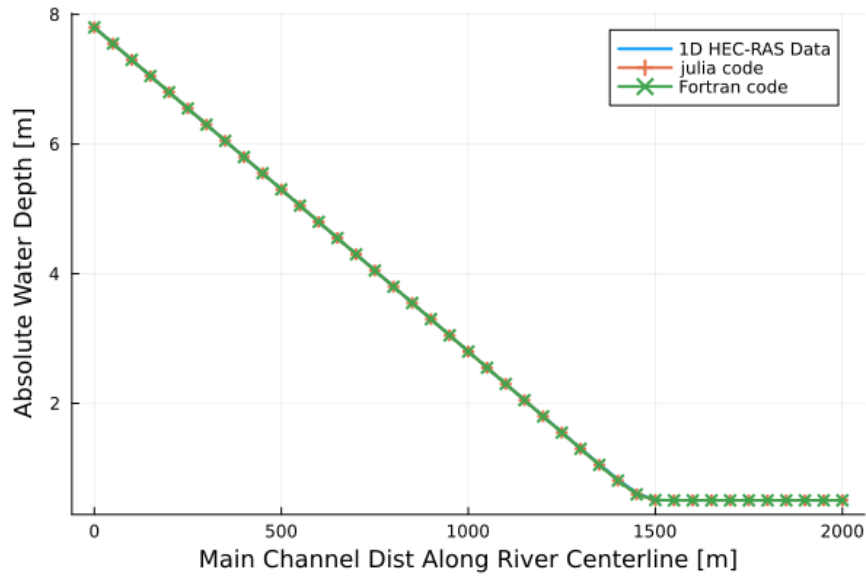


Figure 4.13. Water depth through the centerline of the river.

As shown in the figure, the water depth calculated by the Julia implementation matches closely the other two results. Figure 4.14 presents the relative error in water depth computed from the Julia compared against the CRISSP2D Fortran results.

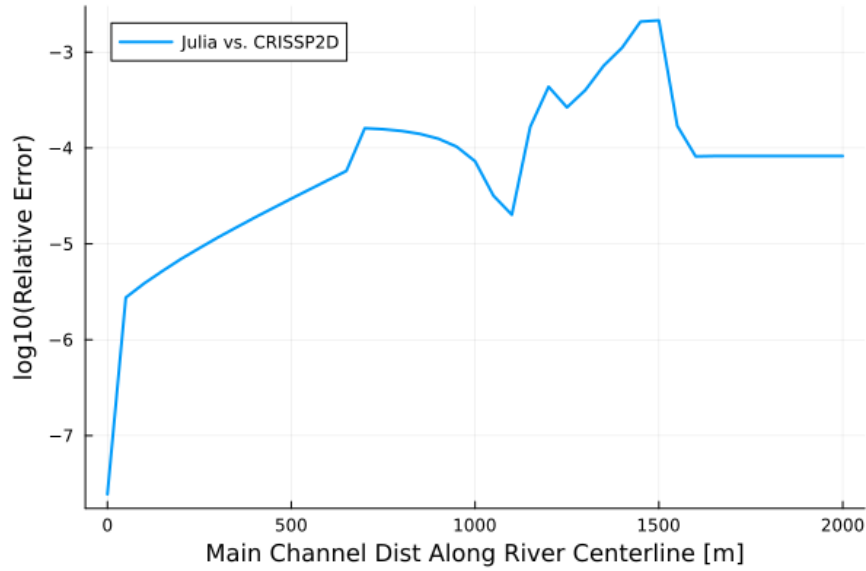


Figure 4.14. Logarithmic relative error of water depth (h) along the river centerline, comparing results from the Julia implementation and CRISSP2D.

Figure 4.14 shows the relative error in water depth along the river centerline for the hot start case. Most of the domain maintains an error below $10^{-2.67}$, indicating that most of the solution remains within 0.214% accuracy compared to CRISSP2D. The largest error appears around $x = 1600$ m, where a noticeable turning point in the water depth profile occurs. As discussed in the cold start case, this region exhibits a steep gradient that can amplify relative differences. Overall, the Julia implementations remain in close agreement with CRISSP2D throughout the domain.

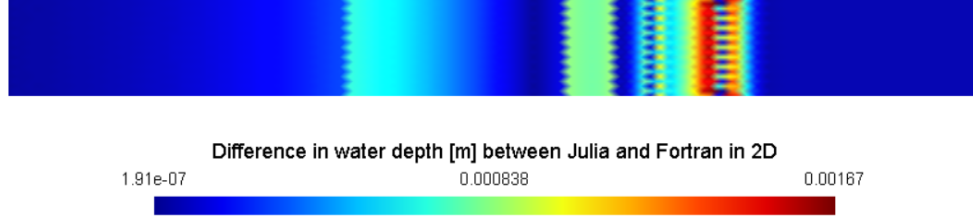


Figure 4.15. Absolute difference in water depth between the Julia implementation and the CRISSP2D Fortran results, evaluated across the 2D river domain.

Notably, both the 1D relative error comparison across all three implementations (Figure 4.14) and the 2D water depth difference between the Julia and Fortran results (Figure 4.15) reveal that the most significant deviation ($\sim 0.214\%$) occurs near the location approximately 1600 m upstream from the downstream boundary.

Figure 4.16 presents a comparison of the profile of x -direction velocity along the river flow, in the three software implementations.

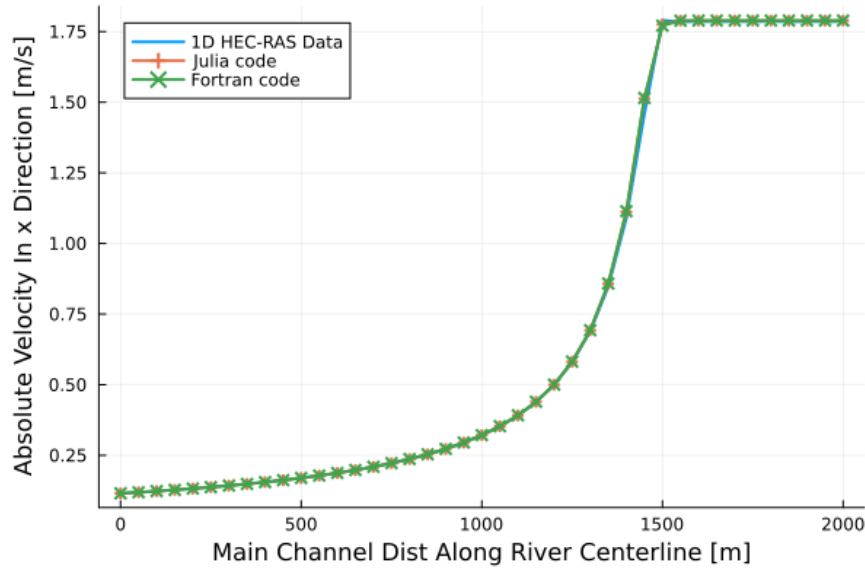


Figure 4.16. x -direction velocity through the centerline of the river.

The relative error in x -direction velocity (u) between the implementation of Julia and CRISSP2D, is stated in figure below.

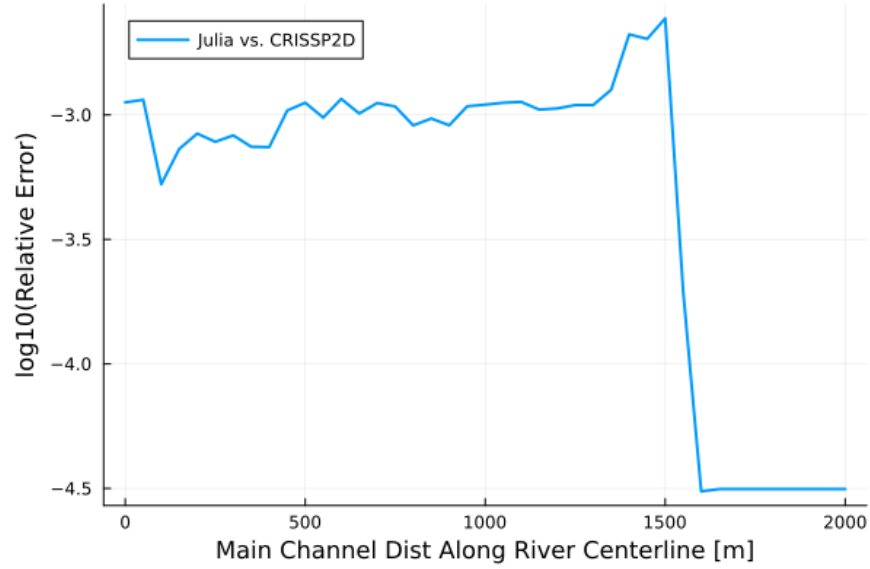


Figure 4.17. Logarithmic relative error of x -direction water velocity, along the river centerline. Results from the Julia is compared against CRISSP2D.

Similarly to the relative error of water depth, most of the relative errors in the x -direction velocity remain around 10^{-3} , indicating a good agreement with the implementation of CRISSP2D. The largest deviation occurs around $x = 1600$ m, corresponding to a turning point in the velocity profile where the flow accelerates rapidly. Both the Julia and Fortran implementations produce consistent results, with relative errors remaining low throughout the domain.

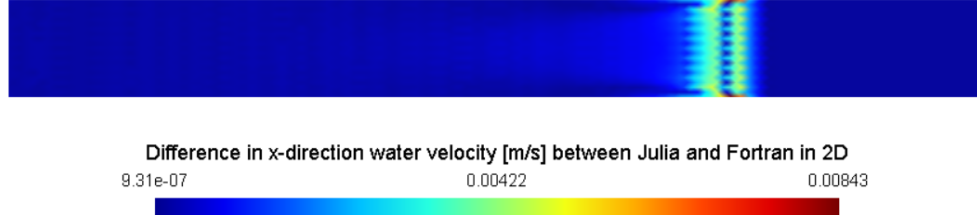


Figure 4.18. Absolute difference in x direction velocity between the Julia implementation and the CRISSP2D Fortran results, evaluated across the 2D river domain.

Similar to the water depth analysis, the velocity differences among all three implementations are primarily concentrated near the region of rapid change, approximately 1600 m upstream from the downstream boundary.

4.2. Subcritical Flow in a Trapezoidal River Channel

This test simulates subcritical flow in a trapezoidal channel and is divided into three stages. In the first stage, the downstream water surface is gradually lowered from an initially high level to below a defined horizontal reference surface. This drawdown is intended to trigger drying near the channel banks, where edge cells may become partially or fully dry. In the second stage, the inflow discharge is reduced over time while the downstream water surface continues to drop, causing additional nodes along the banks to dry out. At this point, the channel becomes partially dry. In the third stage, the inflow discharge is increased and the downstream water surface is raised, leading to the re-wetting of previously dry nodes. These three stages progressively increase the complexity of the test case, allowing evaluation of how well the model handles transitions between wet and dry conditions using the FEM framework with integration by parts.

The river model used in this test case is 2000 m long and 600 m wide, as shown in Figure 4.19.

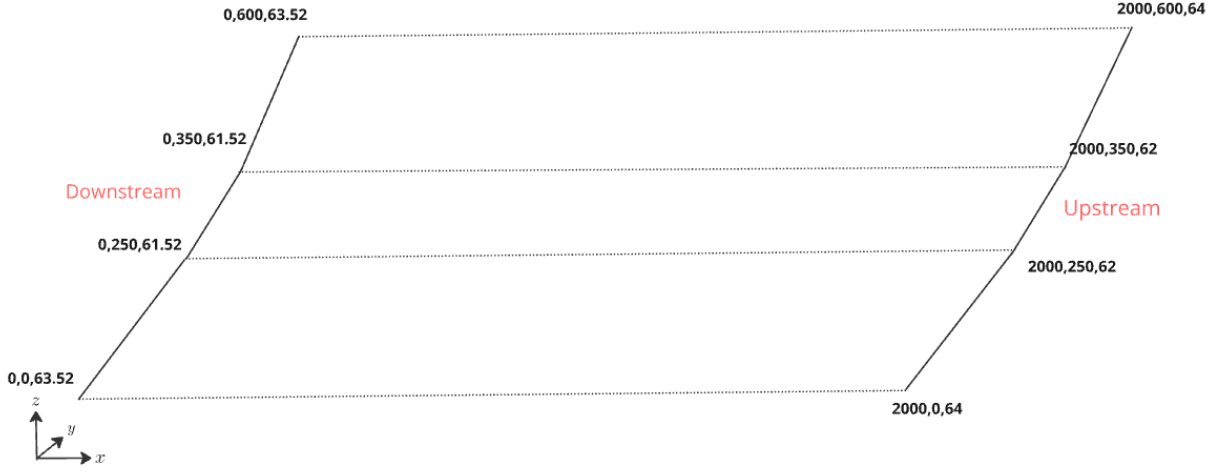


Figure 4.19. 3D rectangular cross sectional river model: Geometric details.

From a top-down view, the domain appears rectangular, but the cross-sectional shape introduces geometric complexity. The river channel has a trapezoidal cross section with a top width of 600 m, a bottom width of 100 m, and a vertical depth of 2 m. The channel extends longitudinally for 2000. The upstream and downstream ends are aligned along the x -direction, while the cross-sectional profiles are oriented in the y - z plane. Based on the river geometry, a two-dimensional, first-order (linear geometry) triangular mesh was generated using the SMS software.

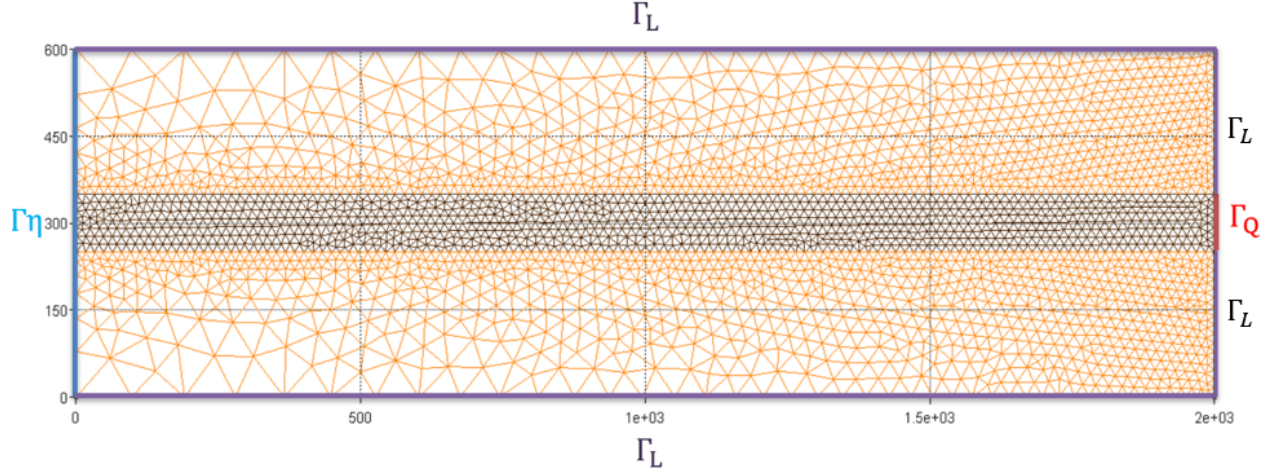


Figure 4.20. Unstructured trapezoidal mesh of the river domain. The left boundary (Γ_η , at $x = 0$, $y = 0 - 600$) in blue represents the elevation boundary. The right boundary (Γ_Q , at $x = 2000$, $y = 250 - 350$) in red defines the flux boundary. The remaining portions of the right boundary, along with the top and bottom boundaries, are treated as land boundaries (Γ_L as shown in purple). Element density is increased around the main channel and flux boundary.

As shown in Figure 4.20, the elevation boundary condition is applied across the full vertical extent of the outflow boundary (Γ_η), while the flux boundary condition is applied only within a narrower vertical band (Γ_Q) along the inflow boundary. The remaining segments of the right boundary are treated as land boundaries (Γ_L). This selective application of the flux boundary condition helps reduce uncertainty associated with arbitrary cross-sectional shapes and ensures that discharge is imposed only on wet nodes (i.e., where water depth $h > \text{a specified threshold}$). It also minimizes the risk of unrealistically high velocities or numerical instability that could occur if discharge were applied at nodes where $h \approx 0$, potentially leading to division by small values. As a result, this approach improves both the stability and physical realism of the simulation.

To support this configuration, the unstructured mesh is constructed with non-uniform element density. As shown in Figure 4.20, finer elements are concentrated around the main

channel region ($y = 250 - 350$ m) and near the inflow boundary. The increased resolution in the main channel allows for accurate representation of the narrow geometry and better captures steep gradients and dynamic flow behavior. Additionally, smaller element sizes near the inflow boundary help maintain flow symmetry and reduce the likelihood of directional bias, which can arise in asymmetric meshes. Overall, the mesh consists of 2,824 nodes and 5,500 elements.

4.2.1. Boundary Conditions

To simulate the dynamic behavior of subcritical flow and test the model's handling of wetting and drying transitions, time-varying boundary conditions were applied over a 30 hour period. These include a prescribed inflow discharge (Figure 4.22) at the upstream boundary and an elevation condition at the downstream boundary (Figure 4.21).

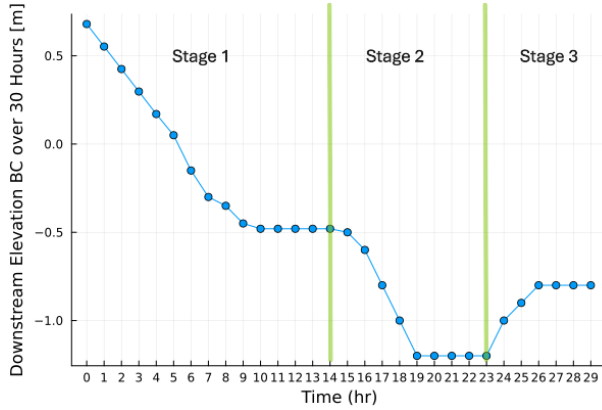


Figure 4.21. Elevation boundary condition over 29 hours.

