Supplement of Geosci. Model Dev., 19, 3953–3987, 2026
https://doi.org/10.5194/gmd-19-3953-2026-supplement
© Author(s) 2026. CC BY 4.0 License.

Geoscientific Model Development
Open Access
EGU

Supplement of

SWEpy: an open-source GPU-accelerated solver for near-field inundation and far-field tsunami modeling

Juan Fuenzalida et al.

Correspondence to: Joaquín Meza (joaquin.meza@usm.cl)

The copyright of individual parts of the supplement might differ from the article licence.

Contents

List of Figures

List of Tables

S1 Introduction

S1.1 About SWEpy

The presented manual describes the theoretical background and operational guidelines of `SWEpy` (Shallow Water Equation Python solver). This software represents a high-performance numerical implementation of the Saint-Venant system, specifically designed to simulate near-field inundation and far-field tsunami propagation with computational efficiency.

`SWEpy` evolves from previous legacy implementations—formerly referred to as `ANSWER` (A Numerical Shallow Water Equation Resolution)—by integrating modern GPU acceleration through *CuPy* and *CUDA* kernels. While the original version relied on linear first-order convergence on CPUs, `SWEpy` implements a high-resolution Second-Order WENO (Weighted Essentially Non-Oscillatory) reconstruction operator. This architectural shift allows for the precise modeling of complex coastlines on unstructured triangular grids without the prohibitive computational cost associated with traditional solvers.

Reliable modeling of geophysical flows requires strict adherence to physical constraints. Any numerical method solving the Shallow Water Equations (SWE) must fulfill the *well-balanced property*—ensuring that the scheme does not generate spurious synthetic perturbations under lake-at-rest conditions—and must strictly preserve the positivity of the water depth at all times [4]. Furthermore, the method must be robust enough to withstand shock discontinuities while remaining computationally efficient. To guarantee these standards, the solver has been validated against a comprehensive suite of analytical benchmarks, laboratory experiments, and field tests [31].

To satisfy these rigorous requirements, `SWEpy` employs a Riemann-solver-free Central-Upwind scheme [8]. This approach is specifically formulated to preserve both the lake-at-rest state and the positivity of the water column [6]. To achieve high-order accuracy in time and ensure stability under non-linear conditions, the spatial discretization is coupled with a Strong Stability Preserving (SSP) Runge-Kutta (RK3) time integration scheme. Additionally, to resolve complex coastal features accurately, the domain is discretized employing unstructured triangular meshes [35], allowing for local refinement in areas of interest as detailed in [6].

S1.2 Model Definition

In this section, it will be defined the main variables that are going to be employed during this document. We will use normal variables q as scalar, while in bold $\mathbf{q}$ as vectors, i.e., we will have $\mathbf{q} = (q_1, q_2, q_3)^T$ as a vector of conserved variables.

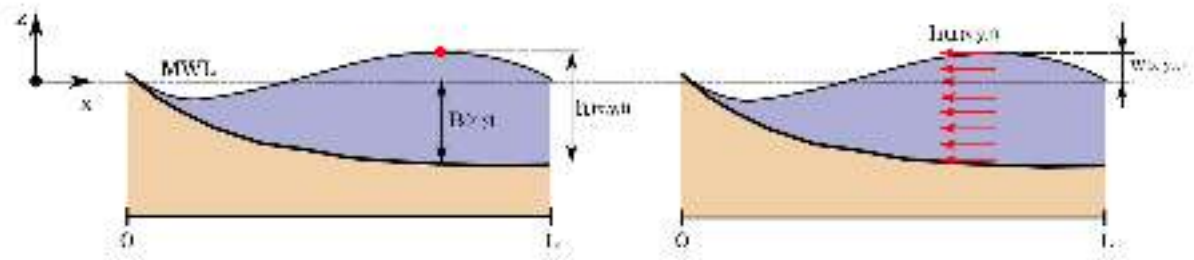


Fig. S1: Main variables of the method.

It is worth defining the following variables displayed in figure(S1):

- $w(x, y, t)$: Represents the surface elevation. It is measured from mean sea level to the water level.‘
- $h(x, y, t)$: Represents the water depth. It is measured from the bed elevation to the water surface.
- $u(x, y, t)$: Represents the velocity field along x-direction.
- $v(x, y, t)$: Represents the velocity field along y-direction.
- $B(x, y)$: Represents the bathymetry. It is measured from the mean sea level to the bottom floor.

There are other associated variables that results mandatory to highlight:

- $hu(x, y, t)$: Represents the flux-discharge along x-direction, $[m^2/s]$.
- $hv(x, y, t)$: Represents the flux-discharge along y-direction, $[m^2/s]$.

Lastly, it is important to mention that the line integral of (hu, hv) represents the caudal of water crossing the line:

$$Q = \int_\Gamma (hu, hv) \cdot d\mathbf{n} = \int_\Gamma (hu\, n_x + hv\, n_y)\, d\Gamma \tag{S1}$$

S1.3 The Shallow Water Equations

Assuming hydrostatic pressure condition, the SWE are obtained by integrating the Navier-Stokes equations over the water depth. A system of bi-dimensional equations is obtained, where the horizontal velocities are an average of the velocity along the water column. Neglecting the kinematic and turbulent terms, the SWE can be written as:

$$h_t + (hu)_x + (hv)_y = 0 \tag{S2}$$

$$(hu)_t + \left(hu^2 + \frac{1}{2}gh^2\right)_x + (huv)_y = -ghB_x + f \cdot hv - gn^2 \frac{(hu)\sqrt{(hu)^2 + (hv)^2}}{h^{7/3}} \tag{S3}$$

$$(hv)_t + (huv)_x + \left(hv^2 + \frac{1}{2}gh^2\right)_y = -ghB_y - f \cdot hu - gn^2 \frac{(hv)\sqrt{(hu)^2 + (hv)^2}}{h^{7/3}} \tag{S4}$$

Equation (S2) represents the mass balance due to the change of water height of the water column in a certain point. This value should be the discharge that the water column should have. Equations (S3) and (S4) represents the moment balance, and are related to the change in the discharge with the weight of the eater column, the Coriolis' force, and the bottom friction's force, etc.

The set of equations(S2), (S3) and (S4) can be written down on its conserved vector form:

$$\frac{\partial \mathbf{q}}{\partial t} + \frac{\partial \mathbf{f}(\mathbf{q})}{\partial x} + \frac{\partial \mathbf{g}(\mathbf{q})}{\partial y} = \mathbf{S}_B(\mathbf{q}) + \mathbf{S}_C(\mathbf{q}) + \mathbf{S}_F(\mathbf{q}) \tag{S5}$$

Where the previous variables represents:

$$\mathbf{q} = [h, hu, hv]^T \tag{S6}$$

$$\mathbf{f}(\mathbf{q}) = \left[hu, hu + \frac{1}{2}gh^2, huv\right]^T \tag{S7}$$

$$\mathbf{g}(\mathbf{q}) = \left[hv, huv, hv + \frac{1}{2}gh^2\right]^T \tag{S8}$$

$$\mathbf{S}_B(\mathbf{q}) = \left[0, -gh\frac{\partial B}{\partial x}, -gh\frac{\partial B}{\partial y}\right]^T \tag{S9}$$

$$\mathbf{S}_C(\mathbf{q}) = \left[0, f \cdot hv, -f \cdot hu\right]^T \tag{S10}$$

$$\mathbf{S}_F(\mathbf{q}) = \left[0, gn^2 \frac{(hu)\sqrt{(hu)^2 + (hv)^2}}{h^{7/3}}, gn^2 \frac{(hv)\sqrt{(hu)^2 + (hv)^2}}{h^{7/3}}\right]^T \tag{S11}$$

Where, $\mathbf{S}(\mathbf{q})$: represents the source term, $\mathbf{C}(\mathbf{q})$: the Coriolis term, $\mathbf{R}(\mathbf{q})$: represents the bottom friction term, f: is a vector that varies with the latitude and longitude, but it can be considered as constant with an average value of $10^{-4}[1/s]$, n: is the Manning's roughness coefficient. As an example for a gravelly channel a value of 0.025 should be employed. For a more detailed list of different Manning's coefficient, see [14].

S1.4 Software Architecture

`SWEpy` follows a modular programming paradigm, partitioning the complex finite volume solver into independent, reusable components or modules [34]. Each module handles different tasks, including grid loading, analysis configuration, preprocessing, time-step integration, spatial reconstruction, numerical flux and source term computations, or data output. This modular structure enhances user accessibility by isolating functionalities into self-contained units. It also promotes community-driven development through simple modification or extension of modules to accommodate additional source terms, boundary conditions, and other user-specific needs.