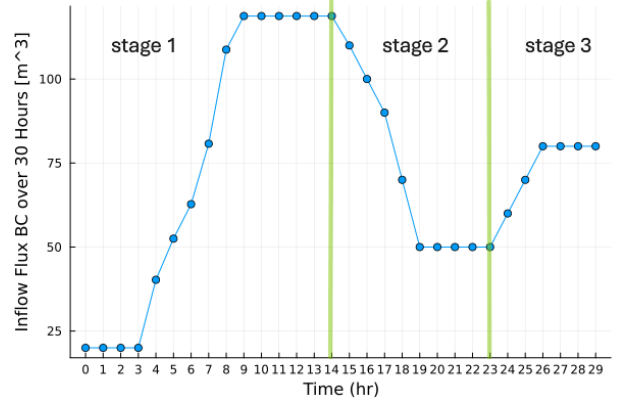


Figure 4.22. Flux boundary condition over 29 hours.

4.2.2. Simulation Result by Stage

With the computational mesh established and the time-varying boundary conditions defined, the simulation was run for a total duration of 30 hours. The results are analyzed in three stages corresponding to the prescribed boundary condition phases.

4.2.2.1. Stage 1 – Initial Draw-down:

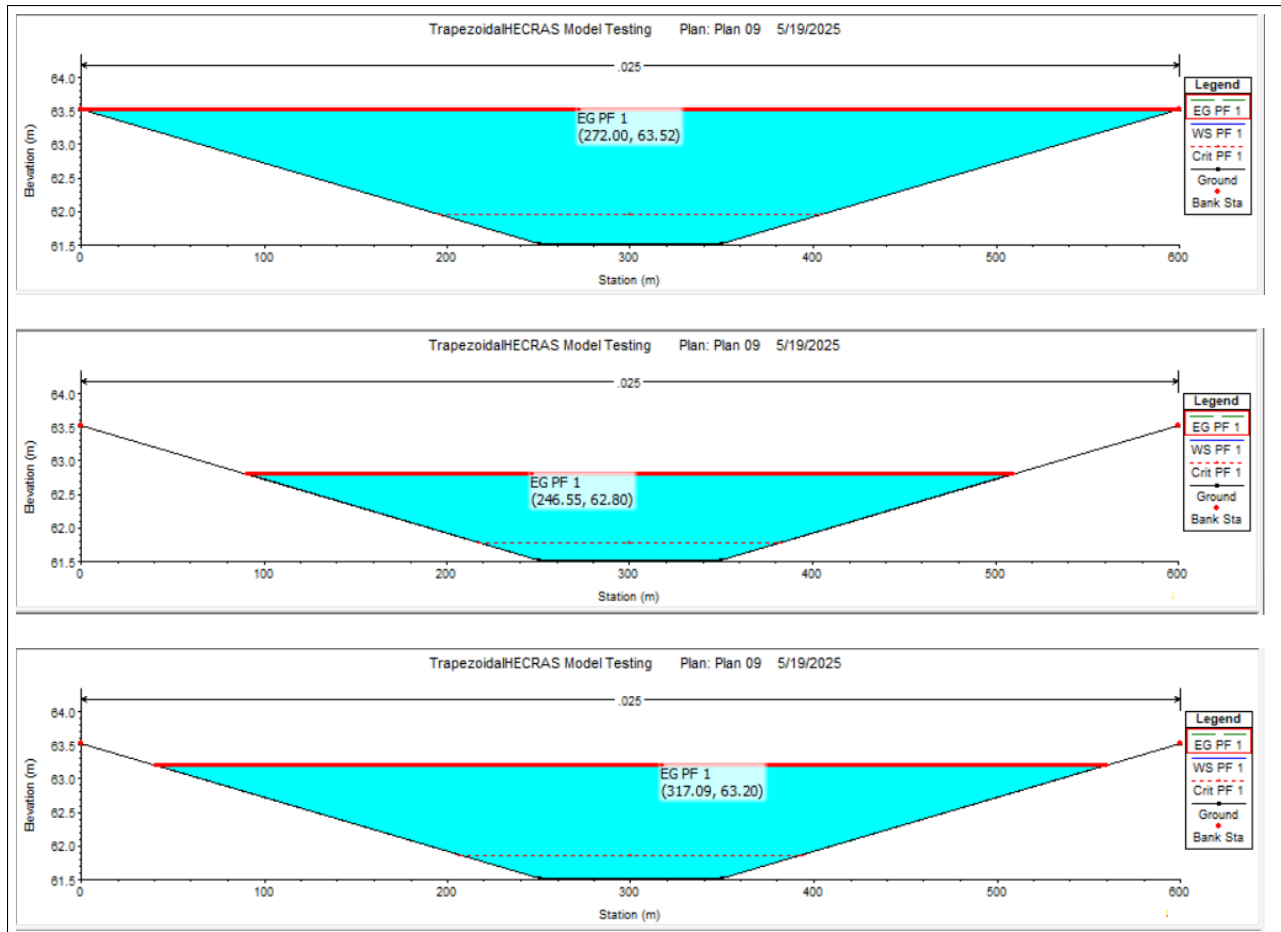


Figure 4.23. Expected downstream cross-sectional profiles at steady state during Stage 1 (top), Stage 2 (middle), and Stage 3 (bottom).

In this stage, the hourly inflow discharge increases from $20 \text{ m}^3/\text{s}$ to $118.75 \text{ m}^3/\text{s}$ and is then held constant to reach a steady state. Meanwhile, the downstream elevation boundary is gradually lowered from 0.68 m above the horizontal reference surface to 0.48 m below it. The expected downstream cross-sectional profile at steady state, based on this elevation boundary, is shown in Figure 4.23. It illustrates water confined to the lower portion of the trapezoidal section, with drying along the channel banks.

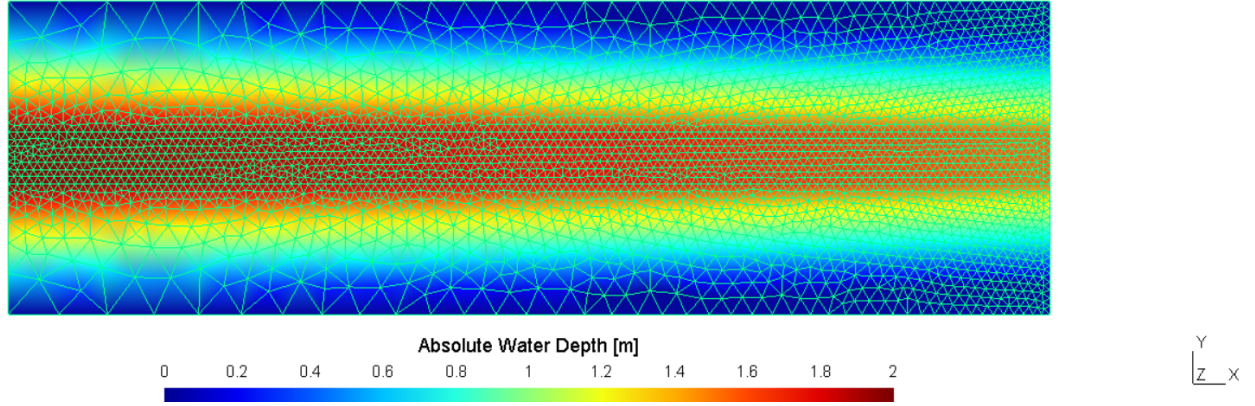


Figure 4.24. 2D water depth distribution at hour 14 during stage 1.

At the end of Stage 1 (hour 14), the 2D water depth distribution simulated by the Julia implementation is shown in Figure 4.24. The results confirm that the model reaches a steady state where the deepest section of the river, located near the outflow boundary on the left side of the domain, exhibits a depth of approximately 2 m. This aligns with the imposed downstream elevation boundary condition. Water remains confined to the central low-elevation region of the trapezoidal channel, while drying occurs progressively toward both banks.

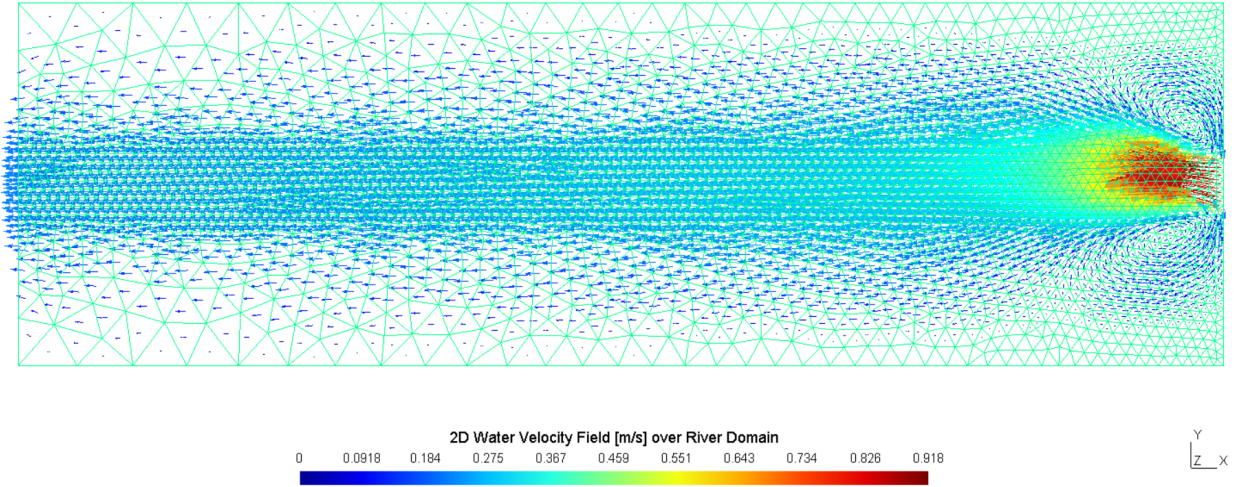


Figure 4.25. 2D velocity vector field at hour 14 during stage 1.

Figure 4.25 presents the 2D velocity vector field at the steady-state flow pattern in Stage 1. Flow enters from the right boundary (inflow) and proceeds toward the left (outflow), with velocity magnitudes indicated by color and arrow size. The inflow boundary is applied only along the central portion of the river channel (from $y = 250$ m to $y = 350$ m), which introduces a non-uniform inlet profile. As a result, recirculation zones can be observed near the inflow region, particularly at the upper and lower edges. Also, the computational mesh is not perfectly symmetric with respect to the river centerline, leading to slight variations in element size and shape across the domain. This asymmetry can contribute to a non-symmetric velocity distribution at the inflow, as the finite element solution is sensitive to mesh geometry. Therefore, the calculated nodal velocities may have small asymmetries.

Downstream of the inflow boundary, the flow field gradually stabilizes. After approximately 500 m from the inflow end, the velocity vectors align predominantly in the x

direction from inflow to outflow, indicating that the initial non-uniformity and recirculation disappear as the flow develops.

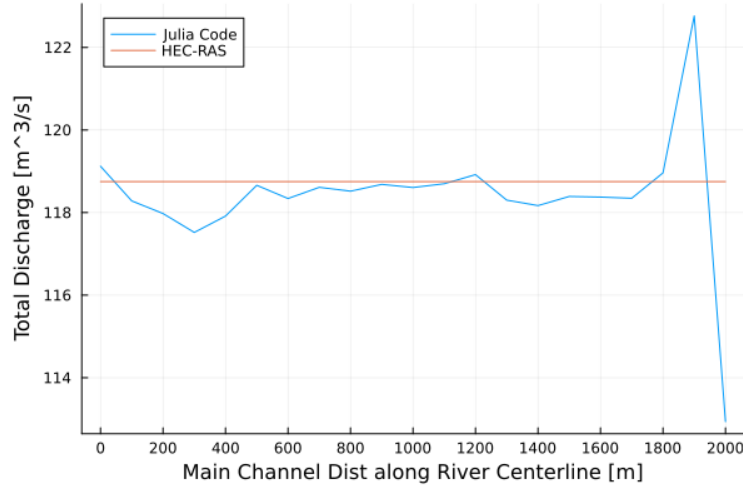


Figure 4.26. Total discharge along the river centerline at hour 14 during stage 1.

Figure 4.26 shows the computed total discharge across 21 cross sections along the river centerline at hour 14 of Stage 1. The total discharge in each cross section is evaluated by integrating the local discharge flux q_n using an appropriate first order quadrature rule. The overall discharge remains close to the prescribed inflow boundary condition of 118.75 m³/s, indicating that the mass conservation is well maintained across the domain.

Near the inflow boundary, damping appears at the far right end of the domain, with a maximum relative reduction in velocity of approximately 4.2% compared to the reference value. This may be attributed to the circular flow structures that develop around the partial-width inflow zone, where flow enters only between $y = 250$ m and $y = 350$ m. In such cases, a portion of the flow may be oriented obliquely or vertically with respect to the cross-sectional plane, therefore reducing the effective component in the streamwise x -direction. As a result,

the total discharge projected along the cross sections may differ from the prescribed net inflow flux, leading to localized over- or under-estimation near the boundary. Elsewhere in the domain, the computed discharge values remain consistent and well-aligned with the expected steady-state behavior.

4.2.2.2. Stage 2 – Drying Phase:

The inflow discharge is reduced to 50 m^3 and held constant, while the downstream elevation remains at a low level. The corresponding steady-state profile based on this elevation boundary is shown in Figure 4.23, where a narrower wetted area appears, and more of the channel cross section is exposed.

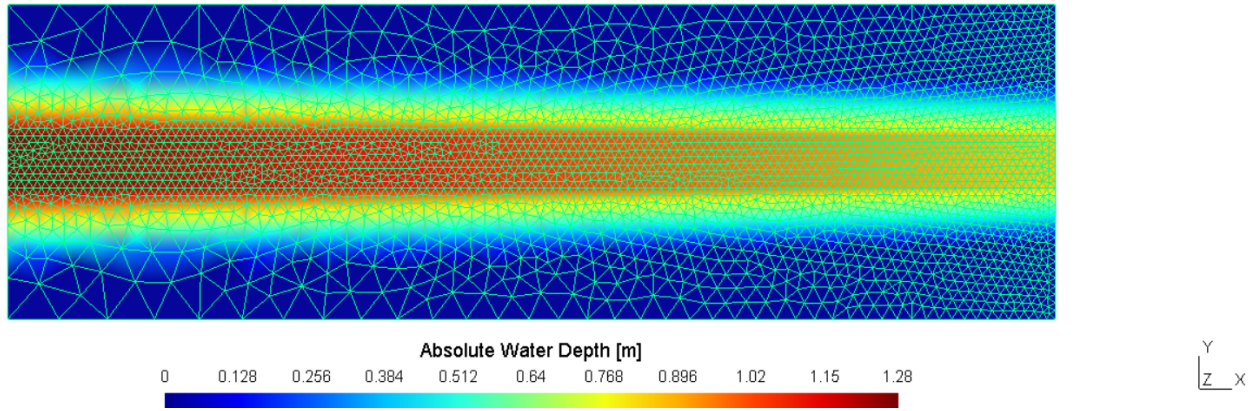


Figure 4.27. 2D water depth distribution at hour 23 during stage 2.

At the end of Stage 2 (hour 23), the simulated 2D water depth distribution is shown in Figure 4.27. The water surface elevation at the outflow boundary reaches approximately 62.80 m, consistent with the sum of the river bed elevation (61.52 m) and the water depth h (1.28 m). Compared to stage 1, the overall wetted region is significantly reduced, reflecting the drop in inflow discharge and lower downstream water surface elevation. The wet zone

narrows considerably and becomes more confined to the central channel. This contraction of the wetted area is followed by an increase in dry nodes along both the left and right banks, particularly in the upstream half of the domain.

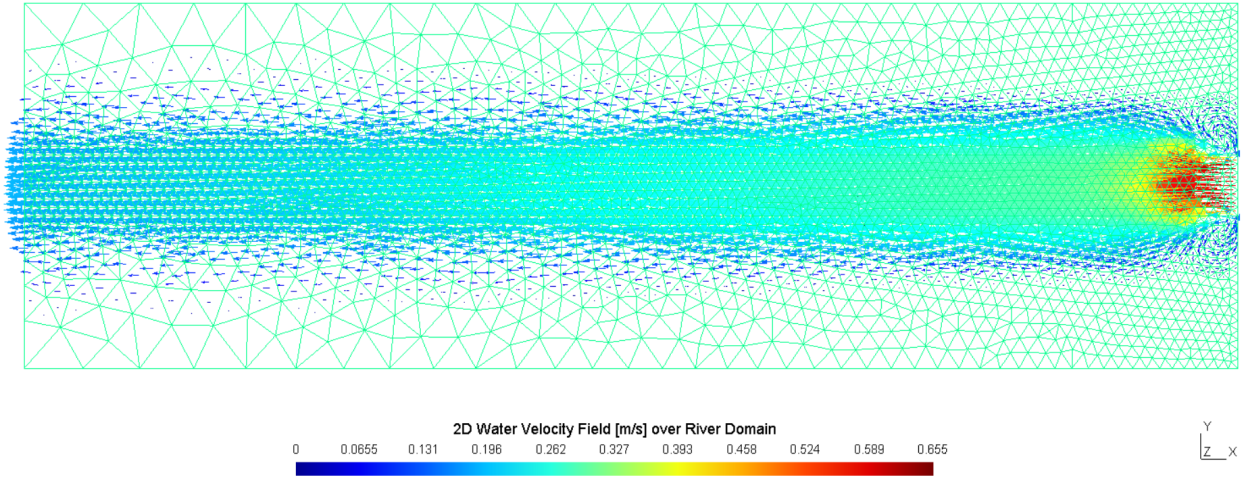


Figure 4.28. 2D velocity vector field at hour 23 during stage 2.

Figure 4.28 presents the 2D velocity vector field $[u,v]$ at hour 23 during Stage 2. With a reduced inflow discharge and lower water surface elevation, the overall velocity magnitude across the domain is also decreased compared to Stage 1. The flow remains directed from right to left, aligning with the inflow and outflow boundary settings. The velocity field is mostly confined to the narrow central channel, consistent with the reduced wetted area observed in the depth distribution. Flow vectors in the lateral regions, especially near the banks, are minimal or absent, indicating dry areas. Some localized irregularities in vector orientation near the inflow boundary exist due to partial-width forcing and mesh asymmetry, but the central streamline remains dominant and stable. The flow appears fully developed and nearly parallel to the x -axis throughout most of the domain.