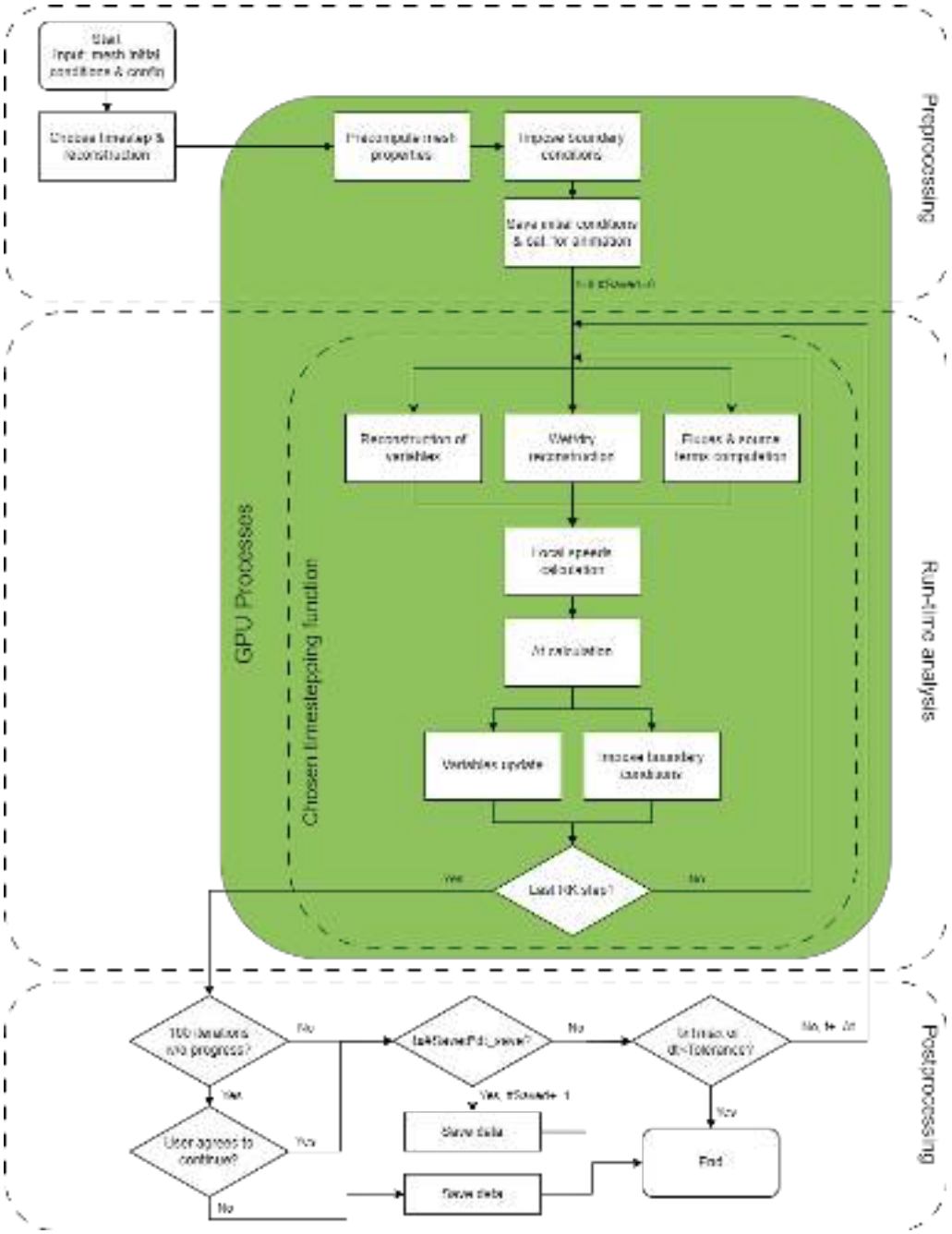


Fig. S2: Overview of the `SWEpy` software parallel structure and architecture. The green box contains tasks performed on the GPU. The inner segmented box contains tasks done by the chosen timestepping method. Outer segmented boxes indicate phases. Arrows going into/out of green box indicate CPU-GPU synchronization.

Developed in `Python`, `SWEpy` utilizes CuPy, a CUDA-accelerated counterpart to NumPy, to perform array-based operations on GPUs. This feature enables significant speedups in parallel computations without demanding expertise in low-level languages like C++ or CUDA [33]. By representing cell-based data (e.g., states and fluxes) as CuPy arrays indexed by cell number—such as storing the mean water level of cell Ω_3 in `Wj[3]`–`SWEpy` exploits data parallelism inherent to finite volume methods. Operations such as reconstructions and flux evaluations are vectorized across the entire grid, reducing CPU-GPU transfers and leveraging the efficiency of single instruction, multiple data (SIMD) execution [22].
This architecture not only accelerates `SWEpy`'s central-upwind scheme on unstructured triangular grids but also scales efficiently for large-scale simulations like the Maule tsunami, delivering high-resolution results on consumer-grade hardware. The runtime execution is organized into three phases: Preprocessing, Run-analysis, and Post-processing, as illustrated in the flowchart (Figure S2). Green boxes enclose GPU-

accelerated tasks, segmented boxes indicate time-stepping operations and general phases, and arrows over the green boxes highlight CPU-GPU synchronizations, providing a conceptual map of data flow and parallelism.

S1.4.1 Preprocessing

Loading input data: As illustrated in the uppermost part of Figure S2, the preprocessing phase initializes the simulation by preparing input data for efficient GPU execution, minimizing subsequent CPU-GPU transfers, and ensuring scalability for large unstructured grids. `SWEpy` accepts a user-generated triangular grid with vertex coordinates, connectivity arrays, and bathymetry values at vertices, along with initial conditions for the conserved variables, a list of boundary ghost cells with their type, and a configuration file specifying simulation parameters such as numerical tolerances, maximum runtime, gravitational acceleration, CFL number, and optional source term coefficients (e.g., Manning roughness or Coriolis parameter). This data is loaded into the *mesh* dictionary, which serves as the main data structure across the runtime.

Bathymetry generation: Bathymetry is then reconstructed using an interpolator chosen based on the selected reconstruction method. For example, a linear interpolator is used with the minmod scheme [7]. This process involves interpolating vertex values to both edge Gaussian points and the cell interiors, which are required for source term evaluations in equation (S26). The interpolation is performed using algebraic expressions, such as calculating the gradient of a plane defined by three vertices. These operations are efficiently executed across the GPU, with precomputed values stored to optimize performance.

Geometric computations on the mesh: Key geometric properties required for the numerical scheme–cell areas $|\Omega_j|$, edge lengths l_{jk}, barycenters (x_j, y_j), perpendicular heights r_{jk}, and second area moments I_x, I_y, I_{xy} *(Utilities.second_ moments)* [17]–are computed in parallel across all cells. Leveraging CuPy's array operations, these calculations exploit vectorized formulas based on vertex coordinates, enabling simultaneous evaluation on the GPU for the entire grid and accelerating setup for high-resolution domains typical in flood or tsunami modeling.

Stencil generation for high-order reconstruction For quadratic WENO reconstruction, a vectorized algorithm identifies first- and second-order neighbors without going through the data each time, flagging boundary cells lacking full stencils for fallback to minmod. Precomputation of least-squares matrices (right-hand side of (S35))–dependent solely on grid geometry–is performed using CuPy's `linalg` module to invert and multiply stacked matrices in parallel, storing local coefficients as arrays for rapid reuse in the run-analysis phase and reducing overhead in time-critical loops. By offloading these computations to the GPU, preprocessing establishes a data-parallel foundation, enabling `SWEpy` to handle complex simulations with minimized runtime delays.

S1.4.2 Run-analysis

The run-analysis phase, as represented in the outer central segmented box of Figure S2, constitutes the core of `SWEpy`'s time evolution, iteratively advancing the solution through spatial reconstructions, source term evaluations, flux computations, and state updates; all optimized for GPU parallelism to exploit the data-local nature of finite volume operations on unstructured grids. This phase leverages CuPy's array-based processing to perform calculations simultaneously across the entire domain.
Before entering the main loop, the program chooses the timestep function to be used according to user selection *(ShallowWater.choose_ timestep)*. This function is central as it contains the sequence of reconstructions, corrections, and calculations to be done in order to update the flux variables each iteration according to the central upwind method and the temporal discetization used. Defining new timestep functions allows for easy switching between reconstructions, source terms, and time discretizations.

Reconstruction: For minmod reconstruction, `SWEpy` computes gradients for each cell using neighboring average states, interpolating conserved variables $\boldsymbol{q}_j(M_{jk})$ at edge midpoints for flux calculations in equation (S23) and water surface values at vertices for wet/dry handling. Users can optionally include a diffusion coefficient, which is commonly used in minmod reconstruction, $\vartheta \in [1, 2]$ to control dissipation, where higher values reduce diffusion but may introduce oscillations. CuPy's Single Instruction Multiple Data (SIMD) capabilities enable parallel gradient computation and interpolation across all cells, replacing sequential iterations in Algorithm 1 with vectorized operations on the GPU for accelerated performance. This means that "for-loops" in alg. 1 are performed in parallel over all cells.
In the case of the WENO reconstruction, the model solves the LSQ linear systems S35 using the precomputed matrices to construct the reconstruction polynomials. These are used to reconstruct the values at the two gaussian midpoints of the sides of each cell (for use in the numerical flux calculation S23), the values in the three gaussian points inside the cell (for use in the source terms calculation S26), and the values of the water surface at the vertices (for use in the wet/dry reconstruction). Since we saved the stencils needed for each cell in the preprocessing phase, we can perform these reconstructions in parallel accelerated by the GPU processing. Once again, this means that iteration in alg. 2 is parallelized over all cells.
Boundary cells lacking full second-order neighbors, identified during preprocessing, default to minmod reconstruction. Wet/dry fronts are corrected by replacing invalid reconstructions (negative depths) using CuPy's fancy indexing to locate and adjust affected cells in parallel, executing Algorithm 3 via SIMD commands on the GPU without explicit loops. This means that "for-loops" in alg. 3 is replaced by SIMD commands for all cells, efficiently performed over the GPU. `SWEpy`'s modularity supports user-defined reconstruction operators, requiring only interpolated values at specified points, facilitating extensions like hybrid schemes while preserving GPU efficiency.

Source Terms: Bathymetry source terms are computed for all cells using equation (S26), leveraging preprocessed bathymetry reconstructions and the active spatial operator. These evaluations are vectorized across the grid on the GPU, ensuring parallel computation even for optional terms. The scheme's flexibility accommodates multiple source terms, allowing users to define custom modules (e.g., for rainfall, rheology, and/or turbulence) integrated seamlessly into the parallel workflow.

Local speeds and time steps: Using reconstructed states, velocities are desingularized and projected onto edge normals to compute local propagation speeds (S25) for all edges simultaneously on the GPU. If adaptive time stepping is enabled, Δt is determined enforcing the CFL stability limit given the local propagation speeds, maximizing step size with the objective of enhancing efficiency and minimizing diffusion (cf. (S39)).

Explicit-Euler integration applies (S23) via array operations on the GPU, with CPU synchronization only for timestep advancement (one cycle in Figure S2). The SSP RK4,3 methods [19], and the classical 4-step RK4 method both decompose into four scaled Forward-Euler steps, executed sequentially on the GPU to avoid iterative solvers and reduce overhead, optionally including friction corrections per stage. Modularity of the *timestep* function enables user-defined integration schemes to be integrated into the GPU workflow for future extensions.

Variable update and correction: Fluxes are assembled using local speeds and reconstructions, updating cell states via the CU scheme in equation (S23) in a single GPU-parallel operation, followed by ghost cell boundary imposition. For Manning friction, an intermediate semi-implicit correction adjusts the discharge vector-wise across the grid.
For multi-stage Runge-Kutta updates, intermediate states and fluxes are computed on the GPU and stored, with optional friction corrections applied per stage, minimizing synchronization and preserving third-order accuracy.

Boundary conditions:
The program imposes the user-defined boundary conditions on the ghost cells' states. Since each ghost cell has its definition, mixed boundary conditions at border cells can be defined and imposed all at once. The modularity of the program even allows for the definition of more complex functions that take care of the boundary conditions, like transport models for coupling other solvers' results, or higher-order extrapolations.
Boundary conditions currently implemented via ghost cells are: zero-order fully permeable (soft) border, by replicating the border cell's state at the neighboring ghost cell; and impermeable (wall) boundary, by replicating the border cell's water height, but inverting the flow direction, resulting in a zero-flow interface.
We highlight that periodic boundary conditions may also be modeled by setting boundary cells as neighbors of other boundary cells. For example, if we want the top and bottom borders of a channel to be periodic, the grid should consider the bottom cells as the upwards neighbors of the top border cells and vice versa. These kinds of boundary conditions are used in some of our experiments.

S1.4.3 Postprocessing

The post-processing phase, as shown in the lower portion of Figure S2, finalizes the simulation by managing data output and termination criteria, leveraging `SWEpy`'s modularity to enable customizable workflows that support both research analysis and operational monitoring. Users can integrate routines—predefined or custom—to export results at any runtime stage, facilitating immediate inspection and post-simulation processing.
For data saving, `SWEpy` offers built-in options to store initial conditions and bathymetry in .vtk format before run-analysis begins, with subsequent snapshots saved at user-defined intervals Δt_{save} until completion. This format is optimized for visualization tools like ParaView [2], enabling rendering of snapshots or animations to track flow evolution over unstructured grids. Time series of states for selected cells (e.g., virtual gauges in tsunami validations) are also supported at the same Δt_{save}. Integrated into the main runtime, this successive saving enables live visualization of emerging solutions, crucial for detecting anomalies in long-running simulations without halting progress. CuPy's GPU-accelerated NumPy equivalents (e.g., `save`, `savez`, `savetxt`) ensure efficient disk writes for large arrays, preserving performance even during high-frequency outputs.
Simulation termination occurs when the elapsed time reaches the user-specified target. Still, modularity allows for tailored criteria, such as divergence detection (e.g., if adaptive Δt drops below a threshold) or stagnation monitoring (e.g., negligible progress after a set number of iterations) by manipulation of the *run* functions, or definition of new ones. In our experiments, these controls prevented unproductive runs, complementing live visualization for on-the-fly adjustments. This flexibility extends `SWEpy`'s utility, allowing integration with external tools for automated error handling or real-time coupling in hybrid modeling setups.

S2 Installation and Environment

SWEpy is a high-performance numerical solver built upon the NVIDIA CUDA parallel computing architecture. Unlike traditional CPU-based software, its execution depends critically on the synergy between the host hardware, the GPU drivers, and the Python runtime environment. To ensure stability, reproducibility, and ease of dependency management, we strongly recommend using **Miniconda** or **Anaconda** to orchestrate the installation.
This chapter guides you through the complete setup process, from hardware verification to your first successful simulation.

S2.1 Prerequisites: Hardware & Drivers

Before attempting to install the software, it is essential to verify that your system meets the minimum hardware specifications for GPU acceleration. Open your terminal (Linux/WSL) or PowerShell (Windows) and execute the standard NVIDIA management utility:

```
nvidia-smi
```

Analyze the output table for two critical pieces of information:

- **GPU Architecture:** SWEpy requires a GPU with Compute Capability 6.0 or higher (Pascal microarchitecture, e.g., GTX 10-series, RTX 20/30/40-series, or Tesla P100/V100/A100). Older architectures (Kepler/Maxwell) may not support the atomic operations used in the finite volume kernels.
- **Driver Compatibility:** Look for the `CUDA Version` in the top-right corner. This indicates the maximum CUDA toolkit supported by your installed driver (e.g., 11.7 or 12.2). Ensure this version is **11.2 or higher** to guarantee compatibility with modern CuPy wheels.

S2.2 Environment Setup

S2.2.1 Installing Miniconda

If you do not have Conda installed, copy and run the following block to install it on Linux:

```
# Download and install Miniconda (Linux)
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh -b
```

Note: After installation, close and reopen your terminal to initialize Conda.

S2.2.2 Cloning the Repository

Download the SWEpy source code and enter the directory:

```
git clone https://github.com/joaquinmeza90/SWEpy.git
cd SWEpy
```

S2.2.3 Creating the Environment

Create and activate an isolated environment named `swepy_env`:

```
conda create -n swepy_env python=3.10 -y
conda activate swepy_env
```

S2.2.4 Installing Libraries (Step-by-Step)

We provide two installation methods. Choose **Option A (Conda)** if you want a stable, pre-compiled environment (Recommended). Choose **Option B (Pip)** if you prefer a lightweight setup or have specific version requirements.
Option A: Installation via Conda (Recommended) Run the following commands one by one to install the scientific stack from the `conda-forge` channel.

```
# 1. Install NumPy (Base mathematics)
conda install -c conda-forge numpy -y

# 2. Install SciPy (Advanced algorithms)
conda install -c conda-forge scipy -y

# 3. Install Matplotlib (Visualization)
conda install -c conda-forge matplotlib -y

# 4. Install CuPy (GPU Acceleration)
# Conda will automatically detect your driver and install the correct CUDA toolkit.
conda install -c conda-forge cupy -y
```

Option B: Installation via Pip If you prefer using `pip`, follow this sequence. Note that for CuPy, you must manually select the version matching your hardware.

```
# 1. Install NumPy
pip install numpy

# 2. Install SciPy
pip install scipy

# 3. Install Matplotlib
pip install matplotlib

# 4. Install CuPy (Select ONE based on your driver: Check your version with "nvidia-smi" and run the corresponding command:

   # For CUDA 11.x (Drivers 450.xx - 520.xx):
   pip install cupy-cuda11x

   # For CUDA 12.x (Drivers 525.xx and newer):
   pip install cupy-cuda12x
```

S2.3 Installation Diagnostics

Before running complex simulations, it is critical to verify that the Python environment can correctly communicate with the GPU hardware. The `SWEpy` repository includes a dedicated diagnostic script named `check_installation.py` located in the root directory. Execute it to validate your CUDA bindings:

```
import sys
import numpy as np

def print_header():
    print("\n" + "="*50)
    print("      SWEpy: GPU Acceleration Diagnostic")
    print("="*50)

def print_success():
    print("-" * 50)
    print("\033[ SUCCESS: SWEpy is ready for simulation.\033[0m")
    print("    You can now execute 'run_sim.py'.")
    print("=" * 50 + "\n")

def print_error(msg, detail=None):
    print("\n" + "-"*50)
    print(f"\033[91m[X] ERROR: {msg}\033[0m")
    if detail:
        print(f"    Details: {detail}")
    print("="*50 + "\n")

try:
    print_header()

    # 1. Import Libraries
    import cupy as cp
    print(f"[*] CuPy Version Detected: {cp.__version__}")

    # 2. Device Handshake & Property Retrieval
    # We use a robust method compatible with both old and new CuPy versions
    dev = cp.cuda.Device(0)

    try:
        # Modern CuPy method (Requires decoding bytes)
        props = cp.cuda.runtime.getDeviceProperties(dev.id)
        gpu_name = props['name'].decode('utf-8')
    except (AttributeError, KeyError):
        try:
            # Legacy CuPy method
            gpu_name = dev.name
        except AttributeError:
```

```
            gpu_name = "Unknown NVIDIA GPU"

    print(f"[*] Target GPU: {gpu_name}")

    # Optional: Display Compute Capability if available
    try:
        cc = dev.compute_capability
        print(f"[*] Compute Capability: {cc}")
    except:
        pass

    # 3. Arithmetic Core Test
    # Performs a simple vector addition to verify memory allocation and kernel execution
    print("[*] Verifying arithmetic kernels...", end=" ")

    x_gpu = cp.array([1.0, 2.0, 3.0])
    y_gpu = cp.array([4.0, 5.0, 6.0])
    z_gpu = x_gpu + y_gpu

    # 4. Result Validation
    expected = cp.array([5.0, 7.0, 9.0])
    if cp.allclose(z_gpu, expected):
        print("OK")
        print_success()
    else:
        print("FAILED")
        print_error("Arithmetic check failed.", "GPU returned incorrect values.")

except ImportError as e:
    print_error("CuPy library not found.",
                "Ensure you installed 'cupy-cudaXX' matching your driver.")

except Exception as e:
    print_error("Critical Runtime Error", str(e))
```

Listing 1: Installation Check Script

S2.3.1 Expected Output

If the installation is correct, the terminal should display the detected CuPy version and your specific GPU model, followed by the green success message. Typical output looks like this:

```
==================================================
      SWEpy: GPU Acceleration Diagnostic
==================================================
[*] CuPy Version Detected: 13.6.0
[*] Target GPU: NVIDIA GeForce XXX XXXX
[*] Compute Capability: 89
[*] Verifying arithmetic kernels... OK
--------------------------------------------------
[] SUCCESS: SWEpy is ready for simulation.
    You can now execute 'run_sim.py'.
==================================================
```

S2.3.2 Troubleshooting Common Failures

If the diagnostic script encounters an issue, it will halt execution and display a red error banner. The most frequent cause of failure is a mismatch between the NVIDIA system driver and the installed CuPy version.

Example Failure Output:

```
==================================================
      SWEpy: GPU Acceleration Diagnostic
==================================================
[*] CuPy Version Detected: 13.6.0
[*] Target GPU: Unknown / Error

```

```
--------------------------------------------------
[X] ERROR: Critical Runtime Error
    Details: cudaErrorInsufficientDriver: CUDA driver
    version is insufficient for CUDA runtime version
==================================================
```

Quick Diagnosis Guide:

- **Error:** `cudaErrorInsufficientDriver`
  Cause: Your NVIDIA GPU driver is too old for the installed version of CuPy.
  Solution: Update your GPU driver to the latest version (525.xx or higher) OR uninstall the current library and install the legacy version using `pip install cupy-cuda11x`.
- **Error:** `ModuleNotFoundError: No module named 'cupy'`
  Cause: The library is not installed, or you are not in the correct Conda environment.
  Solution: Activate the environment via `conda activate swepy_env` and reinstall the libraries following Section 2.2.4.
- **Error:** `Arithmetic check failed`
  Cause: The GPU was detected, but calculation results were incorrect. This indicates a severe hardware malfunction or corrupted memory.
  Solution: Reboot the system. If the problem persists, verify your hardware stability.