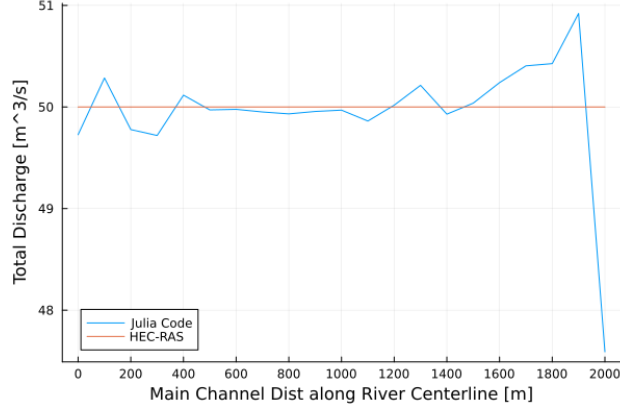


Figure 4.29. Total discharge along the river centerline at hour 23 during stage 2.

Figure 4.29 shows the computed total discharge across 21 cross sections along the river centerline at hour 23 during Stage 2. The overall discharge remains close to the prescribed inflow rate of $50 \text{ m}^3/\text{s}$, confirming mass conservation throughout most of the domain. A notable drop ($\sim 4.6 \%$) is observed near the inflow boundary (right end) of the domain. This reduction is attributed to the same recirculation flow effects discussed in Stage 1. Beyond this region, the computed discharge is stable and aligns with the expected steady-state behavior.

4.2.2.3. Stage 3 – Re-wetting Phase:

Inflow discharge is increased to 80 m^3 and held constant, while the downstream elevation is raised to approximately 63.20 m . The expected steady-state cross section is illustrated in Figure 4.23, where rewetting occurs along the banks and the wetted area expands again.

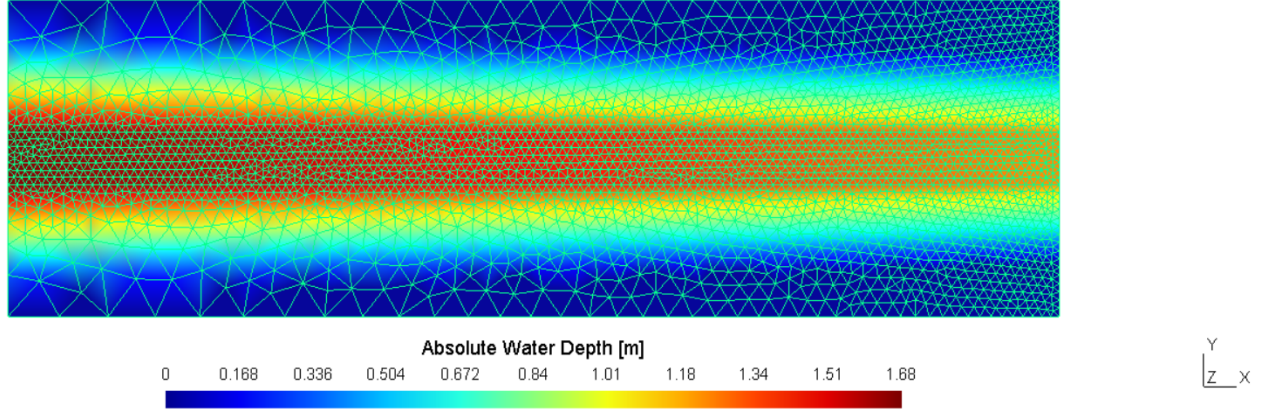


Figure 4.30. 2D water depth distribution at hour 29 during stage 3.

At the end of Stage 3 (hour 29), the 2D water depth distribution is shown in Figure 4.30. Compared to Stage 2, the wetted area has expanded significantly as the inflow discharge increases to $80 \text{ m}^3/\text{s}$ and the downstream water depth rises from 1.28 m to 1.68 m. The water surface gradually reoccupies the side regions of the channel. This rewetting behavior is especially apparent along the lower banks where many previously dry nodes become wet again.

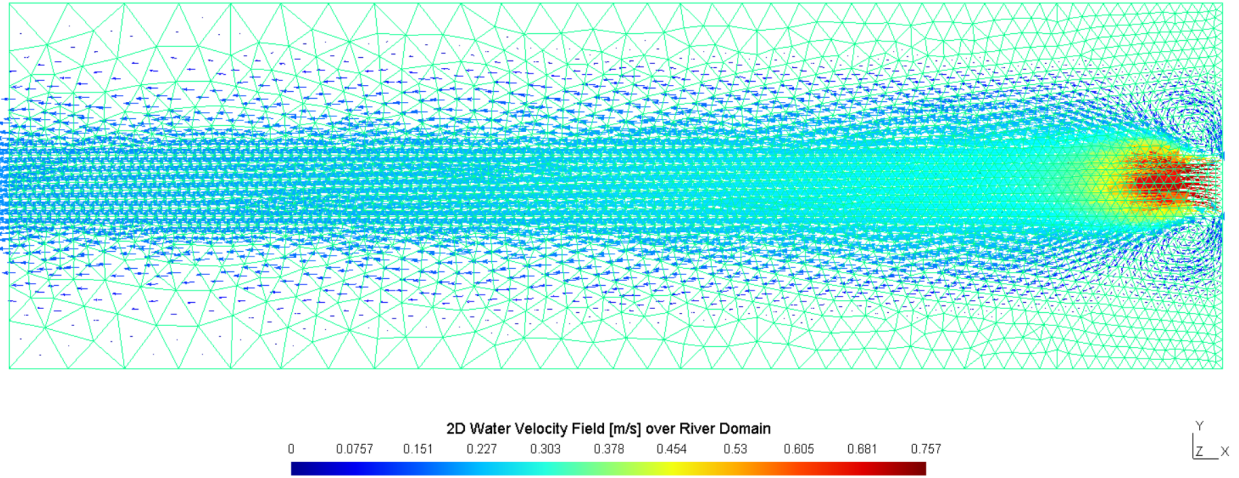


Figure 4.31. 2D velocity vector field at hour 29 during stage 3.

The corresponding 2D velocity vector field is shown in Figure 4.31. With the increased discharge and restored depth, the magnitude of flow velocities increases across the domain. The velocity field is once again concentrated along the central channel, though expanded slightly in width due to rewetting. Local recirculation near the partial-width inflow remains, but the flow realigns horizontally within around 200 m from the inlet.

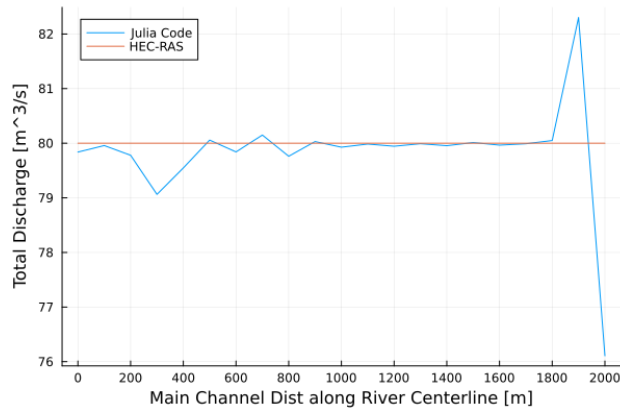


Figure 4.32. Total discharge along the river centerline at hour 29 during stage 3.

Figure 4.32 shows the total discharge along the river centerline at hour 29. The values match closely the prescribed inflow of $80 \text{ m}^3/\text{s}$, with minor oscillations in the interior of the domain. As with earlier stages, a localized spike and a drop ($\sim 5 \%$) are observed at the domain’s inflow end due to the partial-width forcing and circular flow patterns. Away from the boundary, the discharge values remain consistent and validate stable mass conservation at steady state.

4.3. Test Case Summary

The validation results demonstrate that the Julia-based solver can reliably handle both the foundational river model with a rectangular cross-section (under cold and hot start conditions), as well as dynamic wetting and drying transitions in a trapezoidal channel. The simulation outputs across all three stages show physically reasonable behavior and good agreement with reference results from other software such as HEC-RAS and the original Fortran code. These tests provide confidence in the core functionality and numerical stability of the implementation. In the next Chapter, the focus will shift toward improving computational efficiency by integrating parallel programming techniques into the current code base, enabling faster simulations for larger scenarios.

5. FEM Parallelization

As introduced in Section 3.6.2, the hydrodynamic solver was initially implemented as a serial algorithm. To improve computational efficiency and achieve significant speedup, this Chapter extends the solver to parallel execution.

Although Julia supports various parallel programming techniques, including multi-threading, distributed, and GPU computing [13], this thesis prioritize multi-threading and distributed computing.

The Chapter begins with a discussion of a multi-threaded shared memory implementation of the shallow-water hydrodynamic equations, presenting its implementation in Julia and analyzing its performance (Sections 5.2 and 5.3). We then move on to introduces distributed-memory parallelization, covering domain decomposition, parallel solver execution, and performance evaluation (sections 5.4 and 5.5).

5.1. Parallelization Prerequisites

To enable speedup, the original serial algorithm must be parallelizable. That is, its workload must be divisible into tasks that can be executed simultaneously. In the context of the hydrodynamic model used in this work, this condition is satisfied. As introduced in Chapter 3, the solution at each time step is computed based on local values. For example, the nodal discharge p at each time step depends only on the values h , p and q from its connected nodes in the previous time step. This local dependency enables parallelization through decomposition at each time step, allowing each process or thread to store and

process only a subset of the total elements independently, either within its local memory or shared memory space.

5.2. Share Memory Implementation

Parallel computing relies on processes and threads as fundamental execution units. A process is an independent instance of a running program (sequence of instructions), equipped with its own memory space. Processes do not naturally share memory with each other, and in distributed computing, multiple processes run in parallel, each maintaining its own isolated memory, requiring explicit communication to exchange data [22].

In contrast, shared memory parallelism operates within a single process, where multiple threads execute in parallel while sharing the same memory space. This eliminates the need for inter-process communication, making it an efficient choice for parallelism on a single machine [22] with multiple cores, where speedup can be achieved by running different threads on different cores. However, since all threads within a process can read and write to the same memory, synchronization mechanisms (such as `locks` or `atomic` operations) are often required to ensure correct execution when multiple threads operate on shared resources, preventing race conditions when multiple threads attempt to modify the same memory location simultaneously [23].

While distributed memory parallelism, where multiple processes interact, is discussed in later sections, the following discussion focuses on shared memory parallelism in Julia. Although various approaches exist for implementing shared memory parallelism in Julia, this work uses Julia’s built-in multi-threading support provided by the `Threads` module [13], including `Threads.@spawn` for basic threading and external libraries such as `ThreadsX.jl` for higher-level parallelism.

5.3. Shared Memory Parallel Strategy

To achieve parallelism among threads, the first step is to determine how to partition the computational workload efficiently. Proper task distribution is important to balancing the workload among threads, ensuring optimal performance and speedup in parallel execution. In finite element method (FEM) parallelization, two common approaches are used for task partitioning and were implemented as initial parallelization attempts:

1. Element Index-Based Decomposition – Elements are assigned to threads based on their numerical order in the mesh file, such as the sequence returned by Gmsh. This method does not consider spatial connectivity, meaning that adjacent elements in the mesh may be processed by different threads. For example, the assignments might be: elements 1-1000 to thread 1, 1001-2000 to thread 2, and so on.

To parallelize the hydrodynamic equations based on this index-based partitioning of the computational domain decomposition, the `@spawn` macro in Julia is used to spawn multiple threads, with the strategy of having each thread compute a portion of the global flux vector in parallel. Since the global flux vector requires looping over all N elements, the computational complexity is approximately $O(N)$ in serial execution. By distributing the workload among P threads, the ideal theoretical complexity is reduced to $O(N/P)$, assuming perfect load balancing and no synchronization overhead.

A key challenge in this approach is to avoid race conditions when updating the global flux vector ($\underline{\mathcal{F}}_g$). These conflicts arise because nodes are shared between multiple elements, and elements assigned to different threads may contain overlapping nodes. For example, nodes in Elements 1-1000 may also be part of Elements 1001–2000, causing multiple threads to attempt writing to the same locations in $\underline{\mathcal{F}}_g$.

A simple, though suboptimal, approach (in terms of memory overhead) is adopted to eliminate these memory conflicts: each thread i maintains a local flux vector ($\underline{\mathcal{F}}_i$) with the same vector size as $\underline{\mathcal{F}}_g$ and updates only the node indices corresponding to its assigned element range (e.g., Thread 1 updates values for the nodes associated with Elements 1–1000). Once all threads complete their computations, the global flux vector is obtained by summing the local contributions across n threads (using `ThreadsX.mapreduce`¹):

$$\underline{\mathcal{F}}_g = \sum_{i=1}^n \underline{\mathcal{F}}_i.$$

Since no two threads write to the same shared memory location, this strategy avoids race conditions by construction, without requiring synchronization mechanisms such as `@atomic` or `@lock`, which may introduce performance bottlenecks. While this method improves time efficiency by avoiding synchronization overhead, it sacrifices memory efficiency, as each thread must allocate a full-sized local flux vector.

2. Geometry partition - The computational domain is divided into geometrically continuous subdomains, where each thread is responsible for elements within a disjoint localized region.

The implementation of this strategy is identical to the distributed memory parallelization approach described in Section 5.5.1, with the key difference that no inter-thread communication or data exchange is required. Since all threads share the same memory space, each thread can directly access the data it needs without explicit synchronization.

¹Details can be found in [22]

As an initial, simple, attempt at parallelism, both of these shared memory strategies were implemented and tested.

5.4. Shared Memory Performance Analysis

Both partitioning strategies are practical in shared memory parallelization, as all information is accessible to all threads. The choice between them primarily affects computational efficiency rather than feasibility. Unfortunately, neither approach achieved a reasonable speedup after parallelization, with both cases showing a maximum speedup of less than 2, regardless of increases in problem size or the number of threads.

The limited speedup can be attributed to several factors. While there were no explicit thread synchronization costs or obvious workload imbalance, a key limiting factor might be memory access efficiency. Each thread may frequently access global variables stored in deeper memory layers, rather than benefiting from faster cache access. This might add latency likely contributed to the overall performance bottleneck.

While it may be possible to improve the shared memory implementation performance in the future, we assumed that a distributed memory environment would bypass these issues directly. Thus, given the shared memory limitations, distributed memory parallelization was pursued as a more effective alternative, as discussed in the following section.

5.5. Distributed Parallel Implementation

Distributed computing is achieved by running multiple processes, each with its own independent memory space [13]. These processes can be launched on any available processors, whether on a single machine with multiple cores or across multiple computers in a cluster. A processor may run one or multiple processes simultaneously, depending on resource allocation and system configuration. In typical distributed computing environments, the number of

processes is often set equal to the number of physical processors/cores used at run time to maximize hardware utilization. Each process is assigned a unique identifier, which is independent of the physical processor it executes on [13].

In our implementation, we use a Single Program Multiple Data (SPMD) approach, where each process runs the same executable program independently. Since each process operates in its own memory space, a message-passing mechanism is essential for enabling data exchange across processes. A widely used message-passing protocol is the Message Passing Interface (MPI), originally developed in C and accessible in Julia through the MPI.jl wrapper [13]. MPI provides better scalability for large problem sizes and is widely adopted in high-performance computing (HPC) [13]. Julia, on the other hand, has its native message-passing package, implemented through Distributed.jl, which offers greater flexibility with its high-level parallel functions. The implementation approach used in this work is based on Distributed.jl due to its ease of use and compatibility with Julia’s parallel constructs.

5.6. Distributed Parallelize Strategy

To parallelize the solution, each process must handle a portion of the computation independently, which requires a division strategy to distribute the computational workload across multiple processing units efficiently. As mentioned in the shared memory section, there are two common approaches for domain partitioning: element index-based decomposition and geometry-based domain decomposition. While both methods are feasible in a shared memory system, their applicability in distributed memory parallelization differs, as discussed below.

In element index-based decomposition, elements are assigned to processes based on their original ordering in the mesh file without considering their spatial connectivity. While

this method is simple and efficient for shared-memory parallelization, it is less suitable for distributed-memory environments, as continuous FEM computations rely on information from neighboring nodes to assemble global quantities, as discussed in Section 3.2. Since elements assigned to a particular process may not be physically adjacent, each process may need to access node information from a large portion of the global mesh stored on other process to correctly update its assigned elements. This leads to computational inefficiencies through added communication requirements and is generally unsuitable for large-scale distributed parallelization.

To overcome these limitations of element index-based decomposition, a more practical approach is to partition the mesh based on geometry. By dividing the computational domain into geometrically continuous/partialized subdomains, we ensure that neighboring elements remain together within each partition. This reduces inter-process dependencies and minimizes unnecessary message passing. However, communication is still required at the boundaries of each partition to exchange node information, which will be discussed in detail later.

To formalize the decomposition process, we define the 2D computational domain as Ω , which is partitioned into P disjointed subsets, denoted as Ω_P , where each subset is assigned to one process. Ideally, if the mesh contains a total of N elements, each partition will contain approximately N/P elements, as the amount of work per element is largely the same which ensures a balanced workload distribution across processes [24]. The original serial algorithm has a complexity of $O(N)$ for the processing step, which involves assembling the global flux vector and updating the unknowns. This step forms the core of the computation and is parallelized using the distributed parallelization strategy. By partitioning the domain into P subdomains, each process independently assembles the flux terms over N/P elements,

leading to a theoretical parallel complexity of $O(N/P)$, assuming optimal load balancing and negligible communication overhead.

5.6.1. Domain and Work Load Decomposition

To ensure an approximately even distribution of computational workload across processes, the mesh must be partitioned appropriately. In this work, we leverage built-in functionality provided by the Gmsh API, to divide the computational domain into subdomains while retaining element connectivity. This approach allows us to achieve a balanced element distribution, even for complex geometries. Though in what follows we illustrate results on a rectangular domain.