S2.4 Executing the SWEpy Solver

To run the simulation, navigate to the folder containing the `run_sim.py` script.

S2.4.1 Basic Usage

To run the simulation with the default settings (using the provided example, Forward Euler scheme and linear spatial reconstruction), simply execute:

```
python run_sim.py
```

S2.4.2 Custom Configuration

The solver accepts command-line arguments to specify the input folder and the numerical scheme.

Changing the Numerical Scheme: To use a different time integration or reconstruction method, use the `-m` flag. For example, to use the high-order Runge-Kutta 3 (RK3) method:

```
python run_sim.py -m rk3
```

Available Numerical Schemes: Use the `-m` flag to select the specific time integration and spatial reconstruction method:

- `fe`: Forward Euler + Minmod Reconstruction (Default strategy, 1st order time, 2nd order space)
- `cfe`: Forward Euler + Constant Reconstruction (1st order time and space)
- `rk3`: Runge-Kutta 4,3 + Minmod Reconstruction (3rd order time, 2nd order space)
- `feweno`: Forward Euler + WENO Reconstruction (1st order time, high-order space)
- `rk3weno`: Runge-Kutta 4,3 + WENO Reconstruction (High-order time and space)

Using Custom Input Data: To run a simulation using a different dataset, specify the path to the input folder:

```
python run_sim.py ./my_custom_inputs/ -m rk3weno
```

The output files (VTU format for Paraview) will be automatically generated in a `Paraview` folder within your current working directory.

S3 Numerical Framework

S3.1 Semi-discrete formulation

To ensure conservation of mass and momentum while effectively handling discontinuities such as shocks or wet/dry fronts prevalent in shallow water flows, FVM offer a robust numerical framework for solving the SWEs [27, 30, 43]. These methods integrate the governing equations over finite control volumes, approximating cell-averaged states and evolving them by computing fluxes across cell interfaces. This approach naturally captures shock waves, like hydraulic jumps or bore propagation, without requiring additional artificial viscosity [36], making it particularly suited for shallow water applications where abrupt changes in flow regime are common.
For enhanced well-balancing—ensuring exact preservation of steady states, especially over variable bathymetry—we reformulate the equations by substituting the water depth h with the surface elevation $w = h + B$ in the conserved variables vector $\boldsymbol{q}$, resulting in $\boldsymbol{q}(x, y, t) = (w, hu, hv)^T$. Thus, by using the system of equations presented in (S2), (S3), and (S4), equation (S5) will take the form:

$$\boldsymbol{q}_t + \boldsymbol{F}(\boldsymbol{q}, B)_x + \boldsymbol{G}(\boldsymbol{q}, B)_y = \boldsymbol{S_B}(\boldsymbol{q}, B) + \boldsymbol{S}(\boldsymbol{q}, B), \tag{S12}$$

with

$$\boldsymbol{F}(\boldsymbol{q}, B) = \left(hu,\ \frac{(hu)^2}{w - B} + \frac{1}{2} g (w - B)^2,\ \frac{(hu)(hv)}{w - B} \right)^\top, \tag{S13}$$

$$\boldsymbol{G}(\boldsymbol{q}, B) = \left(hv,\ \frac{(hu)(hv)}{w - B},\ \frac{(hv)^2}{w - B} + \frac{1}{2} g (w - B)^2 \right)^\top. \tag{S14}$$

$$\boldsymbol{S_B}(\boldsymbol{q}, B) = (0, -ghB_x, -ghB_y)^\top \tag{S15}$$

and $\boldsymbol{S}(\boldsymbol{q})$ represents additional source terms, such as Coriolis, friction, rheology, or turbulence. In this manual, we focus on bottom friction $\boldsymbol{S_F}(\boldsymbol{q}, B)$ and Coriolis effects $\boldsymbol{S_C}(\boldsymbol{q})$, critical for modeling dam-break scenarios and far-field tsunami propagation, respectively. Thus, the bottom friction–requiring semi-empirical closure laws, specifically the Manning-Strickler parameterization [13]– and coriolis terms are expressed as:

$$\boldsymbol{S_C}(\boldsymbol{q}) = \begin{bmatrix} 0 & fhv & -fhu \end{bmatrix}^T, \tag{S16}$$

$$\boldsymbol{S_F}(\boldsymbol{q}, B) = -gn^2 (w - B)^{-7/3} \left(\sqrt{(hu)^2 + (hv)^2} \right) \begin{bmatrix} 0 & (hu) & (hv) \end{bmatrix}^T \tag{S17}$$

We consider a triangular discretization of the polygonal spatial domain $\Omega = \bigcup_j \Omega_j$, including additional "ghost" cells for boundary conditions, as depicted in Figure S3 (bottom border).

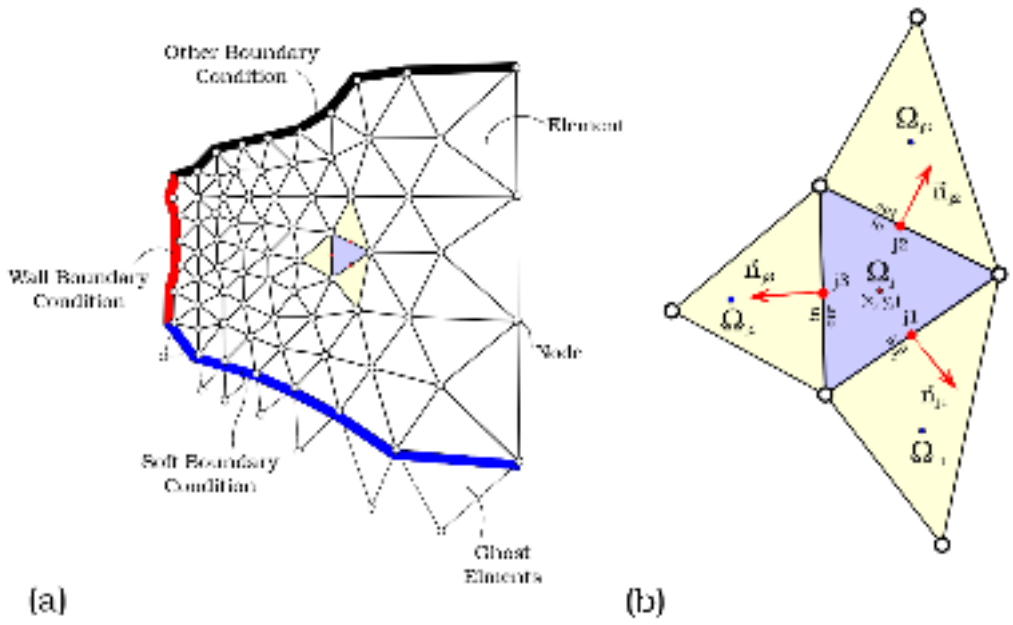


Fig. S3: Illustration of a triangular unstructured grid. The figure shows on the left an example of a finite volume grid, while on the right a typical triangular cell with some attributes used in the semi-discrete formulation

Integrating (S12) over each cell Ω_j and applying the Gauss divergence theorem yields:

$$\frac{d}{dt}\int_{\Omega_j} \boldsymbol{q} d\Omega + \oint_{\partial\Omega_j} (\boldsymbol{F},\boldsymbol{G})\hat{n}_j dl_j = \int_{\Omega} (\boldsymbol{S}_B + \boldsymbol{S}) d\Omega. \tag{S18}$$

Let Ω_{jk} $(k = 1, 2, 3)$ denote the neighboring cells of Ω_j, with (x_j, y_j) as the barycenter coordinates, Γ_{jk} the side toward Ω_{jk}, l_{jk} as its length, and $\hat{n}_{jk} = \cos(\theta_{jk})\hat{\imath} + \sin(\theta_{jk})\hat{\jmath}$ as its outward normal vector, where θ_{jk} is its angle.
The semi-discrete formulation becomes:

$$\frac{d}{dt}\overline{\boldsymbol{q}}_j(t) + \frac{1}{|\Omega_j|}\sum_{k\in\mathcal{N}_j} \mathcal{F}_{jk}\, l_{jk} = \overline{\boldsymbol{S}_{\boldsymbol{B}j}} + \overline{\boldsymbol{S}}_j, \tag{S19}$$

where $\overline{\boldsymbol{q}}_j = \frac{1}{|\Omega j|}\int_{\Omega_j} \boldsymbol{q}(x, y, t)\, d\Omega$ is the average state over the cell, and $\mathcal{F}_{jk}$ is the numerical flux along segment Γ_{jk}, capturing interface interactions. This flux is derived from fields $\boldsymbol{F}$ and $\boldsymbol{G}$ in equation (S13). The scheme relies on approximations of $\boldsymbol{q}$ (and bathymetry B) at Γ_{jk}, denoted $\boldsymbol{q}_{jk}^{in}$ and $\boldsymbol{q}_{jk}^{out}$, and obtained using reconstruction operators. Integrations employ Gaussian quadrature, with the number of Gauss points determined by the degree of this reconstruction for the flow variables. As nodal values are time-dependent, equation (S19) yields a system of ordinary differential equations, setting the stage for the central-upwind discretization detailed in Section 3. Terms $\overline{\boldsymbol{S}_{Bj}}$ and $\overline{\boldsymbol{S}_j}$ are each careful discretizations of the source terms, to be discussed in detail moving forward.