For demonstration purposes, we consider the same mesh as used in Test Case 1 (Figure 3.2), which consists of 2111 nodes and 4000 elements. Using Gmsh, the domain is partitioned into three parts, with approximately equal element counts (part 1: 1333 elements, part 2: 1333 elements, part 3: 1334 elements) as showed in Figure 5.1, with the elements belonging to each subdomain colored and labeled accordingly.

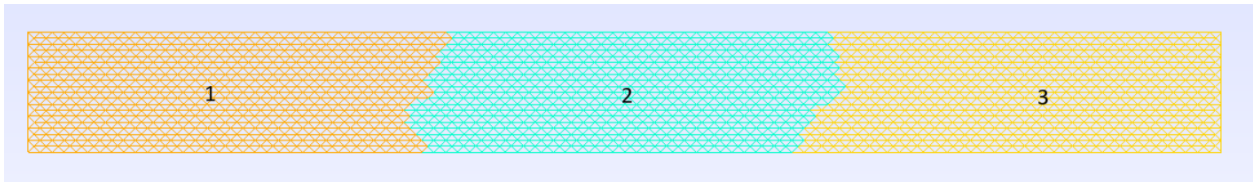


Figure 5.1. Partitioning of a 2D mesh using Gmsh.

To implement a parallel FEM solver based on a partitioning of the mesh, each process is assigned one of the partitions with the goal of computing its local solution (the solution in the elements/nodes belonging to its assigned partition). To ensure correct reconstruction

of the global solution, each partition must update a unique set of nodes, referred to as **local nodes**. These nodes include interior nodes and boundary nodes.

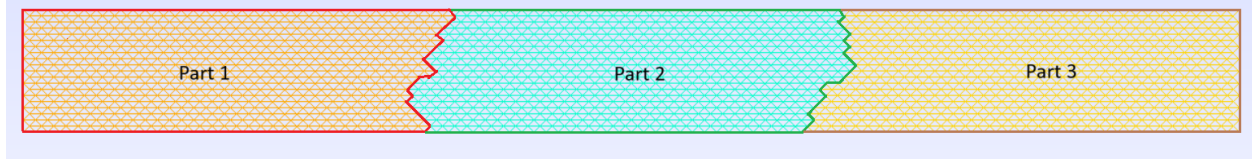


Figure 5.2. Partitioning of the computational domain into three subdomains, each containing a unique set of local nodes (interior + selected boundary nodes). The solid lines indicate partition boundaries, and only the nodes enclosed within each partition are updated by the corresponding process. Boundary nodes are assigned to exactly one partition, ensuring no overlap in updates across partitions.

To illustrate, Figure 5.2, shows how each partition is assigned its own distinct set of local nodes. Partition 1 (red boundary) includes the rightmost boundary nodes, whereas Partition 2 (green boundary) excludes these nodes but includes its own right boundary nodes. Similarly, Partition 3 (brown boundary) includes its own right boundary nodes, ensuring that there is no overlap between partitions. The unknown variables to be updated at these nodes in each partition are denoted $(U_{P_i}^{n+1})$, where i indicates the i -th partition. When the local solutions $U_{P_i}^{n+1}$ from all partitions are merged back into the correct node order in the global solution, the final result U^{n+1} will be identical to the solution produced by the serial algorithm, with differences limited to floating-point rounding errors due to the varying order of operations in parallel execution.

5.6.1.1. Data Exchange in Parallel FEM

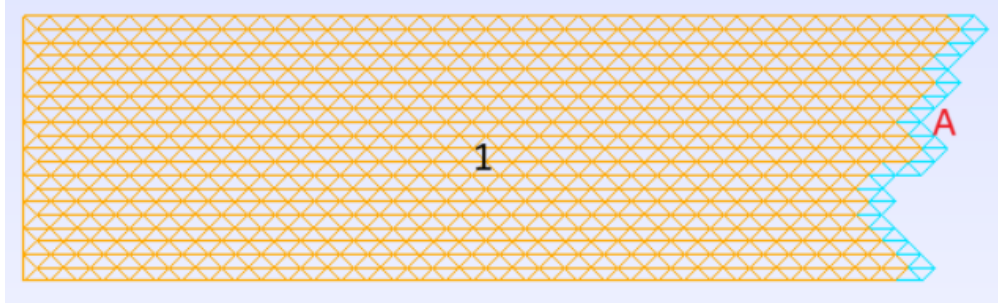
Updating $U_{P_i}^{n+1}$ in parallel would be straightforward without any message passing if these values depended solely on $U_{P_i}^n$. However, in the serial algorithm, the update equation

for U^{n+1} is given by

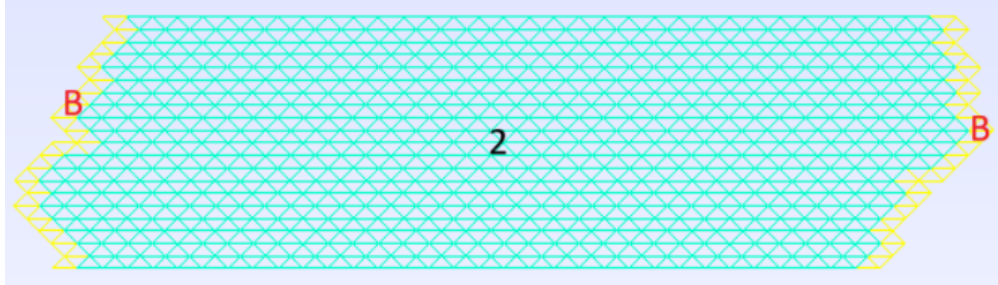
$$U^{n+1} = \mathbf{M}_g^{-1} \mathcal{F}_g(U^n) \Delta t + U^n,$$

where $\mathcal{F}_g(U^n)$ is the global flux vector. As described in Chapter 3, the computation of $\mathcal{F}_g(U^n)$ at any given nodes requires access to the value of U^n from all connected elements. This implies, in the distributed parallel case, that these connected elements may lie across partition boundaries. Since each partition only owns a subset of known node values ($U_{P_i}^n$), without additional information it lacks direct access to the values at nodes belonging to neighboring partitions. As a result, $U_{P_i}^n$ alone is insufficient to update $U_{P_i}^{n+1}$ because the flux contributions from elements that span across partition boundaries are missing. To address this, we introduce halo nodes (also known as ghost nodes) [23], which store additional node values from the adjacent partitions that are required for correctly assembling $\mathcal{F}_g(U^n)$ at partition boundaries. These ghost nodes, along with the local nodes, form the extended set $U_{P_i,ghost}^n$, which includes both local unknowns U_{P_i} and ghost unknowns $U_{ghost(P_i)}$ of the i -th partition. This structure enables each partition to update its solution independently while ensuring numerical accuracy across partitions.

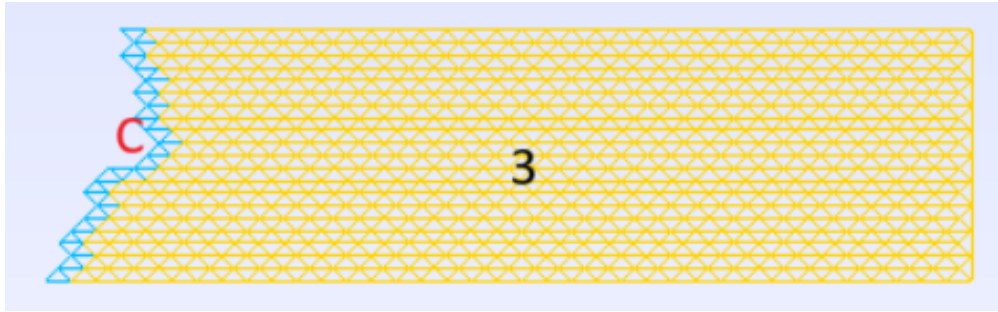
Obtaining halo entries is supported directly in Gmsh's partitioning function that provides an option to return all ghost entities, along with the ability to visualize ghost elements directly in the graphical interface. From the entity tags returned by Gmsh for each partition, we can extract the corresponding local nodes, local elements, ghost nodes, and ghost elements. Figure 5.3, illustrates how Gmsh generates both local and ghost meshes for each partition, where the halo regions are labeled as A, B, and C.



(a) Partition 1 with ghost mesh A.



(b) Partition 2 with ghost mesh B.



(c) Partition 3 with ghost mesh C.

Figure 5.3. Ghost mesh representation for each partition.

These ghost nodes, introduced from adjacent partitions, expand the local node set, ensuring that all required data is available for correctly assembling the local global flux vector $(\mathcal{F}_g(U^n))_{P_i}$. To maintain numerical accuracy and to prevent redundant updates, ghost nodes must not overlap with local nodes. This means that any node classified as a

local node for one partition should never be included as a ghost node for that partition. During implementation, careful distinction between local and ghost nodes is necessary to avoid redundancy and ensure correct data exchange between partitions.

Since the local and ghost regions have already been defined for each partition, we can now modify the serial update equation to reflect parallel execution across multiple partitions. In the parallel formulation, each partition updates its own set of local nodes while using values from ghost nodes to ensure numerical consistency at partition boundaries. The update equation for each partition can be written as:

$$U_{P_i}^{n+1} = \Delta t(\mathbf{M}_{P_i}^{-1} \mathcal{F}_i(U_{P_i,ghost}^n)) + U_{P_i}^n,$$

After computing the updated local solution $U_{P_i}^{n+1}$, we note that $U_{ghost(P_i)}^{n+1}$ is not updated correctly due to a lack of sufficient input data, as it depends on values from neighboring partitions. In time integration FEM solvers, correcting $U_{ghost(P_i)}^{n+1}$ is important, as these values influence the computation of $U_{P_i}^{n+1}$ at the next time step. Since ghost nodes in one partition correspond to a subset of local nodes in adjacent partitions, the correct values of $U_{ghost(P_i)}^{n+1}$ are obtained by exchanging them with the local solutions of neighboring partitions that share the same node indices.

Figure 5.4 illustrates this data exchange process using the same computational mesh but simplified into rectangular partitions for clarity.

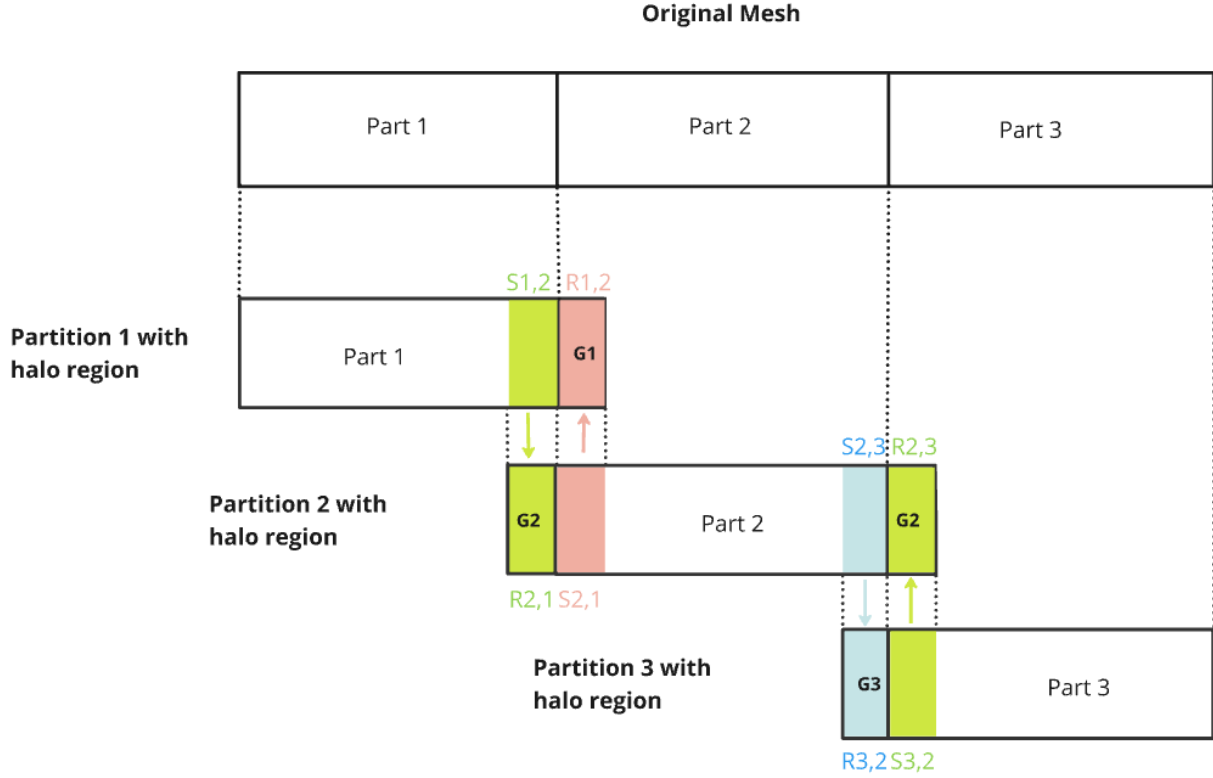


Figure 5.4. Data exchange process between partitions with halo regions. The computational domain is divided into three partitions (Part 1, Part 2, and Part 3), each assigned to a separate process. To ensure accurate updates at partition boundaries, each partition maintains a halo region (G1, G2, G3) where ghost node values are exchanged.

To highlight this exchange, different colors are used:

- Pink: Represents $U_{ghost(P_1)}^{n+1}$ in Partition 1 and their corresponding sub-array in $U_{P_2}^{n+1}$ in Partition 2, from which data will be exchanged.
- Green: Represents $U_{ghost(P_2)}^{n+1}$ in Partition 2, which must receive updates portions $U_{P_1}^{n+1}$ and $U_{P_3}^{n+1}$ from Partition 1 and Partition 3.

- Blue: Represents $U_{ghost(P_3)}^{n+1}$ in Partition 3, which receives updates from the corresponding $U_{P_2}^{n+1}$ in Partition 2.

Each partition follows a send-receive mechanism, denoted as $S_{\xi,\zeta}$ and $R_{\xi,\zeta}$ where

- $S_{\xi,\zeta}$ represents sending $U_{P_\xi}^{n+1}$ from partition ξ to ζ
- $R_{\xi,\zeta}$ represents receiving $U_{P_\zeta}^{n+1}$ from partition ζ to ξ

For this specific partitioning setup, each partition updates its $U_{ghost(P_i)}^{n+1}$ with adjacent partitions, following the data flow direction indicated by the arrows in Figure 5.4. This structured communication ensures that each partition correctly updates its halo nodes using the latest information from neighboring partitions. This is important because, in the next time step, the halo nodes will become part of the local solution, contributing to the accurate computation of $U_{P_i}^{n+1}$.

5.6.2. Parallel Algorithm

With the key components of parallelization established, we now describe the distributed parallel implementation, summarizing the essential steps and how they are structured in Julia. The following pseudocode outlines the overall workflow, providing a structured overview before delving into the detailed Julia implementation, where each step will be explained in detail.

Algorithm 3 Parallel FEM implementation.

```
1: Step 1: Initialize Distributed Environment in Julia
2: if process id == 1 then
3:   Preprocessing:
4:   Same as serial Algorithm
5:   Step 2: Create partitions with Gmsh
6:   Prepare arg_Lists[] to store arguments for each partition
7:   Processing:
8:   Step 3: Call pmap(hydro, arg_List) for all partitions in parallel
9:   Postprocessing:
10:  Merge local solutions  $U_{P_i}^{n+1}$  from all partitions to reconstruct the global solution  $U^{n+1}$ .
11: end if
```

Algorithm 4 Hydrodynamic computation for each partition

```
1: function HYDRO(arg_Pi) ▷ argument for i-th partition
2:   for each time step do
3:     Assemble  $\underline{\mathcal{F}}_{P_i}^{p,q}(U_{P,ghost})$ 
4:     Update water discharges  $p_{P_i}^{n+1}, q_{P_i}^{n+1} = \mathcal{M}_{P_i}^{-1} \underline{\mathcal{F}}_{P_i}^{p,q}(U_{P,ghost}) \Delta t + p_{P_i}^n, q_{P_i}^n$ 
5:     Step 4: Update  $p_{ghost(P_i)}^{n+1}, q_{ghost(P_i)}^{n+1}$  via message passing between processes
6:     Update  $\eta_{P_i}^{n+1.5}$  and  $\eta_{ghost(P_i)}^{n+1.5}$  with the same process
7:     Step 5: Synchronize all processes
8:     if itime % checkpoint_interval == 0 then ▷ Optional merge at checkpoints
9:       Merge local solutions  $U_{P_i}^{n+1}$  to reconstruct the global solution  $U^{n+1}$ .
10:    end if
11:  end for
12: end function
```

5.6.3. Julia Implementation

In Julia, distributed parallel execution usually begins by launching multiple processes, followed by distributing workloads or tasks among them while managing the necessary inter-process communication. The following section systematically breaks down the implementation of this process, following the structured steps outlined in Section 5.5.2.

5.6.3.1. Initialize Distributed Environment

To efficiently parallelize the FEM solver, we use Julia’s distributed computing package, Distributed.jl. This package enables parallel execution by managing clusters of processes, where a main process (process ID: 1) is responsible for adding, removing, and querying worker processes that execute computational tasks in parallel. **Worker processes** are

the computational processes designated for parallel execution, responsible for performing distributed tasks and contributing to the overall speedup of the computation. Since this thesis focuses on parallelization within a single machine (local cluster) rather than a multi-node setup, we first describe the key mechanisms for launching and managing worker processes before detailing the parallel implementation.

Each Julia process runs in an independent memory space and is assigned a unique process ID (pid). By default, only a single process (pid = 1) is available, which can be verified using either:

- `nprocs()` (Returns the total number of processes (including the master process).)
- `nworkers()` (which returns the number of worker processes.)

Since no additional processes exist at the start, `nworkers()` initially returns 1, indicating that the main process also functions as the sole worker process (`nworkers() = nprocs() = 1`). When additional worker processes are added, they operate separately from the main process, and the number of worker processes is given by `nworkers() = nprocs() - 1`. In Julia, there are two common methods for launching and managing additional worker processes in a local cluster: specifying them directly from the Julia REPL (command line) at startup, or dynamically adding and removing them within the code during execution.