S3.2 The Central-Upwind Scheme

Building on the semi-discrete formulation, `SWEpy` implements the central-upwind (CU) finite volume scheme for hyperbolic conservation laws on triangular grids, as originally proposed by [25] and in parallel by [9] who studied the well-balancing condition, and [46] in the same direction, but with a modified flux formulation; and refined in later works. This Riemann-solver-free method offers a balance between computational simplicity and robustness, estimating local propagation speeds to stabilize fluxes without solving the full Riemann problem, making it well-suited for shallow water flows over unstructured grids with variable topography and wet/dry interfaces.
The numerical flux $\mathcal{F}_{jk}$ in (S19) is formulated as the projection onto the edge-normal direction Γ_{jk}:

$$\mathcal{F}_{jk} = \Big(F_{jk}\cos(\theta_{jk}) + G_{jk}\sin(\theta_{jk})\Big) - \frac{a_{jk}^{\text{in}} a_{jk}^{out}}{a_{jk}^{\text{in}} + a_{jk}^{\text{out}}}\sum_{s=1}^{N_s} c_s\big[\boldsymbol{q}_{jk}^{\text{in}}(M_{jk}^s) - \boldsymbol{q}_{jk}^{\text{out}}(M_{jk}^s)\big], \tag{S20}$$

where M_{jk}^s are the scaled Gaussian quadrature points along the edge, and c_s are the associated weights. The number of points N_s depends on the reconstruction order, ensuring accurate integration of higher-degree polynomials. The terms a_{jk}^{in} and a_{jk}^{out} represent the inward and outward local propagation speeds, detailed below.
The projection components are:

$$F_{jk} = \frac{1}{a_{jk}^{\text{in}} + a_{jk}^{\text{out}}}\sum_{s=1}^{N_s} c_s\left[a_{jk}^{\text{in}}\boldsymbol{F}\left(\boldsymbol{q}_{jk}^{\text{in}}(M_{jk}^s), B(M_{jk}^s)\right) + a_{jk}^{\text{out}}\boldsymbol{F}\left(\boldsymbol{q}_{jk}^{\text{out}}(M_{jk}^s), B(M_{jk}^s)\right)\right], \tag{S21}$$

$$G_{jk} = \frac{1}{a_{jk}^{\text{in}} + a_{jk}^{\text{out}}}\sum_{s=1}^{N_s} c_s\left[a_{jk}^{\text{in}}\boldsymbol{G}\left(\boldsymbol{q}_{jk}^{\text{in}}(M_{jk}^s), B(M_{jk}^s)\right) + a_{jk}^{\text{out}}\boldsymbol{G}\left(\boldsymbol{q}_{jk}^{\text{out}}(M_{jk}^s), B(M_{jk}^s)\right)\right]. \tag{S22}$$

Here, $\boldsymbol{q}_{jk}^{\text{out}}$ represents the limit $\boldsymbol{q}(x, y) \to \boldsymbol{q}_{jk}^{\text{out}}(M_{jk}^s)$ as $(x, y) \in \Omega_j$, while $\boldsymbol{q}_{jk}^{\text{in}}$ is the limit $\boldsymbol{q}(x, y) \to \boldsymbol{q}_{jk}^{\text{in}}(M_{jk}^s)$ as $(x, y) \in \Omega_{jk}$. Furthermore, since divisions by zero may appear near wet/dry fronts given the form of F_{jk} and G_{jk}, an adequate treatment of the fluxes is required to avoid these singularities as addressed in the positivity-preserving reconstruction (Section 3.4). Then, substituting (S21) and (S22) into (S20) and integrating into the semi-discrete scheme (S19) results in:

$$\begin{aligned}\frac{d\overline{\boldsymbol{q}}_j}{dt} =& -\frac{1}{|\Omega_j|}\sum_{k=1}^{3}\frac{l_{jk}\cos\theta_{jk}}{a_{jk}^{\text{in}} + a_{jk}^{\text{out}}}\sum_{s=1}^{N_s} c_s\left[a_{jk}^{\text{in}}\boldsymbol{F}\left(\boldsymbol{q}_{jk}(M_{jk}^s), B(M_{jk}^s)\right) + a_{jk}^{\text{out}}\boldsymbol{F}\left(\boldsymbol{q}_j(M_{jk}^s), B(M_{jk}^s)\right)\right]\\ &-\frac{1}{|\Omega_j|}\sum_{k=1}^{3}\frac{l_{jk}\sin\theta_{jk}}{a_{jk}^{\text{in}} + a_{jk}^{\text{out}}}\sum_{s=1}^{N_s} c_s\left[a_{jk}^{\text{in}}\boldsymbol{G}\left(\boldsymbol{q}_{jk}(M_{jk}^s), B(M_{jk}^s)\right) + a_{jk}^{\text{out}}\boldsymbol{G}\left(\boldsymbol{q}_j(M_{jk}^s), B(M_{jk}^s)\right)\right]\\ &+\frac{1}{|\Omega_j|}\sum_{k=1}^{3} l_{jk}\frac{a_{jk}^{\text{in}} a_{jk}^{\text{out}}}{a_{jk}^{\text{in}} + a_{jk}^{\text{out}}}\sum_{s=1}^{N_s} c_s\left[\boldsymbol{q}_{jk}(M_{jk}^s) - \boldsymbol{q}_j(M_{jk}^s)\right] + \overline{\boldsymbol{S}_{\boldsymbol{B}j}} + \overline{\boldsymbol{S}}_j,\end{aligned} \tag{S23}$$

where $\overline{\boldsymbol{q}}_j$ is the cell-averaged state, B follows the same reconstruction criterion as the flux variables, and $\overline{\boldsymbol{S}}_j$ is the discretized source term, discussed in the following subsection.
To define the one-sided local speeds a_{jk}^{in} and a_{jk}^{out}, which represent the maximum wave speeds at which information propagates inward or outward across the jk interface, we first compute desingularized velocities at the Gaussian points to avoid singularities near dry states:

$$u = \frac{\sqrt{2}h(hu)}{\sqrt{h^4 + \max(h^4, \varepsilon)}}, \quad v = \frac{\sqrt{2}h(hv)}{\sqrt{h^4 + \max(h^4, \varepsilon)}}, \tag{S24}$$

where $h = w - B$ is the water depth, and ε is a small tolerance parameter to prevent singularities. These equations represent the structure to be used with the respective reconstruction operators of the variables evaluated at the necessary points. These velocities are then projected onto the edge normal:

$$u_j^{\theta}(M_{jk}^s) = \cos(\theta_{jk})u_j(M_{jk}^s) + \sin(\theta_{jk})v_j(M_{jk}^s)\ ;\ u_{jk}^{\theta}(M_{jk}^s) = \cos(\theta_{jk})u_{jk}(M_{jk}^s) + \sin(\theta_{jk})v_{jk}(M_{jk}^s)$$

using reconstructions from cells j and its neighbor jk. The local speeds are then determined as the extrema over the Gaussian points:

$$\begin{aligned} a_{jk}^{\text{out}} &= \max_s \left\{ \max \left\{ u_j^\theta(M_{jk}^s) + \sqrt{g h_j(M_{jk}^s)},\ u_{jk}^\theta(M_{jk}^s) + \sqrt{g h_{jk}(M_{jk}^s)},\ 0 \right\} \right\}, \\ a_{jk}^{\text{in}} &= -\min_s \left\{ \min \left\{ u_j^\theta(M_{jk}^s) - \sqrt{g h_j(M_{jk}^s)},\ u_{jk}^\theta(M_{jk}^s) - \sqrt{g h_{jk}(M_{jk}^s)},\ 0 \right\} \right\}. \end{aligned} \tag{S25}$$

We remark that points M_{jk}^s are a number of Gaussian points that depend on the degree of the reconstruction used. In our case, $N_s = 1$ for the linear reconstruction and $N_s = 2$ for the quadratic case.

S3.3 Well-balancing of the source terms

A key requirement for robust SWE solvers, particularly in applications involving complex topography like dam-breaks and tsunamis, is well-balancing: the exact preservation of steady-state solutions without introducing artificial oscillations. This property is essential to maintain physical accuracy in lake-at-rest scenarios or geostrophic equilibria, where source terms must counterbalance flux gradients [26, 7, 29, 11, 12, 16, 21, 28, 10]. In `SWEpy`, we achieve well-balancing through careful discretization of the source terms, ensuring numerical fluxes align with physical conditions across both fully wet domains and variable bathymetry.

S3.3.1 Bathymetry gradient contribution

For the bathymetry source term, we derive a balanced discretization by assuming lake-at-rest conditions and equating it to the momentum flux contributions, as described in [7]. Using the polynomial reconstructions of the variables, we integrate over the cell interior and its edges via Gaussian quadrature, yielding a general form applicable to arbitrary reconstruction orders:

$$\begin{aligned} \overline{\boldsymbol{S}_{\boldsymbol{B}}}_j^{(l)} &= -g \sum_{s=1}^{N_s^{int}} c_s \frac{\partial w(M_j^s)}{\partial x} \left(w_j(M_j^s) - B(M_j^s) \right) + \frac{g}{2|\Omega_j|} \sum_{k=1}^{3} \sum_{s=1}^{N_s} c_s l_{jk} (w_j(M_{jk}^s) - B(M_{jk}^s))^2 \cos\theta_{jk}, \\ \overline{\boldsymbol{S}_{\boldsymbol{B}}}_j^{(3)} &= -g \sum_{s=1}^{N_s^{int}} c_s \frac{\partial w(M_j^s)}{\partial y} \left(w_j(M_j^s) - B(M_j^s) \right) + \frac{g}{2|\Omega_j|} \sum_{k=1}^{3} \sum_{s=1}^{N_s} c_s l_{jk} (w_j(M_{jk}^s) - B(M_{jk}^s))^2 \sin\theta_{jk}, \end{aligned} \tag{S26}$$

where N_s^{int} is the number of Gaussian points M_j^s for quadrature over the cell interior Ω_j, B is the reconstructed bottom via a same-order operator as the one used for the variables, and c_s are the corresponding weights. This formulation ensures the source term discretization mirrors the flux contributions, preventing spurious flows over uneven topography and maintaining equilibrium states critical for realistic simulations, as demonstrated in our steady-state benchmarks (Section 5).