Once additional worker processes have been launched, they do not automatically have access to the functions, variables, or packages available in the main process. Since each process operates in an independent memory space, any code or data needed for execution must be explicitly loaded on all processes in Julia, this is achieved using the `@everywhere` macro, which ensures that a given expression is executed on all active processes, making the defined functions, variables, or imported packages available to all workers. For example:

`@everywhere using Package` ensures that all processes load the `Package` module, allowing it to be used in parallel computations. Similarly, scripts containing function definitions or preloaded variables can be distributed using: `@everywhere include("file.jl")` which loads `file.jl` on every process, making its contents globally accessible within the cluster. By ensuring that all required functions and data are available across processes, we enable a consistent execution environment for distributed parallel computations.

5.6.3.2. Create Partitions with Gmsh

After establishing the distributed environment, the next step is to ensure an even distribution of workload across all processes. This is achieved by partitioning the computational domain, allowing each process to handle a subset of the mesh along with additional ghost regions (as shown in Figure 5.3). Julia, in combination with Gmsh, provides built-in partitioning methods to achieve this efficiently. Gmsh offers two primary partitioning techniques: SimplePartition plugin and METIS [25]. For additional details on the METIS and SimplePartition algorithms, refer to Appendix E. This thesis employs METIS with a recursive bisection partitioning algorithm [26] to achieve an evenly distributed workload, particularly for irregular river geometries. The recursive approach ensures better load balancing while preserving the connectivity of neighboring elements, reducing computational imbalances [27]. Additionally, METIS provides an option to minimize inter-process communication by reducing the communication volume between partitions.

METIS-based partitioning in Gmsh can be performed through three methods. However, this thesis focuses exclusively on runtime partitioning, as it offers flexibility by dynamically adapting to the number of available processes. This approach aligns with the parallelization strategy adopted here, where each process handles a single partition. For

details on the alternative methods—using the Gmsh graphical user interface (GUI) or geo file-based partitioning—please refer to Appendix F.

To practically apply the METIS partitioning strategy within Julia at runtime, the following command is used:

```
gmsh.model.mesh.partition(M),
```

where M specifies the number of partitions to create, typically matching the number of available processes to ensure balanced workload distribution across all processes. The effectiveness of the partitioning depends on how the mesh is divided and how communication is minimized. To further optimize the partitioning process, Gmsh provides several customization options. These settings can be configured using the `gmsh.option.setNumber()` function prior to invoking the partitioning command. For example, to enable ghost region creation for each partition, the following command is used:

```
gmsh.option.setNumber("Mesh.PartitionCreateGhostCells", 1).
```

This setting ensures that each partition includes a ghost region, comprising elements that share at least one node with neighboring partitions [13].

After partitioning the mesh, the next step is to extract the relevant mesh information (elements and nodes) for each partition. This process involves identifying partition entities, which represent both the partitioned mesh and its corresponding ghost regions. The partition entities can be accessed using the `gmsh.model.getPartitions`² command. There are two types of entities of interest:

²For detailed examples, refer to the `partition.py` file in the Gmsh tutorial [25].

- “Partition surface” entity: Represents the main mesh region assigned to a partition.
- “Ghost Surface” entity: Represents mesh elements from neighboring partitions required for computations at partition boundaries.

From these entities, the corresponding elements and nodes can be retrieved using Gmsh’s built-in functions.

At the end of this step, each partition P_i will have access to its local elements and nodes, along with the corresponding ghost elements and nodes required for inter-partition computations.

5.6.3.3. Parallel Hydrodynamic Computation Using `pmap`

With the computational domain now partitioned, the next step is to distribute the partitions across all the available workers and perform the hydrodynamic updates in parallel. Each worker is responsible for computing the unknown variables U_{P_i} associated with the assigned partition. To efficiently manage this parallel computation, we use Julia’s `pmap` function, which is designed for scenarios where each function call involves heavy computation [13] - making it well-suited for time-step updates in hydrodynamic simulations.

The primary objective of the `pmap` function in Julia is to concurrently execute a specified function across available worker processes, distributing tasks efficiently to optimize performance. The general syntax for the `pmap` function is:

$$\text{pmap}(f, c)$$

where

- `c`: The collection of arguments

The collection `c` of arguments is denoted as `arg_List` in this study. Each element `arg_Pi` in `arg_List` can be viewed as an independent task or workload unit. In the context of the hydrodynamic solver, `arg_Pi` represents a tuple (A tuple is an immutable sequence of values, which may be of different types, arranged in a fixed order and separated by commas [13]) of partition-specific information, where `Pi` refers to the partition index rather than the process ID (`pid`).

- `f`: The function to be executed

`f` can be a lambda/anonymous function or a predefined function (such as `hydro` in this study) to be launched with each task independently. To ensure accessibility across all worker processes, `hydro` must be defined on all processes. This is achieved using Julia’s `@everywhere` macro, which loads the function definition across all available processes.

By default, when calling `pmap`, Julia employs a caching pool [13] that uses all available worker processes. This mechanism is based on dynamic task assignment, meaning:

1. No fixed relationship exists between a specific worker and a task (For example: `arg_P` does not necessarily get assigned to the worker with `pid = Pi`).
2. Tasks are distributed on demand—when a worker completes its current task, it immediately picks up the next available one from the task pool.

While `pmap` allows for flexible task distribution, in this study, we explicitly set:

```
length(arg_List) or ntasks = nworkers()
```

This ensures:

- One partition per worker: Each worker handles exactly one partition, aligning with the parallelization strategy where one partition corresponds to one computational task.
- Synchronous updates: All partitions are processed concurrently, maintaining consistency in inter-process communication during hydrodynamic computations, as each process can efficiently exchange halo information ($U_{ghost(P_i)}$) with its neighboring processes.

As previously discussed, `arg_Pi` represents a tuple containing partition-specific information required by the `hydro(arg_Pi)` function. Each `arg_Pi` provides the necessary data for updating the local solution U_{P_i} of partition P_i . The critical components of `arg_Pi` include:

1. Partition Index (P_i): Identifies the specific partition. This ensures that each process recognizes the partition it is responsible for, independent of the dynamically assigned process ID (`pid`)
2. Local and Ghost Solutions (U_{P_i} and $U_{ghost(P_i)}$):
 - At the first iteration (`iter = 0`), U_{P_i} and $U_{ghost(P_i)}$ are extracted from the global initial condition U^0 using local and ghost node indices associated with each partition.
 - After the first iteration (`iter  $\neq$  0`), each process retains its updated U_{P_i} and $U_{ghost(P_i)}$, eliminating the need for further global casting until the final merging step, unless the results are needed at a midpoint in the loop for validation or inspection.
3. Element Indices (`elementIndices_P_ghost`): Contains the indices of elements belonging to partition P_i and its ghost region, enabling the correct assembly of $\underline{\mathcal{F}}(U_{P_i,ghost})$.

4. A list of global channel dictionaries reference (`channel_dict`): Stores a dictionary of `Remote Channel` objects, where each key corresponds to a partition id (`Pi`). This structure is the foundation for data exchange between processes. The exact reason for distributing `channel_dict` to all workers and how it works in inter-process communication will be discussed in a later section.
5. Exchange Data Dictionary (`exchangeDataDict_P`): The `exchangeDataDict_P` plays an important role in managing inter-process communication, ensuring the correct update of ghost node information after each iteration. For each partition P_i , the dictionary contains:

- Key: The index of a neighboring partition that requires data exchange with partition P
- Value(Item): A tuple consisting of two lists:
 - `send_indices`($S_{\xi,\zeta}$): Contains the node indices in partition ξ whose U values need to be sent to the neighboring partition ζ .
 - `replace_indices` ($R_{\xi,\zeta}$): Specifies the ghost node indices in partition ξ that will be updated with the corresponding $U_{ghost(P_\xi)}$ values received from partition ζ .

For example, referring to Figure 5.4

$$Partition1 : \text{exchangeDataDict_1} = \{2 \Rightarrow (S_{1,2}, R_{1,2})\}$$

$$Partition2 : \text{exchangeDataDict_2} = \{1 \Rightarrow (S_{2,1}, R_{2,1}), 3 \Rightarrow (S_{2,3}, R_{2,3})\}$$

$$Partition3 : \text{exchangeDataDict_3} = \{2 \Rightarrow (S_{3,2}, R_{3,2})\}$$

Note that `exchangeDataDict_P` only stores node indices, not the actual solution values. As a result, it can be fully prepared during the preprocessing stage and remains unchanged throughout the simulation.

Once the workload is distributed and the FEM solver computation is launched using `pmap`, the next focus is on the internal workings of the `hydro` function. A important component within this function is the communication between processes, ensuring that each process obtains the correct $U_{ghost(P_i)}$ values necessary for the subsequent time step.

5.6.3.4. Inter-Processes Communication via RemoteChannel

As previous discussed, while each process updates its local solution U_{P_i} independently, these updates are not entirely isolated. The $U_{ghost(P_i)}$ values depend on the solutions of neighboring partitions, requiring efficient data exchange between corresponding processes.

Julia provides several mechanisms for data exchange in distributed computing. In this work, `RemoteChannel` is chosen due to its flexibility and high-level abstraction for inter-process communication. It allows multiple processes to read from and write to the same channel, supporting both one-to-one and one-to-many communication patterns [13]. Additionally, `RemoteChannel` enables asynchronous operations, allowing processes to send and receive data without blocking, which simplifies coordination and reduces synchronization delays [13]. These features make `RemoteChannel` particularly suitable for complex communication scenarios, such as exchanging halo data among neighboring partitions in parallel hydrodynamic computations.

The `RemoteChannel` in Julia can be conceptually understood as a data pipeline or queue, where data is placed (using `put!`) at one end and retrieved (using `take!`) from the other. The channel follows a first-in, first-out (FIFO) order, ensuring that data is processed

in the sequence it was sent [13]. While a single `RemoteChannel` instance could be used for all data exchanges, this approach would be inefficient in complex communication scenarios. Retrieving the correct data for each partition would require each process to sequentially parse all data in the channel, leading to unnecessary overhead and complexity. To address this, a dictionary of `RemoteChannel` objects (`channel.Dict`) is introduced, providing a dedicated communication pipeline for each partition. In this structure, each key in the dictionary represents a partition index, while the associated value is the `RemoteChannel` object linked to that specific partition. This design simplifies data management by ensuring that each process interacts only with the relevant channels corresponding to its assigned partition. Specifically, the `exchangeDataDict_P` maintained by each process identifies the required channels by matching its keys with the corresponding partition indices in the global `channel_dict`. This mapping guarantees that each process exchanges data solely with the necessary neighboring partitions. Consequently, data exchange becomes more efficient and organized, as each partition can directly access the relevant halo data without additional filtering or processing.

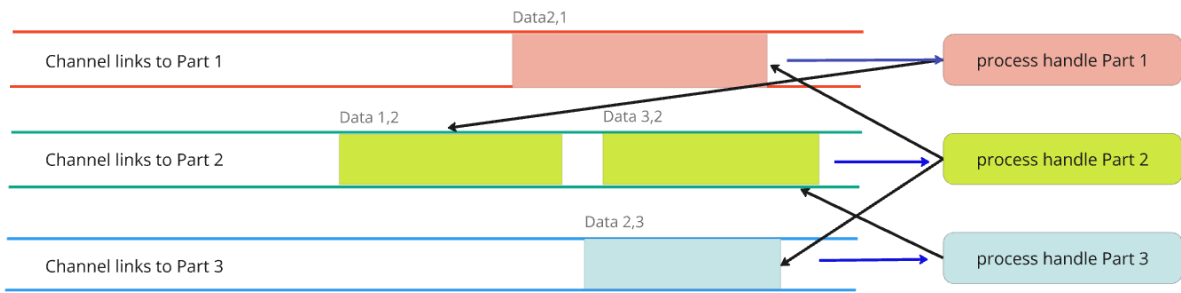


Figure 5.5. Schematic of the data exchange process using remote channels. Each process handles a designated partition and interacts with the corresponding channel. Black arrows indicate data being put into the channels by the sending processes, while blue arrows represent the retrieval (take) operations performed by receiving processes to update their ghost nodes value.

Figure 5.5 illustrates the data exchange mechanism using `channel.Dict`. Each channel is visually represented by two parallel horizontal lines of the same color, associated with a specific partition (indicated by the corresponding color). The colored data chunks within each channel represent the halo data to be exchanged between partitions. The notation $Data_{\xi,\zeta}$ denotes that the data is sent by partition ξ and should be received by partition ζ . After each process completes the update of its local solution U_{P_i} , it initiates the data exchange process. Specifically:

- *Put Operation* (Black Arrows): The process places the relevant portion of U_{P_i} onto the remote channel linked to the target partition (specified by the key in `exchangeDataDict_P`). The portion of U_{P_i} to be sent is determined based on the `send_indices` from `exchangeDataDict_P`, which specifies the node values that need to be shared with neighboring partitions.
- *Take Operation* (Blue Arrows): The process retrieves the halo data $U_{ghost(P_i)}$ from the remote channel linked to the current partition (i.e., the partition index P_i that the process is responsible for). The retrieved data is then used to update the halo nodes in the local solution. This update is performed by referencing the `replace_indices` in `exchangeDataDict_P`, using the key corresponding to the sending partition's index (i.e., the partition from which the data originated).

This design ensures that each process can obtain the correct halo data from the appropriate neighboring partitions, maintaining data consistency across the computational domain before advancing to the next time step.

To implement the data exchange mechanism illustrated in Figure 5.5, the process is divided into two main stages, each accompanied by corresponding code snippets:

1. Establishing and Distributing the Remote Channel Dictionary (Preprocessing Stage):
Before entering the processing loop, create a global dictionary of remote channels is created and distributed across all worker processes.
2. Performing Data Exchange During Processing (Processing Stage): Once the remote channel dictionary is established, each process performs data exchange operations at every time step. This stage involves **put** and **take** operations.

The following code snippet demonstrates how the remote channel dictionary (`channel_dict`) is created. As mentioned earlier, `channel_dict` is then distributed across all worker processes as part of the argument list when calling `pmap`, ensuring each worker has access to the global `channel_dict` inside the `hydro` function. The global dictionary `channel_dict` is initialized at line 2. This dictionary uses the partition index `iPart` as the key and stores a corresponding `RemoteChannel` instance as the value. The dictionary is then filled by looping over all partitions (`nParts`), where each partition index in the loop (`iPart`) is assigned a `RemoteChannel` instance.

```
1 #create channel dictionary
2 global channel_dict = Dict{Int, RemoteChannel}()
3     if nParts != 1
4         for iPart = 1 : nParts
5             channel_dict[iPart] = RemoteChannel(() -> Channel{Tuple{
6                 Int64, Vector{Float64}, Vector{Float64}}}(length(
7                     exchangeDataDict_List[iPart])), workers()[iPart])
6         end
7     end
```

In this implementation, `RemoteChannel` is constructed using the Julia function:

```
RemoteChannel(f,pid).
```

Where `f` is a function that, when executed on the specified worker process (`pid`), returns an `AbstractChannel` and `pid` specifies the worker process id on which the channel will be stored [13]. Here, the function `f` is explicitly defined as:

```
f=()→Channel{dataType}(dataSize)
```

where

- `dataType` is the structure of data passing through the channel:

```
Tuple{Int64, Vector{Float64},Vector{Float64}}
```

This tuple consists of:

1. The sending partition's index (`send_p_id`), which identifies the process responsible for the data.
2. A vector of p values, containing the relevant horizontal discharge data of U_{P_i} .
3. A vector of q values, similarly containing relevant vertical discharge data of U_{P_i} .

- `dataSize` is defined as:

```
dataSize=length(exchangeDataDict_List[iPart])
```

Here, `exchangeDataDict_List[iPart]` represents the `exchangeDataDict_P` for Partition `iPart`. The size of exchange data chunks (number of `Tuples`) in this partition is determined by the number of keys in `exchangeDataDict_P`, making `dataSize` the correct capacity for the channel.

By structuring the implementation this way, each worker process is assigned a `RemoteChannel` that stores partition-specific data.

Now that the channel dictionary (`channel_dict`) has been created, the next step is to distribute its reference across all worker processes. This can be done by passing it as an argument in `pmap` (as implemented in this work) or through other data-passing techniques such as `remotecall_fetch`³. Distributing a shared reference rather than independent copies ensures that all workers interact with the same `channel_dict`. If each worker maintained a local copy (using `@everywhere channel_dict`), no actual data exchange would occur, as processes would only modify their isolated versions of the channels. By sharing a single global reference, all processes can read from and write to the channels, regardless of where the channel is physically stored. This approach simplifies data management, as each process interacts only with its assigned partition and neighboring partitions during halo data exchanges.

With this setup in place, we now focus on the data exchange operations that take place within the `hydro` function at each time step. The following snippets outlines the data exchange process between partitions during parallel hydrodynamic computations.

The process begins by each process placing the computed local solution values into the channels associated with neighboring partitions (line 1 - 5).

```
1 for (dest_p_id, (send_indices, _)) in exchangeDataDict_P
```

³For further details, please refer to Julia Doc [13].

```

2      #partialPNew/partialQNew refer to U_P,ghost, Pi is the current
      partition index
3      put!(channel_dict[dest_p_id], (Pi, partialPNew[send_indices],
      partialQNew[send_indices]))
4  end

```

This is achieved by iterating through the `exchangeDataDict_P`, where each key represents the index of a neighboring partition (`dest_p_id`), to which data needs to be sent. The corresponding value provides the node indices (`send_indices`) in the current partition P_i that should be sent. The `put!` operation ensures that the correct portion of U_{P_i} is sent to the appropriate neighboring partitions. The `put!` function writes data into the `RemoteChannel` linked to the target partition `dest_p_id`. Each data entry consists of:

- The current partition index P_i identifying the sender.
- A subset of p values (`partialPNew[send_indices]`) from the sender's local solution.
- A subset of q values (`partialQNew[send_indices]`) from the sender's local solution.

By directing this data to the appropriate channel (`channel_dict[dest_p_id]`), the process ensures that neighboring partitions receive the necessary halo values to maintain consistency during computation.

Since all necessary halo values have been placed in the channels associated with their respective partitions, the next step is for each process to retrieve the data from its own channel. This begins with obtaining the correct channel for the current partition P_i on line 2 of the following code snippet. Once the correct channel is identified, the process retrieves all incoming data chunks from neighboring partitions using the `take!` operation (line 4).

```

1      # Get the current channel from channel dictionary
2  chnl = channel_dict[Pi]
3  # Retrieve all the data from the current channel corresponding to
    halo updates
4  result = [take!(chnl) for _ in keys(exchangeDataDict_P)]

```

It is important to note that, although Julia provides other data retrieval methods such as `fetch`, `take!` is used here because it removes the retrieved data from the channel, preventing it from becoming full [13]. In this implementation, the channel capacity is explicitly set based on the number of halos, with a maximum size of: `number of halos × sizeof(Tuple)`. If the buffer reaches its limit, new `put!` operations are blocked until space becomes available. By clearing the channel after each retrieval, `take!` ensures non-blocking communication and smooth data exchange across time steps.

Using the partition index from the retrieved data—which retains the same tuple structure sent during the `put!` operation, the process queries `exchangeDataDict_P` to obtain the corresponding replacement indices (`replace_indices`). The received values are then placed at these indices in the local $U_{P_i,ghost}$. This guarantees that each partition retains the correct halo information from its neighboring partitions before proceeding to the next iteration of the simulation.

```

1  # Update the halo node values of the current partition with the
    received data
2  for (send_p_id, pVec, qVec) in result
3      (_, replace_indices) = exchangeDataDict_P[send_p_id]
4      partialPNew[replace_indices] .= pVec

```

```

5     partialQNew[replace_indices] .= qVec
6 end

```

5.6.3.5. Synchronization before next Time Step

In the distributed implementation in Algorithm 3, Step 6 (explicit synchronization) is required due to the structure of the parallel execution, in which the entire time-stepping loop is wrapped inside a single `pmap` call. This structure is used in the current implementation and is shown in Algorithm 3. An alternative strategy—shown below in Algorithm 4—would place the `pmap` call inside the time loop, allowing each time step to be processed independently in parallel:

Algorithm 5 Alternative approach.

```

1: Processing:
2: for each time step do
3:   Step 3: Call pmap(hydro, arg_List) for all partitions in parallel
4: end for
5: function HYDRO(arg_Pi)
6:   Step 4 and 5: Update  $p$ ,  $q$  and  $h$ 
7: end function

```

This is a typical approach in parallel computing and offers several potential advantages. It provides a straightforward way to integrate additional parallelized components, such as thermal dynamics and ice dynamics. Furthermore, it eliminates the need for an explicit synchronization mechanism, as `pmap` automatically forces synchronization at the end of

each function call, ensuring that all processes are in sync before advancing to the next time step. While this approach may offer benefits in terms of modularity and synchronization handling, the current implementation (shown in Algorithm 3) follows a different structure due to specific considerations for computational efficiency. As this structure is not explicit reflected in Algorithm 3, we simplify it here for the reader’s clarity.

Algorithm 6 Simplified current approach.

```

1: Processing:
2: Step 3: Call pmap(hydro, arg_List) for all partitions in parallel
3: function HYDRO(arg_Pi)                                ▷ put time iterations inside the hydro function
4:   for each time step do
5:     Step 4 and 5: Update p, q and h
6:     Step 6: Synchronize after completing updates
7:   end for
8: end function

```

Although `pmap` is a powerful tool for parallelizing functions with heavy computations, it also involves an implicit process to distribute tasks among workers. Specifically, `pmap` assigns each worker an independent tuple of arguments (`arg_Pi`), which serves as the input when executing the `hydro` function, as mentioned in the previous section. When `pmap` is placed inside the time loop, this task distribution process repeats at every time step, leading to an accumulation of task assignment overhead across all workers. As the number of workers increases, more communication is required to broadcast tasks, resulting in higher communication overhead and potential performance degradation. Performance analysis including a speed up figure of this approach can be found in the Appendix G.

To achieve better speedup and time efficiency in parallel execution, we wrap the time iterations inside the `pmap` function (shown in Algorithm 5), ensuring that the task assignment process occurs only once, rather than at every time step. However, this approach introduces synchronization uncertainties, as each process executes all time steps within the `hydro()` function. Since each process performs multiple time-step computations independently, it is necessary to ensure that halo data exchange (`put!` and `take!`) occurs at the same time step across all workers. Failure to maintain synchronized data exchange could result in workers using incorrect information, leading to inconsistencies in the simulation.

Unlike the `MPI.jl` package, which provides built-in synchronization functions such as `barrier()`, the `Distributed.jl` package does not include a dedicated synchronization mechanism based on investigation. Therefore, a user-defined synchronization function is necessary to ensure all workers remain in sync across time steps. In this implementation, one worker is selected as the control worker, with a user defined id (`control_id`), responsible for managing synchronization. This process involves two `RemoteChannel` objects:

- `barrier` – Used by workers to send a completion signal to the control worker.
- `notify` – Used by the control worker to signal all workers to proceed to the next time step.

The whole synchronization process works as follows with accompanying code shown in the `sync_barrier()` code below:

1. Each worker executes its assigned computations within the `hydro()` function
2. Upon completing its task for the current time step, each worker sends a completion signal to the control worker by calling `put!(barrier, 1)` (Line 7).

3. The control worker gathers completion signals from all workers by clearing the barrier channel using `take!(barrier)` in a loop of workers (Lines 3-5).
4. Once all signals are received, the control worker sends a notification signal via another remote channel called `notify`, using `put!(notify, 1)` for each worker (Lines 6-8).
5. All other workers wait for the notification signal and resume execution only after receiving the signal from control worker `take!(notify)` (Line 10).

The following function, `sync_barrier()`, implements this synchronization strategy:

```
1  @everywhere function sync_barrier(barrier, notify, control_id)
2  if myid() == control_id
3      for _ in 1:(nworkers() - 1)
4          take!(barrier)
5      end
6      for _ in 1:(nworkers() - 1)
7          put!(notify, 1)
8      end
9  else
10     put!(barrier, 1)
11     take!(notify)
12 end
13 end
```

5.6.4. Performance Analysis

Now that the parallel implementation has been described step by step, we can move on to evaluate its performance and effectiveness. The distributed parallel strategy, implemented

using Julia’s `Distributed.jl` package, was tested on a single-node workstation equipped with 16-core 3.1 GHz Intel Xeon(R) W5-3435X processors (with Hyper-Threading enabled) and 32 GB of RAM. This evaluation focuses solely on local parallel performance, analyzing speedup, scalability, and efficiency across multiple cores.

5.7. Distributed Computing Parallel Speedup and Test Case Description

The efficiency of the parallel implementation is assessed by measuring the speedup as the number of worker processes increases. Speedup is defined as:

$$S(P) = T_{serial}/T_{parallel}, \quad (5.1)$$

where P represents the number of processes, T_{serial} is the execution time of the best sequential algorithm using a single worker process, and $T_{parallel}$ is the execution time when running with n worker processes. In an ideal scenario, where the parallel algorithm scales linearly without communication overhead, the maximum achievable speedup would be the linear speed up limit:

$$S(P) = T_{serial}/(T_{serial}/P) = P. \quad (5.2)$$

To evaluate whether the parallel implementation produces results identical to the serial execution and whether it achieves linear speedup as defined by Equation (5.2), we first test the rectangular river model Figure 4.1 with corresponding mesh (Figure 4.2). This mesh consists of 2,111 nodes and 4,000 elements and was previously used in Chapter 4, applying the same initial and boundary conditions. The simulation is run for 500 iterations, and the results of the distributed parallel implementation (3 worker processes) are compared against the serial execution (1 process) to verify numerical consistency. Figures 5.6 and 5.7 show a

comparison of water depth and discharge between the serial and parallel implementations at the 500 time step.

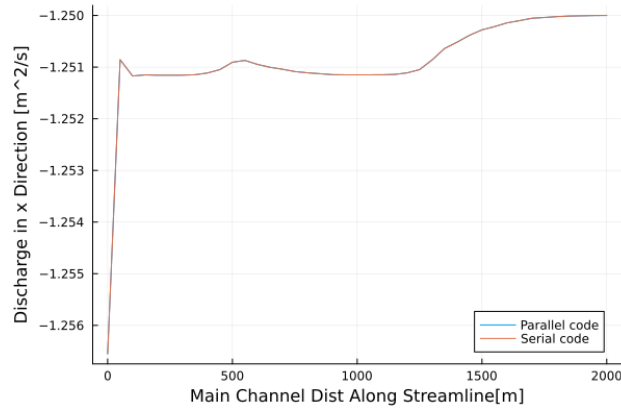


Figure 5.6. Comparison between parallel (3 worker processes) and serial solutions for water discharge in the x -direction at 500 time step.

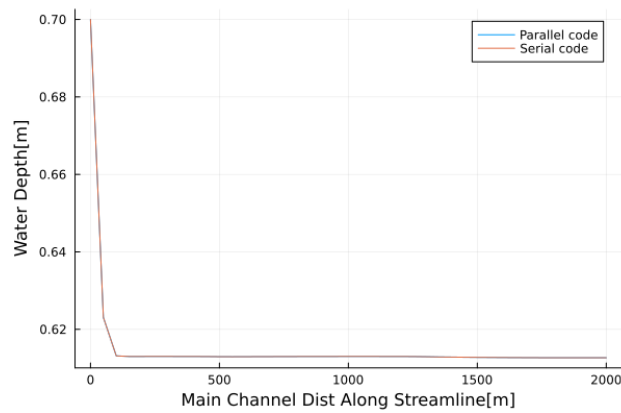


Figure 5.7. Comparison between parallel (3 worker processes) and serial solutions for water depth at 500 time step.

As shown in the figure, the results from the parallel and serial implementations closely match, with a maximum difference on the order of 10^{-6} . This small variation is within the

expected tolerance for floating-point computations in distributed numerical simulations, confirming that the parallel code preserves overall numerical accuracy and behaves consistently with the serial version.

While the parallel implementation produces accurate results, Figure 5.8 illustrates that the observed speedup does not follow the ideal linear factor of P as defined by Equation 5.2.

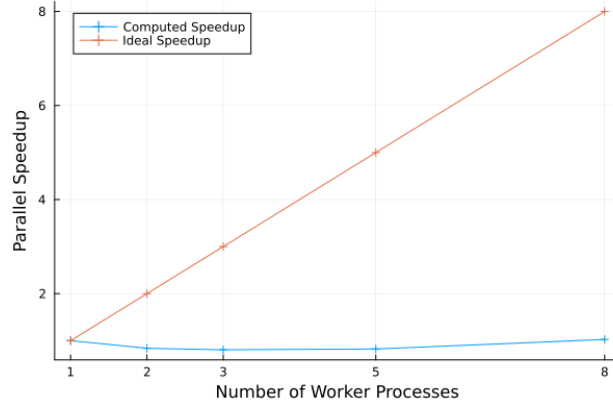


Figure 5.8. Parallel speedup. An ideal scaling curve is included for reference.

As seen in the figure, the speedup remains significantly below the theoretical maximum, indicating that performance is limited by communication overhead, synchronization delays, and the fraction of serial execution in the program. This result is reasonable because, in practical parallel implementations, a portion of the computation remains serial, and Amdahl's law puts a limit on speedup [28]. According to Gustafson's Law, increasing the problem size reduces the impact of serial execution, allowing the parallel portion to dominate overall computation and achieve better speedup [23].

To further investigate the scalability of the parallel approach, we refine the mesh and increase the number of nodes, thereby increasing the number of unknowns (p, q and h)

updated at each time step. By expanding the computational workload, parallel execution becomes the dominant factor, making it possible to observe speedup trends closer to linear scaling. To assess parallel efficiency in practice, the same river model from Figure 4.1 (2000 m in length and 200 m in width) was used, but with a refined mesh to increase computational complexity. By reducing the average triangular element size, the total number of elements increased from 4,000 to 930,496, while the number of nodes rose from 2,111 to 465,349. This refinement significantly expanded the number of unknowns 1296,047, ensuring a sufficiently large problem size for evaluating parallel performance.

To evaluate parallel performance, speedup and memory efficiency were measured using the refined river model over 500 time iterations. The execution times for both serial and parallel runs were averaged over 10 executions. The parallel execution time ($T_{parallel}$) was measured for increasing numbers of worker processes, where the number of worker processes corresponds to the total number of parallel tasks (not including the main process when $n > 1$). The speedup was then computed followed Equation (5.1) in Figure 5.9.

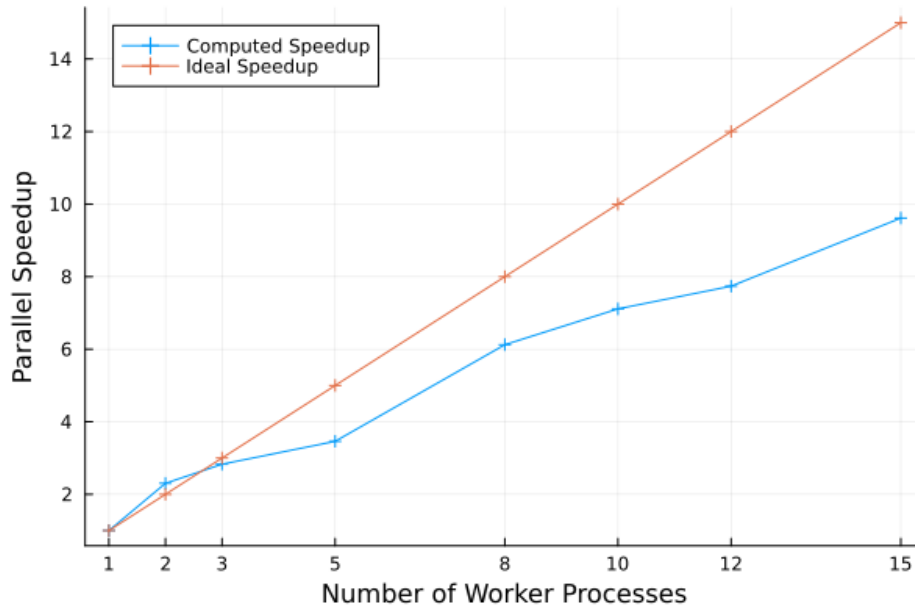


Figure 5.9. Parallel speedup. An ideal scaling curve is included for reference.

As shown in Figure 5.9, the computed speedup is consistently below the ideal maximum speedup, except for a slight outlier at 2 worker processes, where the observed speedup slightly exceeds expectations. As the number of worker processes increases, the gap between the ideal and computed speedup increases, primarily due to communication overhead and synchronization costs. However, despite this deviation from the ideal trend, the computed speedup continues to increase with the number of processes, demonstrating that the parallel implementation remains effective in reducing computational time.

Memory efficiency is evaluated based on the worst-case memory usage among all processes over 500 time iterations. The measured memory efficiency is plotted in Figure 5.10, alongside the ideal efficiency curve for reference.

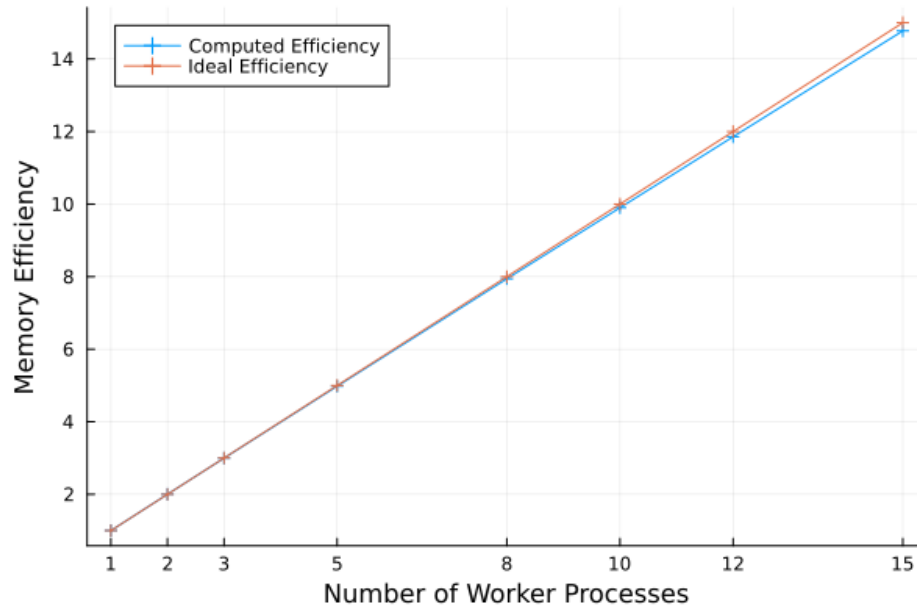


Figure 5.10. Memory efficiency computed using the worst-case memory usage among all processes over 500 time iterations. An ideal efficiency curve is included for reference.

The results indicate that memory efficiency closely follows the ideal trend, suggesting that the parallel implementation effectively distributes memory usage across processes.

6. Conclusion

This thesis presents the development of a Julia-based parallel FEM solver for the shallow water equations. The key contributions and potential future improvements are summarized below.

6.1. Summary of Accomplishments

The primary accomplishment of this work is the development of a finite element method (FEM) based simulation tool implemented in Julia for solving the shallow water hydrodynamic equations. The model is built on a mesh composed of first-order triangular elements, meaning it uses linear basis functions over linearly interpolated geometry, with the potential for future extension to higher-order basis functions and curved elements. The implementation was guided by the mathematical formulation and selected functionalities of an existing Fortran-based solver, but restructured in a modern, modular framework to improve readability, extensibility, and maintainability. To address the computational challenges associated with large-scale river simulations, this work also introduces parallel programming techniques to accelerate performance. Both shared-memory and distributed-memory parallel algorithms were implemented to support scalability across different hardware environments. A flexible mesh partitioning strategy was integrated and managed dynamically at runtime with the help of the Gmsh package, enabling the domain to be efficiently divided among processors without requiring manual preprocessing.