S3.3.2 Manning friction

The Manning friction source term's structure makes it rather straightforward to discretize in a well-balanced manner, since it is proportional to the discharge components $\overline{hu}$ and $\overline{hv}$; thus, the term vanishes identically, preserving equilibrium without further modifications. However, near wet/dry fronts or in low-depth regions where $h \to 0$, desingularization is required to avoid division by zero and ensure numerical stability. Following kurganovfriction, we introduce the discrete friction coefficient:

$$\mathcal{G}(\overline{\boldsymbol{q}}_j) := -g n^2 \left(\frac{2\overline{h}_j}{\overline{h}_j^2 + \max\left(\overline{h}_j^2, \varepsilon^2\right)} \right)^{7/3} \sqrt{(\overline{hu})_j^2 + (\overline{hv})_j^2}, \tag{S27}$$

where $\overline{h}_j = \overline{w}_j - \overline{B_j}$, and ε as indicated before, yielding the discretized friction term as:

$$\overline{\boldsymbol{S}_{\boldsymbol{F}}}_j = \mathcal{G}(\overline{\boldsymbol{q}}_j) \begin{bmatrix} 0 & (\overline{hu})_j & (\overline{hv})_j \end{bmatrix}^T. \tag{S28}$$

This formulation is incorporated semi-implicitly in time integration (Section 3.5) to handle the stiffness of the friction source term kurganovfriction. In `SWEpy`, $\mathcal{G}$ is computed vectorially across all cells on the GPU, enabling efficient parallel evaluation even for large grids.
Although it may vary across the domain, in our experiments the n coefficient is set as a constant for the whole grid (cf. sect. S7.5.1)

S3.3.3 Coriolis

The Coriolis source term vanishes identically under zero-discharge equilibria where $hu = hv = 0$, ensuring inherent well-balancing in friction-dominated regimes, requiring no additional discretization strategies beyond direct averaging:

$$\overline{(\boldsymbol{S}_{\boldsymbol{C}})}_j = f \begin{bmatrix} 0 & (\overline{hv})_j & -(\overline{hu})_j \end{bmatrix}^\top, \tag{S29}$$

where f is the Coriolis parameter, typically approximated as $10^{-4}, s^{-1}$ in mid-latitude regions [24], as used in our Maule 2010 tsunami validation (Section 5.2.2). However, for geophysical flows such as large-scale oceanic or atmospheric circulations–relevant to the far-field tsunami propagation modeled in `SWEpy`–a more subtle form of well-balancing, known as geostrophic balance, is often required. This non-static steady state equilibrates horizontal pressure gradients with Coriolis forces. Recent schemes have addressed this for rotating SWEs through distinct approaches [16, 12]. However, `SWEpy`'s central-upwind framework employs standard balancing, with potential for extensions to enhance geostrophic fidelity in future developments.

S3.4 Spatial Reconstruction Operators and Scheme Formulation

This section presents the methodologies for spatial reconstruction of flow variables within `SWEpy`'s central-upwind finite volume scheme, tailored to address the physical and numerical demands of SWES on unstructured triangular grids. The reconstruction approach is influenced by problem-specific features—such as variable bathymetry, roughness, and domain extent—which dictate the need for accurate approximations to capture gradients and discontinuities, particularly in near-field shocks and far-field wave propagation validated in Section **??**. These approximations must satisfy critical properties: well-balancing, ensuring exact preservation of steady states (e.g., lake-at-rest or geostrophic equilibria over complex topography) to avoid spurious oscillations [7], and positivity preservation, maintaining non-negative water depths at wet/dry fronts to ensure physical realism in inundation scenarios. Numerical experiments with long-range tsunami waves (Section **??**) revealed that constant and linear reconstructions introduce excessive diffusion, compromising wave height accuracy, thus motivating the development of higher-order operators. The reconstruction operators are defined as piecewise polynomials over each cell Ω_j, expressed as:

$$\tilde{q}_j(x,y) = \overline{q}_j + p_j^q(x,y), \tag{S30}$$

where $\overline{q}_j$ is the cell-averaged variable to be reconstructed, p_j^q the interpolating polynomial with coefficients derived from local geometry and neighboring variable cell-averaged values. This cell-wise approach allows tailored approximations, with stencil selection critical for accuracy and stability.

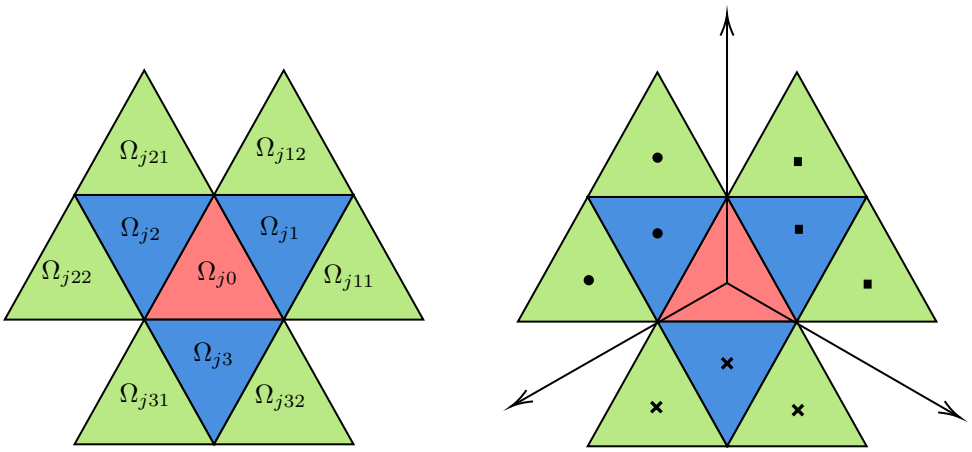


Fig. S4: Stencil illustration for the j-th cell (left) and its sectorial division (right). The blue triangles represent first-order neighbors Ω_{jk}, while the green triangles denote second-order neighbors Ω_{jkl}.

Figure S4 illustrates the stencil structure, where Ω_{j0} (red) is the reference cell, surrounded by first-order neighbors (blue) and second-order neighbors (green). For each Ω_{j0}, the stencils are defined as

$$\{\Omega_{j_0},\Omega_{j_1},\Omega_{j_2},\Omega_{j_3}\};\{\Omega_{j_0},\Omega_{j_1},\Omega_{j_{11}},\Omega_{j_{12}}\};\{\Omega_{j_0},\Omega_{j_2},\Omega_{j_{21}},\Omega_{j_{22}}\};\{\Omega_{j_0},\Omega_{j_3},\Omega_{j_{31}},\Omega_{j_{32}}\}.$$

Linear reconstructions utilize the first group Ω_{ji}, while quadratic reconstructions (employing two Gaussian points per edge) incorporate all cells of the big stencil Ω_{jkl}. The right panel of Figure S4 depicts the selection process for these sub-stencils, illustrating how barycenters of the chosen cells are constrained to lie within cones formed by lines connecting the reference cell's barycenter to its vertices.

S3.4.1 Linear Piecewise Reconstruction with Minmod Gradient Limiter

In this context, the general form of the interpolator (S30) is given by:

$$\tilde{q}_j(x,y) = \overline{q}_j + (q_x)_j(x-x_j) + (q_y)_j(y-y_j), \tag{S31}$$

with $Dq_j = ((q_x)_j,(qy)_j)$ denoting the numerical gradient. The selection criterion for this gradient determines the reconstruction and responds to simulation needs. The variety of selection methods is extensive, as seen in classical approaches [32, 39, 45], finite volume treatments [1, 15, 23, 27], and central-upwind schemes for Saint-Venant systems [7, 25]. In SWEpy's implementation, we follow [7] by constructing three conservative interpolating polynomials $L^j_{k,l}(x,y)$ over Ω_j and pairs $\Omega_{j,k}$, $\Omega_{j,l}$ (see Figure S4). With $\theta \in [1,2]$, define ${q'}_j = \theta\text{minmod}\{\nabla L^j_{kl}\}$. If substituting ${q'}_j$ in (S31) causes midpoints to exceed local extrema, a constant plane through the cell's mean value is imposed; otherwise, $Dq_j = {q'}_j$. This ensures monotonicity and supports well-balancing by aligning with source term discretizations, though it may introduce diffusion in smooth regions, as observed in Section **??**. Details of the procedure are synthesized in Algorithm 1 in the appendices.

The minmod operator, adapted from [7], constructs a piecewise linear interpolant using the cell and two neighbors, minimizing the magnitude of the gradient unless the midpoints exceed the local extrema, in which case a constant value is imposed. The formulation is given as:

$$\tilde{u}_j = \overline{u}_j + \phi_x^{lin}(x-x_j) + \phi_y^{lin}(y-y_j), \tag{S32}$$

where $(\phi_x^{lim},\phi_y^{lim})$ are the regularization parameters (limiters) computed according to the employed method. This ensures monotonicity and supports well-balancing by aligning with source term discretizations, though it may introduce diffusion in smooth regions, as observed in Section **??**.

The minmod linear reconstruction is detailed in Algorithm 1.

Algorithm 1 Minmod linear reconstruction

identify neighbors of each cell (cf. group Ω_{jk} in Fig. S4)
construct planes passing through the mean value of the variable at the barycenters of the cell and its neighbors.
calculate $((q_x)_j, (q_y)_j)$
select the plane whose gradient's magnitude is lowest among its three constructed planes
for `each cell` **do**
  for `each side` **do**
    if not `mean value at cell<reconstructed value at midpoint<mean value at side's neighbor` **then**
      impose a constant plane passing through mean value of the variable over the cell
    end if
  end for
end for

This ensures monotonicity by selecting the least oscillatory interpolant. The algorithm is implemented in parallel for all cells in the `SWEpy` code by operating arrays containing the mentioned quantities.

S3.4.2 Quadratic (WENO)

To mitigate numerical diffusion observed in lower-order reconstructions during long-range wave propagation (e.g., tsunami simulations in Section **??**), we implement a quadratic weighted essentially non-oscillatory (WENO) reconstruction operator, adapted from [47] and applied to unstructured triangular grids like [38], while adhering to the original spatial constraints for stability.
The reconstruction combines a least-squares quadratic polynomial $p_{q,j}$ over the full stencil with four linear polynomials $p_{k,j}$ $(k = 1, \ldots, 4)$ over sub-stencils, expressed as:

$$\tilde{q}_j(x,y) = \frac{w_0}{\gamma_0}\left(p_0(x,y) - \sum_{k=1}^{4} \gamma_k p_{k,j}(x,y)\right) + \sum_{k=1}^{4} w_k p_{k,j}(x,y), \tag{S33}$$

where p_0 is the quadratic polynomial and $p_{k,j}$ are the linear ones, with nonlinear weights w_0, w_k $(k = 1, \ldots, 4)$ computed from smoothness indicators β_0 (quadratic) and β_k (linear) as $w_l = \overline{w}_l/(w_0 + \sum_{k=1}^{4} \overline{w}_k)$, where $\overline{w}_l = \gamma_l(1 + \tau/(\epsilon + \beta_l))$ and τ is a corrector parameter derived from the β_l values newweno.
The quadratic interpolant $p_{q,j}$ is obtained via least-squares fitting to the cell-averaged states over Ω_j and its first- and second-order neighbors, ensuring exact reproduction of the mean in Ω_j. Details of this construction, leveraging only geometric information (e.g., barycenters and area moments) without numerical quadrature for efficiency, are provided in [17]–representing a key contribution to streamlined WENO implementations on CU schemes over unstructured grids.
For stencil selection, grids with sufficient structure enable a fast, loopless index-based search; However, an efficient geometric search algorithm is to be implemented to relax the requirements on the grid further. The procedure is summarized in Algorithm 2, highlighting `SWEpy`'s efficient, GPU-parallelizable design. This WENO adaptation guarantees high-order accuracy in smooth regions while minimizing errors near abrupt gradients, a crucial enhancement for `SWEpy`'s far-field applications where diffusion must be controlled without Riemann solvers kurganov$_c entral-upwind_2$005.
The quadratic WENO reconstruction is summarized in Algorithm 2, which outlines the steps for computing the operator on unstructured triangular grids.