In addition to parallelism, several algorithmic improvements were made. One notable improvement is the application of the integration by parts method to handle boundary

conditions more effectively. This approach avoids the need to explicitly rotate the nodal discharge or velocity components (p, q, u and v) into the local normal and tangential directions along land and flux boundaries. By eliminating this rotation step and applying the boundary condition directly through the weak form, the solver achieves better numerical efficiency, cleaner code logic, and improved flexibility, especially when handling wet and dry node transitions near complex boundaries. Furthermore, the Julia implementation adopts a matrix-based approach for solving the system of ODEs, allowing the computation of $\mathcal{F}$ at all element nodes simultaneously using matrix operations. In contrast, the original Fortran code assembles the update equations for each node individually, effectively breaking down the matrix into explicit row-wise calculations for $\mathcal{F}$ at each local node in the element. This matrix-based formulation in Julia not only streamlines the implementation, but also uses Julia’s strengths in high-performance numerical computing.

6.2. Future Work

Several directions can be considered to extend and improve the current implementation:

- **Coupling with Thermal and Ice Dynamics:** The current solver focuses solely on hydrodynamics. Future work could incorporate modules for thermal and ice dynamics.
- **Support for Higher-Order Geometry and Basis Functions:** Extending the model to handle higher-order triangular or curved elements, along with higher-order basis functions, would improve accuracy, particularly for complex geometries.
- **Improved Time Integration Algorithm:** Other time integration schemes could be investigated, such as the implicit leapfrog method, which would enable dynamic time stepping and improve computational efficiency.

- **Enhanced Parallelization Across All Modules:** While parallelism has been applied to the hydrodynamic solver, future improvements could involve parallelizing the thermal and ice components as well.

In addition, GPU computing could be explored as a way to accelerate the code, particularly for smaller problem sizes where CPU-based parallelism may be less efficient. Note that Julia has libraries supporting GPU computing.

- **General Code Optimization:** Additional performance improvements can be achieved by optimizing algorithms and code architecture, minimizing memory allocation overhead, and improving the speed up in parallelization, all of which contribute to reducing computation time in large-scale simulations.

Appendix A: Rewriting the Hydrodynamic Equation in Conservative Form to Align with the Non-Conservative Structure

The process of deriving hydrodynamic equation into the form that align with non-conservative structure is relatively straightforward.

We begin with the original hydrodynamic equations.

$$\begin{aligned}\frac{\partial \eta}{\partial t} + \frac{\partial p}{\partial x} + \frac{\partial q}{\partial y} &= 0 \\ \frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(\frac{p^2}{h}\right) + \frac{\partial}{\partial y}\left(\frac{pq}{h}\right) &= \frac{\tau_{sx} - \tau_{bx}}{\rho} - gh\frac{\partial \eta}{\partial x} + \frac{1}{\rho}\left(\frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y}\right) \\ \frac{\partial q}{\partial t} + \frac{\partial}{\partial x}\left(\frac{pq}{h}\right) + \frac{\partial}{\partial y}\left(\frac{q^2}{h}\right) &= \frac{\tau_{sy} - \tau_{by}}{\rho} - gh\frac{\partial \eta}{\partial y} + \frac{1}{\rho}\left(\frac{\partial T_{xy}}{\partial x} + \frac{\partial T_{yy}}{\partial y}\right).\end{aligned}$$

By manipulating the second equation and substituting $\eta = h + Z_0$, we can proceed with the derivation.

$$\begin{aligned}\frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(\frac{p^2}{h}\right) + \frac{\partial}{\partial y}\left(\frac{pq}{h}\right) &= \frac{\tau_{sx} - \tau_{bx}}{\rho} - gh\frac{\partial \eta}{\partial x} + \frac{1}{\rho}\left(\frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y}\right) \\ = \frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(\frac{p^2}{h}\right) + \frac{\partial}{\partial y}\left(\frac{pq}{h}\right) &= \frac{\tau_{sx} - \tau_{bx}}{\rho} - gh\frac{\partial(h + Z_0)}{\partial x} + \frac{1}{\rho}\left(\frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y}\right).\end{aligned}\tag{A.1}$$

Since

$$gh\frac{\partial h}{\partial x} = \frac{\partial}{\partial x}\left(\frac{gh^2}{2}\right),\tag{A.2}$$

plugging this into the momentum equation gives us:

$$\begin{aligned}
& \frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(\frac{p^2}{h}\right) + \frac{\partial}{\partial y}\left(\frac{pq}{h}\right) = \frac{\tau_{sx} - \tau_{bx}}{\rho} - gh \frac{\partial \eta}{\partial x} + \frac{1}{\rho} \left(\frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y} \right) \\
& = \frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(\frac{p^2}{h}\right) + \frac{\partial}{\partial y}\left(\frac{pq}{h}\right) = \frac{\tau_{sx} - \tau_{bx}}{\rho} - gh \frac{\partial(h + Z_0)}{\partial x} + \frac{1}{\rho} \left(\frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y} \right) \\
& = \frac{\partial p}{\partial t} + \frac{\partial}{\partial x}\left(\frac{p^2}{h} + \frac{gh^2}{2}\right) + \frac{\partial}{\partial y}\left(\frac{pq}{h}\right) = \frac{\tau_{sx} - \tau_{bx}}{\rho} - gh \frac{\partial Z_0}{\partial x} + \frac{1}{\rho} \left(\frac{\partial T_{xx}}{\partial x} + \frac{\partial T_{yx}}{\partial y} \right).
\end{aligned} \tag{A.3}$$

Similarly, we can manipulate the third equation as

$$\begin{aligned}
& \frac{\partial q}{\partial t} + \frac{\partial}{\partial x}\left(\frac{pq}{h}\right) + \frac{\partial}{\partial y}\left(\frac{q^2}{h}\right) = \frac{\tau_{sy} - \tau_{by}}{\rho} - gh \frac{\partial \eta}{\partial y} + \frac{1}{\rho} \left(\frac{\partial T_{xy}}{\partial x} + \frac{\partial T_{yy}}{\partial y} \right) \\
& = \frac{\partial q}{\partial t} + \frac{\partial}{\partial x}\left(\frac{pq}{h}\right) + \frac{\partial}{\partial y}\left(\frac{q^2}{h} + \frac{gh^2}{2}\right) = \frac{\tau_{sy} - \tau_{by}}{\rho} - gh \frac{\partial Z_0}{\partial y} + \frac{1}{\rho} \left(\frac{\partial T_{xy}}{\partial x} + \frac{\partial T_{yy}}{\partial y} \right).
\end{aligned} \tag{A.4}$$

Appendix B: Rewriting the Hydrodynamic Equations to Explicitly Determine Components in the Non-Conservative Form

This appendix derives the non-conservative form of the hydrodynamic equations used in Chapter 3.

Defining the right hand side of the Equation (3.39) as the source term $\underline{S} = [S_1, S_2, S_3]$, we obtain the following:

$$\begin{aligned}\frac{\partial \eta}{\partial t} + \frac{\partial p}{\partial x} + \frac{\partial q}{\partial y} &= S_1 \\ \frac{\partial p}{\partial t} + \frac{\partial}{\partial x} \left(\frac{p^2}{h} + \frac{gh^2}{2} \right) + \frac{\partial}{\partial y} \left(\frac{pq}{h} \right) &= S_2 \\ \frac{\partial q}{\partial t} + \frac{\partial}{\partial x} \left(\frac{pq}{h} \right) + \frac{\partial}{\partial y} \left(\frac{q^2}{h} + \frac{gh^2}{2} \right) &= S_3.\end{aligned}\tag{B.1}$$

Let $c = \sqrt{gh}$ and focus on the derivative terms in the second equation of (B.1)

$$\begin{aligned}\frac{\partial}{\partial x} \left(\frac{p^2}{h} + \frac{gh^2}{2} \right) &= p \frac{\partial}{\partial x} \left(\frac{p}{h} \right) + \frac{p}{h} \frac{\partial}{\partial x} p + gh \frac{\partial}{\partial x} h \\ &= p \left(p \frac{\partial}{\partial x} \left(\frac{1}{h} \right) + \frac{1}{h} \frac{\partial}{\partial x} p \right) + \frac{p}{h} \frac{\partial}{\partial x} p + gh \frac{\partial}{\partial x} h \\ &= -p^2 \frac{\partial}{\partial x} h + \frac{p}{h} \frac{\partial}{\partial x} p + \frac{p}{h} \frac{\partial}{\partial x} p + gh \frac{\partial}{\partial x} h \\ &= \left(gh - \frac{p^2}{h^2} \right) \frac{\partial h}{\partial x} + 2u \frac{\partial}{\partial x} p \\ &= (c^2 - u^2) \frac{\partial}{\partial x} h + 2u \frac{\partial}{\partial x} p\end{aligned}\tag{B.2}$$

$$\begin{aligned}
\frac{\partial}{\partial y} \left(\frac{pq}{h} \right) &= \frac{p}{h} \frac{\partial}{\partial y} q + q \frac{\partial}{\partial y} \left(\frac{p}{h} \right) \\
&= \frac{p}{h} \frac{\partial}{\partial y} q + \frac{q}{h} \frac{\partial}{\partial y} p + qp \frac{\partial}{\partial y} \left(\frac{1}{h} \right) \\
&= \frac{p}{h} \frac{\partial}{\partial y} q + \frac{q}{h} \frac{\partial}{\partial y} p - \frac{qp}{h^2} \frac{\partial}{\partial y} h \\
&= u \frac{\partial}{\partial y} q + v \frac{\partial}{\partial y} p - uv \frac{\partial}{\partial y} h.
\end{aligned} \tag{B.3}$$

Following similar steps, we can express the partial derivative terms in the third equation of (B.1) as:

$$\begin{aligned}
\frac{\partial}{\partial x} \left(\frac{pq}{h} \right) &= u \partial_x q + v \partial_x p - uv \partial_x h \\
\partial_y (q^2/h + gh^2/2) &= (c^2 - v^2) \partial_y h + 2v \partial_y q.
\end{aligned} \tag{B.4}$$

Substitute (B.2), (B.3) and (B.4) back into (B.1) we can obtain

$$\begin{aligned}
\frac{\partial \eta}{\partial t} + \frac{\partial p}{\partial x} + \frac{\partial q}{\partial y} &= S_1 \\
\frac{\partial p}{\partial t} + (c^2 - u^2) \frac{\partial h}{\partial x} + 2u \frac{\partial p}{\partial x} + u \frac{\partial q}{\partial y} + v \frac{\partial p}{\partial y} - uv \frac{\partial h}{\partial y} &= S_2 \\
\frac{\partial q}{\partial t} + u \frac{\partial q}{\partial x} + v \frac{\partial p}{\partial x} - uv \frac{\partial h}{\partial x} + (c^2 - v^2) \frac{\partial h}{\partial y} + 2v \frac{\partial q}{\partial y} &= S_3.
\end{aligned} \tag{B.5}$$

Appendix C: Detail Result for Matrix Multiplication in the Non-Conservative Form

This appendix presents the detailed derivation of the matrix form resulting from applying the test function Φ' to the non-conservative form of the hydrodynamic equations in Chapter 3.

If we define

$$W_\xi A_\zeta = \begin{bmatrix} (w_\xi a_\zeta)^{11} & (w_\xi a_\zeta)^{12} & (w_\xi a_\zeta)^{13} \\ (w_\xi a_\zeta)^{21} & (w_\xi a_\zeta)^{22} & (w_\xi a_\zeta)^{23} \\ (w_\xi a_\zeta)^{31} & (w_\xi a_\zeta)^{32} & (w_\xi a_\zeta)^{33} \end{bmatrix} \quad W_\xi = \begin{bmatrix} (w_\xi)^{11} & (w_\xi)^{12} & (w_\xi)^{13} \\ (w_\xi)^{21} & (w_\xi)^{22} & (w_\xi)^{23} \\ (w_\xi)^{31} & (w_\xi)^{32} & (w_\xi)^{33} \end{bmatrix} \quad (\text{C.1})$$

we can get the detail result after matrix multiplication as following:

$$\begin{aligned} & \int_{\Omega^e} \left(\omega \frac{\Delta x}{2} \frac{\partial \Phi}{\partial x} W_x + \omega \frac{\Delta y}{2} \frac{\partial \Phi}{\partial y} W_y \right) \left(A_x \frac{\partial \tilde{U}^e}{\partial x} + A_y \frac{\partial \tilde{U}^e}{\partial y} - \tilde{S}^e(U) \right) d\Omega \\ &= \omega \frac{\Delta x}{2} W_x A_x (\mathbf{S}_{xx} \underline{u}^e(t)) + \omega \frac{\Delta y}{2} W_y A_x (\mathbf{S}_{yx} \underline{u}^e(t)) \\ & \quad + \omega \frac{\Delta x}{2} W_x A_y (\mathbf{S}_{xy} \underline{u}^e(t)) + \omega \frac{\Delta y}{2} W_y A_y (\mathbf{S}_{yy} \underline{u}^e(t)) \\ & \quad - \omega \frac{\Delta x}{2} W_x (\mathbf{S}_x^T \underline{s}^e(t)) - \omega \frac{\Delta y}{2} W_y (\mathbf{S}_y^T \underline{s}^e(t)) \end{aligned} \quad (\text{C.2})$$

$$\begin{aligned}
&= \omega \frac{\Delta x}{2} \begin{bmatrix} \mathcal{S}_{xx}^e((w_x a_x)^{11} \underline{u_1(t)} + (w_x a_x)^{12} \underline{u_2(t)} + (w_x a_x)^{13} \underline{u_3(t)}) \\ \mathcal{S}_{xx}^e((w_x a_x)^{21} \underline{u_1(t)} + (w_x a_x)^{22} \underline{u_2(t)} + (w_x a_x)^{23} \underline{u_3(t)}) \\ \mathcal{S}_{xx}^e((w_x a_x)^{31} \underline{u_1(t)} + (w_x a_x)^{32} \underline{u_2(t)} + (w_x a_x)^{33} \underline{u_3(t)}) \end{bmatrix} \\
&+ \omega \frac{\Delta y}{2} \begin{bmatrix} \mathcal{S}_{yx}^e((w_y a_x)^{11} \underline{u_1(t)} + (w_y a_x)^{12} \underline{u_2(t)} + (w_y a_x)^{13} \underline{u_3(t)}) \\ \mathcal{S}_{yx}^e((w_y a_x)^{21} \underline{u_1(t)} + (w_y a_x)^{22} \underline{u_2(t)} + (w_y a_x)^{23} \underline{u_3(t)}) \\ \mathcal{S}_{yx}^e((w_y a_x)^{31} \underline{u_1(t)} + (w_y a_x)^{32} \underline{u_2(t)} + (w_y a_x)^{33} \underline{u_3(t)}) \end{bmatrix} \\
&+ \omega \frac{\Delta x}{2} \begin{bmatrix} \mathcal{S}_{xy}^e((w_x a_y)^{11} \underline{u_1(t)} + (w_x a_y)^{12} \underline{u_2(t)} + (w_x a_y)^{13} \underline{u_3(t)}) \\ \mathcal{S}_{xy}^e((w_x a_y)^{21} \underline{u_1(t)} + (w_x a_y)^{22} \underline{u_2(t)} + (w_x a_y)^{23} \underline{u_3(t)}) \\ \mathcal{S}_{xy}^e((w_x a_y)^{31} \underline{u_1(t)} + (w_x a_y)^{32} \underline{u_2(t)} + (w_x a_y)^{33} \underline{u_3(t)}) \end{bmatrix} \\
&+ \omega \frac{\Delta y}{2} \begin{bmatrix} \mathcal{S}_{yy}^e((w_y a_y)^{11} \underline{u_1(t)} + (w_y a_y)^{12} \underline{u_2(t)} + (w_y a_y)^{13} \underline{u_3(t)}) \\ \mathcal{S}_{yy}^e((w_y a_y)^{21} \underline{u_1(t)} + (w_y a_y)^{22} \underline{u_2(t)} + (w_y a_y)^{23} \underline{u_3(t)}) \\ \mathcal{S}_{yy}^e((w_y a_y)^{31} \underline{u_1(t)} + (w_y a_y)^{32} \underline{u_2(t)} + (w_y a_y)^{33} \underline{u_3(t)}) \end{bmatrix} \\
&- \omega \frac{\Delta x}{2} \begin{bmatrix} \mathcal{S}_x^{eT}((w_x)^{11} \underline{s_1(t)} + (w_x)^{12} \underline{s_2(t)} + (w_x)^{13} \underline{s_3(t)}) \\ \mathcal{S}_x^{eT}((w_x)^{21} \underline{s_1(t)} + (w_x)^{22} \underline{s_2(t)} + (w_x)^{23} \underline{s_3(t)}) \\ \mathcal{S}_x^{eT}((w_x)^{31} \underline{s_1(t)} + (w_x)^{32} \underline{s_2(t)} + (w_x)^{33} \underline{s_3(t)}) \end{bmatrix} \\
&- \omega \frac{\Delta y}{2} \begin{bmatrix} \mathcal{S}_y^{eT}((w_y)^{11} \underline{s_1(t)} + (w_y)^{12} \underline{s_2(t)} + (w_y)^{13} \underline{s_3(t)}) \\ \mathcal{S}_y^{eT}((w_y)^{21} \underline{s_1(t)} + (w_y)^{22} \underline{s_2(t)} + (w_y)^{23} \underline{s_3(t)}) \\ \mathcal{S}_y^{eT}((w_y)^{31} \underline{s_1(t)} + (w_y)^{32} \underline{s_2(t)} + (w_y)^{33} \underline{s_3(t)}) \end{bmatrix}
\end{aligned}$$

Appendix D: Applying the Divergence Theorem to the Continuity Equation with Galerkin Testing

This appendix presents the derivation of the matrix form resulting from applying the test function Φ^e to the conservative form of the continuity equation. By applying the divergence theorem, the surface integral over the boundary of the element naturally introduces a boundary term, leading to the identification of a natural boundary condition in the finite element formulation.

$$\int_{\Omega^e} \Phi^e \left(\frac{\partial h}{\partial t} + \nabla \cdot \vec{Q} \right) d\Omega = \int_{\Omega^e} \Phi^e \frac{\partial h}{\partial t} d\Omega + \Phi^e \int_{\Omega^e} \nabla \cdot \vec{Q} d\Omega. \quad (\text{D.1})$$

To reduce the order of the derivative in the second term on the right hand, we apply the product rule to shift the gradient from the flux vector $\vec{Q}$ onto the test function Φ^e . This gives:

$$\int_{\Omega^e} \Phi^e (\nabla \cdot \vec{Q}) = - \int_{\Omega^e} \nabla \Phi^e \cdot \vec{Q} d\Omega + \oint_{\Gamma_Q^e} \Phi^e (\vec{Q} \cdot \hat{n}) d\Gamma^e, \quad (\text{D.2})$$

where $\oint_{\Gamma_Q^e} \Phi^e (\vec{Q} \cdot \hat{n}) d\Gamma^e$ represents the weighted boundary flux, which can be used to impose boundary conditions.

Then we define $q_n = \vec{Q} \cdot \hat{n}$ as the user-defined scalar boundary flux, and by expanding the dot product $\nabla \Phi^e \cdot \vec{Q}$ as:

$$\begin{bmatrix} \frac{\partial \Phi^e}{\partial x} \\ \frac{\partial \Phi^e}{\partial y} \end{bmatrix} \cdot \begin{bmatrix} p \\ q \end{bmatrix} = \frac{\partial \Phi^e}{\partial x} p + \frac{\partial \Phi^e}{\partial y} q.$$

Equation (D.2) can be rewritten as:

$$\int_{\Omega^e} \underline{\Phi}^e (\nabla \cdot \vec{Q}) = - \int_{\Omega^e} \left(\frac{\partial \underline{\Phi}^e}{\partial x} p + \frac{\partial \underline{\Phi}^e}{\partial y} q \right) d\Omega + \oint_{\Gamma_Q} \underline{\Phi}^e q_n d\Gamma. \quad (\text{D.3})$$

Here, the boundary integral $\oint_{\Gamma_Q^e} \underline{\Phi}^e q_n d\Gamma^e$ represents the weighted scalar boundary flux q_n , which is used to apply boundary conditions along the flux boundary Γ_Q^e .

Substituting the expression for the divergence term from Equation (D.2) into the residual form of the continuity equation (3.56), we obtain the governing equation for the continuity equation with Galerkin testing and the boundary integral term as:

$$\mathcal{F} = \int_{\Omega^e} \underline{\Phi}^e \left(\frac{\partial h}{\partial t} + \nabla \cdot \vec{Q} \right) d\Omega = \int_{\Omega^e} \underline{\Phi}^e \frac{\partial h}{\partial t} d\Omega - \int_{\Omega^e} \left(\frac{\partial \underline{\Phi}^e}{\partial x} p + \frac{\partial \underline{\Phi}^e}{\partial y} q \right) d\Omega + \oint_{\Gamma_Q^e} \underline{\Phi}^e q_n d\Gamma^e. \quad (\text{D.4})$$

Appendix E: Overview of METIS and SimplePartition Algorithms

This appendix provides a brief overview of METIS and SimplePartition, two domain discretization methods available in the Gmsh library for partitioning the mesh for parallel computation.

E.1. SimplePartition Plugin

Simple partition¹ method partitions the mesh by dividing it into a specified number of slices along the chosen axes (X , Y , or Z). The number of slices in each direction is user defined, for example, setting `NumSlicesX = 4`, `NumSlicesY = 3`, and `NumSlicesZ = 1` will divide the domain into $4 \times 3 \times 1$ partitions. Additionally, the distribution of these slices can be controlled using mapping functions. For instance, setting `MappingX = t` results in equidistant slices along the x -axis. This approach is computationally efficient and well-suited for structured and regularly shaped domains. However, for complex or irregular geometries, it may lead to non-optimal load balancing and increased interprocess communication due to rigid partition boundaries. Partitioning using the SimplePartition plugin in Julia can be performed with the following commands:

```
gmsh.plugin.setNumber("SimplePartition", "NumSlices $\xi$ ", N)
```

```
gmsh.plugin.run("SimplePartition")
```

¹For more details, see the Gmsh source code: <https://gitlab.onelab.info/gmsh/gmsh>

The first line specifies the number of slices along the ξ -axis (`NumSlices $\xi$  = N`), while the second line executes the partitioning operation based on the defined settings.

E.2. METIS

METIS employs a multilevel partitioning strategy, first introduced by Karypis and Kumar in 1998 [27]. The algorithm partitions a large mesh G through a three-step process:

1. Coarsening: The original mesh G is transformed into a coarse approximation G_c by collapsing adjacent vertices and edges, reducing the number of nodes while preserving the overall structure [27]. This step simplifies the partitioning problem while maintaining connectivity information.
2. Partitioning: The coarse mesh G_c is then divided into k partitions using a user-defined approach, either Recursive Bisection or K-Way Partitioning, depending on the desired trade-off between computational efficiency and load balancing.
 - K-way partitioning: G_c is directly divided into k parts in a single step. This approach is generally faster but may result in less balanced partitions
 - Recursive Bisection: G_c is repeatedly bisected until k partitions are reached. This method often produces better load balancing but requires more iterations [27].

The partitioning algorithm in Julia can be selected using the following command:

```
gmsh.option.setNumber("Mesh.MetisAlgorithm", 1)
```

where 1 specifies the recursive bisection algorithm and 0 selects the K-way partitioning approach.

3. Uncoarsening and Refinement: The partitioned coarse mesh is mapped back to the original mesh G , with refinements applied to improve the quality of the partition [27].

Appendix F: METIS Partitioning Techniques: GUI and GEO File Methods

F.1. Graphical User Interface (GUI) Method

Users can manually specify the number of partitions, partitioning algorithm (recursive bisection or k-way partitioning), and refinement (uncoarsening) options through the Gmsh GUI. An example interface is shown in Figure F.1, where these options can be adjusted before exporting the partitioned mesh.

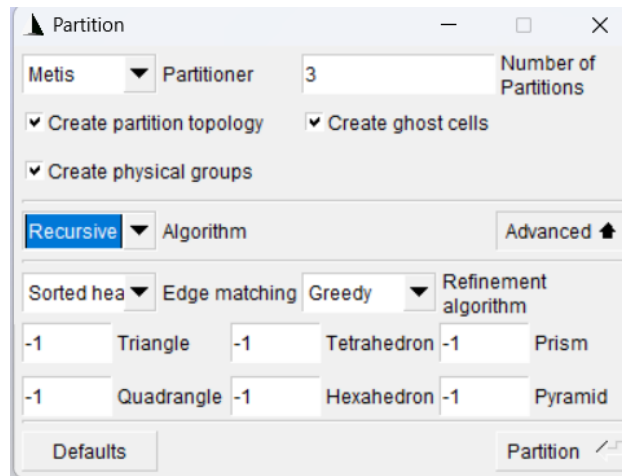


Figure F.1. Gmsh GUI.

F.2. geo File-based Method

Partitioning can also be predefined in a geo script, allowing users to generate a partitioned mesh before executing the simulation code. However, this approach assumes that the mesh is created within Gmsh from a geo file. In river simulations, the computational mesh

is often generated by external hydraulic modeling software, such as SMS, which directly produces an .msh file without an associated geo file. While it may be possible to reconstruct a geo file from an existing msh file, this process is neither straightforward nor practical for flexible, user-adaptable workflows. As this thesis focuses on runtime partitioning with Gmsh, we will not explore this method further. For more details, refer to the Gmsh tutorial t21¹.

¹Gmsh tutorial <https://gmsh.info/doc/texinfo/gmsh.html>

Appendix G: Performance Analysis for alternative Distributed Parallel Approach

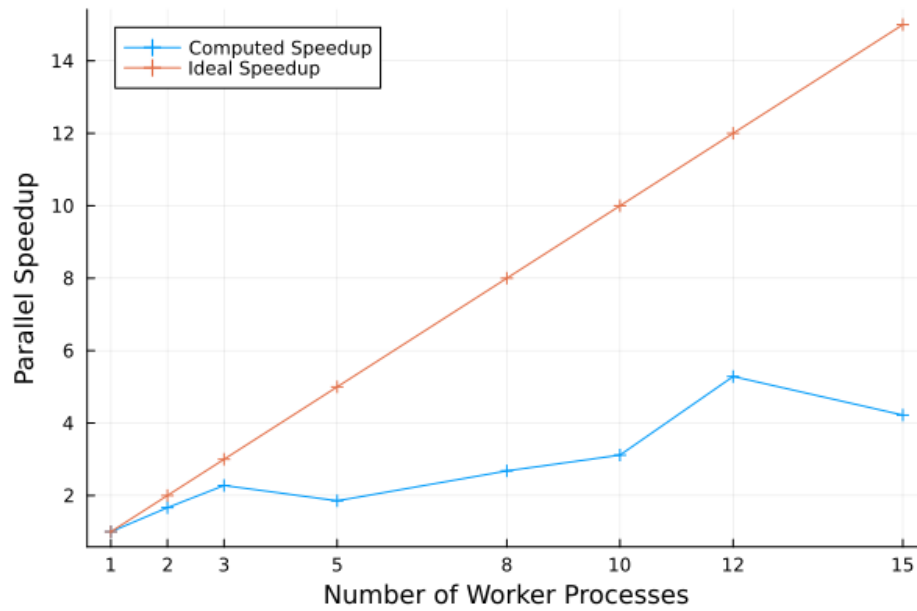


Figure G.1. Parallel Speedup. An ideal scaling curve is included for reference.

As shown in Figure G.1, the maximum speedup achieved by the parallel algorithm remains below 6, regardless of the increasing number of worker processes.

Appendix H: Comparative Analysis of HEC-RAS, Julia, and Fortran Results

This appendix provides additional logarithmic error results analyzing the simulation outputs from HEC-RAS, Julia, and Fortran implementations for the cold and hot cases examined in this thesis.

H.1. Cold start case

Figure H.1 presents the relative error in water depth, plotted on a logarithmic scale, comparing both the Julia and Fortran implementations against the HEC-RAS benchmark along the river centerline.

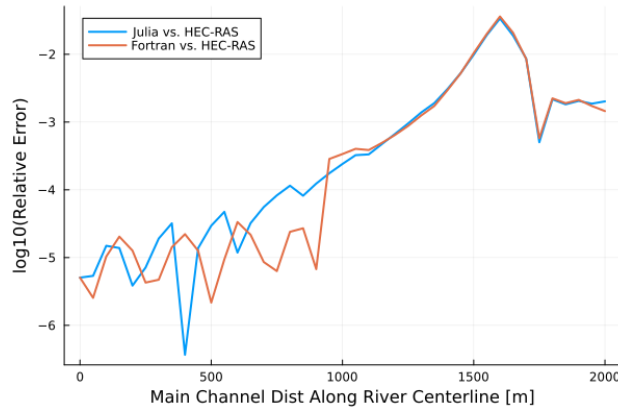


Figure H.1. Logarithmic relative error of water depth h , along the river centerline. Results from both the Julia and Fortran implementations are compared against HEC-RAS.

The relative error (in log scale) of the x -direction velocity component (u) between the three models is presented along the channel centerline in Figure H.2.

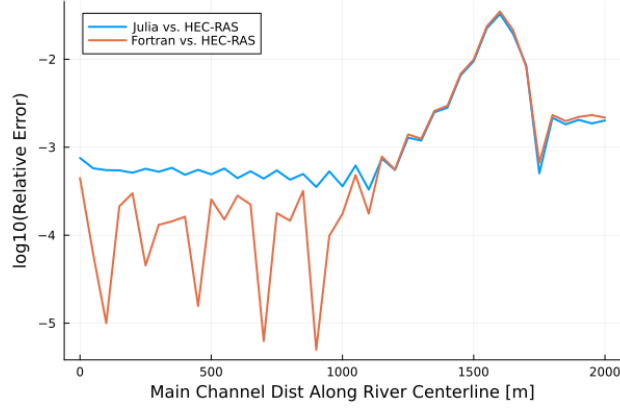


Figure H.2. Logarithmic relative error of x -direction velocity, along the river streamline. Results from both the Julia and Fortran implementations are compared against HEC-RAS.

Overall, both implementations show strong agreement with the reference HEC-RAS solution, with the Julia results closely matching the performance of the original Fortran code.

H.2. Hot start case

Figure H.3 presents the relative error in water depth computed from both the Julia and Fortran implementations, each compared against the HEC-RAS benchmark results.

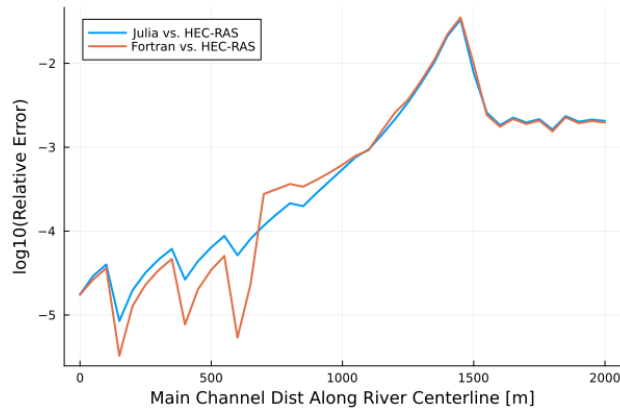


Figure H.3. Logarithmic relative error of water depth h , along the river centerline. Results from both the Julia and Fortran implementations are compared against HEC-RAS.

The relative error in x -direction velocity (u) for both the Julia and Fortran implementations, compared against the HEC-RAS results, is stated in figure below.

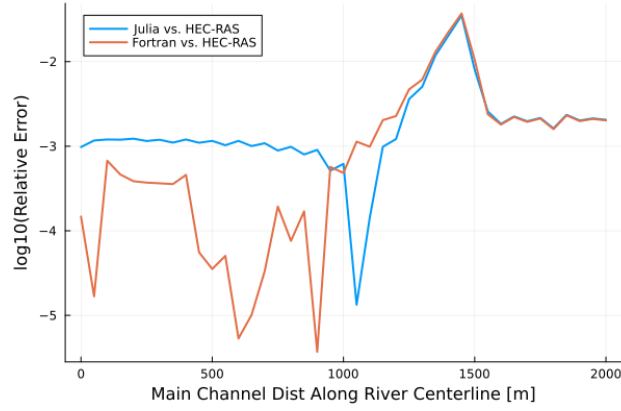


Figure H.4. Logarithmic relative error of x -direction water velocity, along the river centerline. Results from both the Julia and Fortran implementations are compared against HEC-RAS.

Both the Julia and Fortran implementations produce consistent results, with relative errors remaining low throughout the domain.

Bibliography

- [1] F. Hicks and P. Steffler, “Characteristic dissipative galerkin scheme for open-channel flow,” *Journal of Hydraulic Engineering*, vol. 118, no. 2, pp. 337–352, 1992.
- [2] A. H. Ghanem, “Two-dimensional finite element modeling of flow in aquatic habitats,” 1995.
- [3] I. Popescu, *Computational hydraulics*. IWA Publishing, 2014.
- [4] G. Birkhoff, R. S. Varga, and D. Young, “Alternating direction implicit methods,” in *Advances in computers*. Elsevier, 1962, vol. 3, pp. 189–273.
- [5] HEC-RAS website. [Online]. Available: <https://www.hec.usace.army.mil/software/hec-ras/>.
- [6] River1D website. [Online]. Available: <https://www.riverice.ca/modeling>.
- [7] MIKE-11 website. Available [Online]: https://docs.mikepoweredbydhi.com/engine-libraries/mike1d/mike1d_api/. Accessed: July 2025.
- [8] MIKE-21 website. [Online]. Available: <https://www.dhigroup.com/technologies/mikepoweredbydhi/mike2d-overland>.
- [9] SMS website. [Online]. Available: <https://aquaveo.com/software/sms/introduction>.
- [10] TELEMAC website. [Online]. Available: <https://hydro-informatics.com/numerics/telemac/telemac.htm>.

- [11] *CRISSP2D User's Mannual*, CEATI Project No. T012700-0401, Manitoba Hydro, 2022.
- [12] H. T. Shen, "Mathematical modeling of river ice processes," *Cold Regions Science and Technology*, vol. 62, no. 1, pp. 3–13, 2010.
- [13] *julia v1.11.2*, The Julia Project, Julia Programming Language, 2024.
- [14] P. Steffler and J. Blackburn, "Two-dimensional depth averaged model of river hydrodynamics and fish habitat," *River2D user's manual*, *University of Alberta, Canada*, 2002.
- [15] N. G. Adrien, *Computational hydraulics and hydrology: an illustrated dictionary*. CRC Press, 2003.
- [16] G. P. Nikishkov, "Introduction to the finite element method," *University of Aizu*, pp. 1–70, 2004.
- [17] J.-M. Jin, *The finite element method in electromagnetics*. John Wiley & Sons, 2015.
- [18] B. A. Finlayson and L. E. Scriven, "The method of weighted residuals—a review," *Appl. Mech. Rev.*, vol. 19, no. 9, pp. 735–748, 1966.
- [19] J. S. Hesthaven and T. Warburton, *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*. Springer Science & Business Media, 2007.
- [20] A. N. Brooks and T. J. Hughes, "Streamline upwind/petrov-galerkin formulations for convection dominated flows with particular emphasis on the incompressible navier-stokes equations," *Computer methods in applied mechanics and engineering*, vol. 32, no. 1-3, pp. 199–259, 1982.

- [21] G. J. Le Beau and T. E. Tezduyar, “Finite element computation of compressible flows with the supg formulation,” 1991.
- [22] A. Razoumov. (2021) Julia programming tutorial, simon fraser university. [Online]. Available: <https://wgpages.netlify.app/julia/>
- [23] B. Wilkinson and M. Allen, *Parallel programming: techniques and applications using networked workstations and parallel computers: numerical methods and modelling*. USA: Prentice-Hall, Inc., 1998.
- [24] I. Jeffrey, “Finite-volume simulations of maxwell’s equations on unstructured grids,” Ph.D. dissertation, University of Manitoba, 2011.
- [25] *Gmsh 4.14.0*, Gmsh, Gmsh, 2025.
- [26] S. T. Barnard and H. D. Simon, “Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems,” *Concurrency: Practice and experience*, vol. 6, no. 2, pp. 101–117, 1994.
- [27] W. Wang, S. Du, Z. Guo, and L. Luo, “Polygonal clustering analysis using multilevel graph-partition,” *Transactions in GIS*, vol. 19, no. 5, pp. 716–736, 2015.
- [28] B. Wilkinson, “Fundamentals of multicore hardware and parallel programming.” 2012.