Algorithm 2 Quadratic WENO reconstruction

identify neighbors of each cell (cf. group Ω_{jk} in Fig. S4)
identify neighbors of neighbors of each cell (cf. group Ω_{jkl} in Fig. S4)
for `each cell` **do**
  Construct quadratic polynomial interpolator u_q (cf. Eq.S34)
  Construct linear interpolators $p_{1,j}, p_{2,j}, p_{3,j}, p_{4,j}$
  Set linear weights $\gamma_0, \gamma_1, \gamma_2, \gamma_3, \gamma_4$
  Calculate smoothness indicators $\beta_0, \beta_1, \beta_2, \beta_3, \beta_4$ (Eq. (2.8) in [47])
  Calculate corrector τ for WENO-Z procedure (Eq. (2.13) en [47])
  Calculate and normalize nonlinear weights w_0, w_1, w_2, w_3, w_4
  Construct reconstruction operator over cell as $\tilde{q}_j = \frac{w_0}{\gamma_0}\left(p_0 - \sum_{k=1}^{4} \gamma_k p_{k,j}\right) + \sum_{k=1}^{4} w_k p_{k,j}$
end for

The detailed calculation procedure for the quadratic polynomial $p_{j_0}(x, y)$ relies solely on geometric information associated with the control cell Ω_j. Consider the quadratic form:

$$p_{j_0}(x,y) = a_0 + a_1 x + a_2 y + a_3 x^2 + a_4 y^2 + a_5 xy, \tag{S34}$$

shown in S33, the associated integrals in the construction of the quadratic polynomials and smoothness indicators are calculated exactly.
This polynomial approximates the mean values over the stencil cells and exactly reproduces the mean in Ω_{j_0}. The associated integrals for smoothness indicators are computed exactly. Since the system is overdetermined (9 equations for 5 coefficients), it is solved via least-squares: given $M\boldsymbol{a} = \Delta\boldsymbol{q}$,

$$\boldsymbol{a} \overset{lsq}{=} (M^t M)^{-1} M^t \Delta\boldsymbol{q}. \tag{S35}$$

This approach ensures high-order accuracy while preserving well-balancing in the central-upwind scheme. A further detailed explanation of the reconstructor and its usage in the CU scheme will be in [17].

S3.5 Wet/dry fronts reconstruction

High-order reconstructions, while effective for reducing diffusion in smooth regions, can produce unphysical negative water depths near wet/dry interfaces, where the water surface intersects the bathymetry. To preserve positivity—ensuring non-negative depths for numerical stability and physical realism—we conservatively modify the reconstruction following [7]. This procedure replaces affected reconstructions with a linear polynomial that maintains the cell-averaged value, thus conserving mass, and handles cases with one or two dry vertices differently to align the surface with the bathymetry at those points.
For a cell where reconstructed depths at vertices yield negatives, the surface is redefined as a plane passing through points that enforce positivity. In the two-dry-vertex case, the plane connects the dry vertices at bathymetric levels and the barycenter at the mean surface elevation. For one dry vertex, it connects the dry vertex at bathymetry, a wet vertex at an adjusted elevation, and the barycenter. Mathematically, the plane equation is fitted to these points, ensuring $h = w - B \geq 0$ across the cell while preserving the mass. A schematic is provided in Figure S5, illustrating the geometric adjustment.

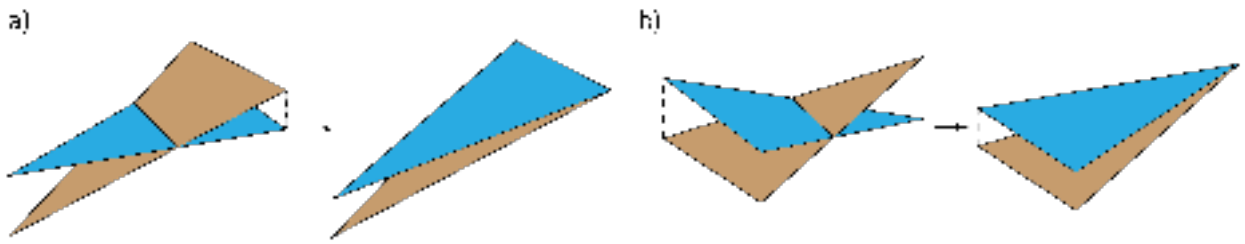


Fig. S5: Schematic representation of the wet/dry treatment: (a) two dry points and (b) one dry point

This method guarantees positivity of the water column ($h \geq 0$) across the domain, essential for avoiding instabilities in inundation problems like in the Conical Island test, or the Malpasset Dam failure case (Section S7.5.1), with the correction algorithm summarized in Algorithm 3 and executed in parallel via GPU vectorization to identify and adjust interface cells efficiently. However, it does not ensure well-balancing near fronts, where slight imbalances may occur [29], suggesting potential extensions with advanced reconstructions for future work.
The positivity-preserving correction for wet/dry fronts is outlined in Algorithm 3.

Algorithm 3 Positivity preserving wet/dry reconstruction

find cells at the wet/dry front
for `each cell in front` **do**
  determines if the cell has one or two dry vertices
  if `cell has two dry vertices` **then**
    $v_1 := (x_{jk1}, y_{jk1}) \leftarrow$ `dry vertex 1`
    $v_2 := (x_{jk2}, y_{jk1}) \leftarrow$ `dry vertex 2`
    $v_3 := (x_j, y_j) \leftarrow$ `cell barycenter`
    $B_{jk1}, B_{jk2}, W \leftarrow$ `bathymetry at` v_1`, bat. at` v_2`, cell mean water level`
    replace reconstruction with plane passing through (v_1, B_{jk1}), (v_2, B_{jk2}), and (v_3, W)
  else if `cell has one dry vertex` **then**
    $v_1 := (x_{jk1}, y_{jk1}) \leftarrow$ `dry vertex`
    $v_2 := (x_{jk2}, y_{jk2}) \leftarrow$ `wet vertex`
    $v_3 := (x_j, y_j) \leftarrow$ `cell barycenter`
    $B_{jk1}, w, W \leftarrow$ `bat. at` v_1`, cell mean w.l., 3/2(w - cell mean bat.)+bat. at` v_2
    replace reconstruction with plane passing through (v_1, B_{jk1}), (v_2, W), and (v_3, w)
  end if
end for

This algorithm corrects reconstructions yielding negative depths by fitting a mass-conserving linear plane, executed in parallel on the GPU for efficiency.

S3.6 Temporal discretization

Following the spatial discretizations detailed in previous subsections, the next challenge is to integrate the resulting system of ordinary differential equations (S19) in time, ensuring stability and accuracy across varying flow regimes. We implement both the Forward-Euler (FE) and a four-stage, third-order strong stability-preserving Runge-Kutta scheme (SSP RK4,3, referred to as RK4,3 throughout our work) [20] for time integration.
For problems involving Manning friction a semi-implicit treatment is incorporated with the objective to enhance stability without compromising efficiency kurganovfriction. We define the flux operator as:

$$\mathcal{H}_j(\overline{\boldsymbol{q}}_j, \tilde{\boldsymbol{q}}_j) = -\frac{1}{|\Omega_j|} \sum_{k \in \mathcal{N}_j} (\mathcal{F}_{jk}) l_{jk} + \overline{\boldsymbol{S}\boldsymbol{B}_j}, \tag{S36}$$

where $\overline{\boldsymbol{q}}_j$ are the conserved variables, $\tilde{\boldsymbol{q}}_j$ denotes the reconstructions. In a Shu-Osher form, the semi-implicit RK update is then:

$$\left(\overline{\boldsymbol{q}}_j\right)^i = \sum_{l=0}^{i-1} \alpha_{i,l} \left(\overline{\boldsymbol{q}}_j\right)^l + \frac{\Delta t}{2} \sum_{l=0}^{i-1} \beta_{i,l} \mathcal{H}_j^l + \Delta t \left(\mathcal{G}(\overline{\boldsymbol{q}}_j)\right)^{i-1} \begin{bmatrix} 0 & (\overline{hu}_j^i) & (\overline{hv}_j^i) \end{bmatrix}^T, \quad i = 1, 2, 3, 4 \tag{S37}$$

with $\overline{\boldsymbol{q}}_j^l$ are the intermediate state values, $\overline{\boldsymbol{q}}_j^0 = \overline{\boldsymbol{q}}_j^{(n)}$ and $\overline{\boldsymbol{q}}_j^{(n+1)} = \overline{\boldsymbol{q}}_j^4$. The nonzero coefficients for SSPRK(4,3) are

$$\begin{array}{cccccccccc} \begin{pmatrix}\alpha_{1,0} & \alpha_{2,1} & \alpha_{3,0} & \alpha_{3,2} & \alpha_{4,3}\end{pmatrix} & = & \begin{pmatrix}1 & 1 & 2/3 & 1/3 & 1\end{pmatrix}, \\ \begin{pmatrix}\beta_{1,0} & \beta_{2,1} & \beta_{3,2} & \beta_{4,3}\end{pmatrix} & = & \begin{pmatrix}1 & 1 & 1/3 & 1\end{pmatrix}. \end{array} \tag{S38}$$

while the FE formulation is identical with nonzero coefficients $\alpha_{1,0} = 1, \beta_{1,0} = 1$.
This semi-implicit approach tackles numerical stability challenges from stiff, strongly coupled, or problematic source terms, particularly in friction-dominated flows near wet/dry fronts, as validated in our dam-break cases.
Finally, as a quantifier of the control process between the model evolution, the grid, and the information transport speed, the Courant–Friedrichs–Lewy condition can be forced so the time step Δt is adaptively computed as:

$$\Delta t = CFL \cdot \min_{jk} \frac{r_{jk}}{\max\{a_{jk}^{\text{in}}, a_{jk}^{\text{out}}\}}. \tag{S39}$$

where r_{jk} is the perpendicular height from edge jk to the opposite vertex, and CFL is user-defined. Per [7], CFL $\leq 1/3$ is recommended, though our scheme supports larger values empirically, which ensures the positiy preserving condition $h = w - B \geq 0$. For SSP RK4,3, Δt is calculated in the first stage and scaled for subsequent ones, balancing stability and efficiency.

S3.7 Boundary conditions

The numerical treatment of domain boundaries is handled through a dual approach: explicit *Ghost Cells* for physical boundaries and *Topological Connectivity* for periodic conditions.
For physical boundaries, the domain Ω is extended by a layer of virtual cells. The state vector $\mathbf{U}_{ghost}$ in these cells is reconstructed at the beginning of each time step to enforce specific flow constraints at the interface Γ. Currently, `SWEpy` supports four fundamental boundary types:

S3.7.1 Prescribed Water Head

This Dirichlet-type boundary condition is implemented when the water level at the boundary is dictated by external forcing, such as tidal signals or reservoir stages. Consistent with the formulation described in Chapter 3, the numerical flux computation requires the full vector of conserved variables $\mathbf{U} = [w, hu, hv]^T$ to be defined at both sides of the interface.
To ensure a well-posed Riemann problem, the ghost cell state vector $\mathbf{U}_{ghost}$ is assembled by coupling the prescribed target elevation with the velocity field extrapolated from the inner domain. Thus, by constructing the vector $\mathbf{U}_{ghost}$ in this manner, the solver can correctly compute the numerical fluxes $\mathcal{F}(\mathbf{U}_{ghost}, \mathbf{U}_{inner})$, allowing mass to enter or exit the domain driven by the hydrostatic pressure gradient while minimizing artificial wave reflections.

S3.7.2 Reflective (Wall) Boundary

This boundary condition is used to model solid, impermeable obstacles that close the computational domain, such as channel walls, dams, or coastline boundaries. Physically, it imposes a slip condition where the velocity component normal to the boundary must vanish ($\mathbf{u} \cdot \mathbf{n} = 0$), reflecting any wave information propagating towards it.

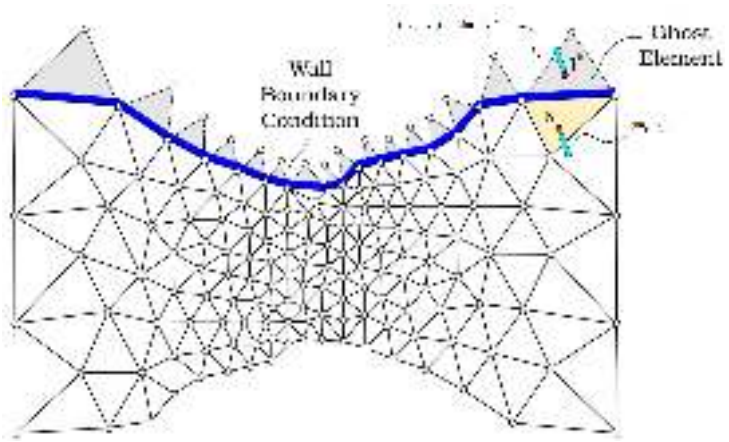


Fig. S6: Schematic representation of the Wall boundary condition using ghost cells.

To enforce this zero-flux condition numerically, the ghost cell state vector $\mathbf{U}_{ghost}$ is constructed to mirror the hydrostatics of the interior cell while inverting the momentum. Following the notation established, the reconstruction is defined as:

$$\mathbf{U}_{ghost} = \begin{bmatrix} w_{ghost} \\ \mathbf{q}_{ghost} \end{bmatrix} = \begin{bmatrix} w_{inner} \\ -\mathbf{q}_{inner} \end{bmatrix} \tag{S40}$$

By setting $\mathbf{q}_{ghost} = -\mathbf{q}_{inner}$, the Riemann solver at the interface computes a net flux of zero across the wall edge, effectively simulating a perfect reflection.

S3.7.3 Transmissive (Soft) Boundary

This boundary condition, also referred to as an open or radiation boundary, is designed to simulate an infinite domain by allowing waves and flow information to exit the computational mesh with minimal non-physical reflection. It is conceptually depicted in Figure S7.

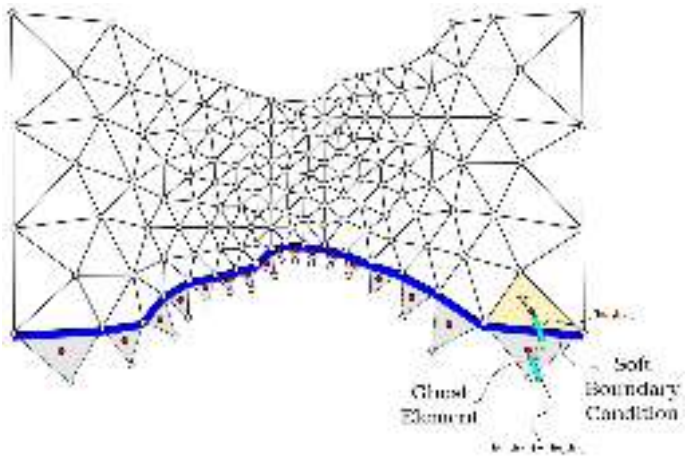


Fig. S7: Schematic of the Soft or Transmissive boundary condition.

In `SWEpy`, this condition is implemented via a zero-order extrapolation (Neumann boundary condition with zero gradient). The state vector of the ghost element $\mathbf{U}_{ghost}$ is set equal to the state of the adjacent interior element $\mathbf{U}_{inner}$. Following our standard notation:

$$\mathbf{U}_{ghost} = \begin{bmatrix} w_{ghost} \\ \mathbf{q}_{ghost} \end{bmatrix} = \begin{bmatrix} w_{inner} \\ \mathbf{q}_{inner} \end{bmatrix} = \mathbf{U}_{inner} \tag{S41}$$

By equating the ghost state to the interior state, the numerical flux function at the interface $\mathcal{F}(\mathbf{U}_{ghost}, \mathbf{U}_{inner})$ effectively transports the interior information out of the domain. Note that if the interior cell has a non-zero velocity directed outwards, this condition will replicate it, generating a continuous outflow. Conversely, if the velocity is directed inwards, it will replicate an inflow.

S3.7.4 Periodic Boundary

Unlike the previous conditions which rely on ghost cell reconstruction, the periodic boundary is implemented *topologically*. It is applicable to rectangular or regular domains where the discretization of opposing boundaries is symmetric (node-conforming).
Instead of creating virtual elements, the mesh connectivity graph is modified such that an element on the left boundary (Γ_{left}) identifies a specific element on the right boundary (Γ_{right}) as its direct physical neighbor, and vice versa.

Fig. S8: Conceptual representation of the Periodic boundary condition. The domain effectively becomes continuous (toroidal), eliminating the need for ghost cells.

This topological "stitching" effectively transforms the domain into a continuous loop. Consequently, the numerical scheme treats boundary elements exactly as interior elements, avoiding the need for special reconstruction or flux inversion.

S3.7.5 Numerical Considerations for Truncated Bathymetry

Special attention is required when imposing boundary conditions on domains with complex topography, particularly where the wet/dry front interacts directly with the domain edge (truncated bathymetry). This scenario is common in dam-break benchmarks such as the Malpasset case (see Section S7.5.1).
This reconstruction technique ensures stability by balancing the flux gradients with the source terms. However, at a physical boundary, the ghost cell must not only reconstruct the state vector $\mathbf{U}$, but also assume a bathymetry profile B_{ghost}.

If a boundary cell is initialized as dry ($h \approx 0$) but features a steep bathymetric slope, a naive reconstruction of the ghost cell elevation w_{ghost} may place the water surface below the bed elevation ($w < B$) or artificially create a pressure gradient. As illustrated in Figure S9, this mismatch can generate artificial inflow or localized instabilities.

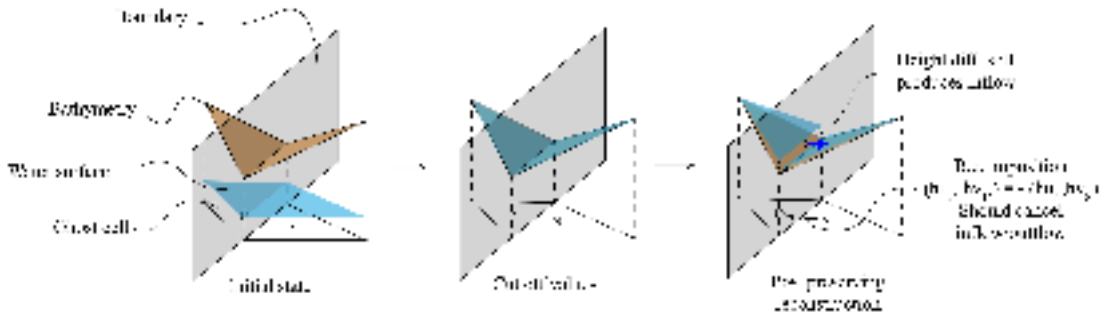


Fig. S9: Schematic of the hydrostatic reconstruction issue at boundaries. If the ghost cell bathymetry does not match the interior slope near a wet/dry front, the reconstruction may induce artificial fluxes.

Best Practices for Mesh Design: To mitigate these numerical artifacts, it is strongly recommended to ensure **topological consistency** at the boundaries. The user should design the mesh such that:

1. The bathymetry $B(x, y)$ is locally flat ($\nabla B \approx 0$) in the layer of elements adjacent to the boundary.
2. Alternatively, the computational domain should be extended slightly outwards so that the complex topography (and the wet/dry interface) lies entirely within the interior domain, leaving the boundaries in a stable, fully wet or fully dry region.

S4 Code Architecture & Implementation

One of the core design philosophies of `SWEpy` is modularity. The codebase is structured to separate the numerical core, time integration schemes, and problem configuration, allowing users to extend the solver's capabilities without altering the underlying engine.

S4.1 The Mesh Dictionary: Unified Data Structure

The central data structure in `SWEpy` is the `mesh` dictionary. It acts as a container for all simulation data, including grid geometry, flow variables, and configuration parameters. This dictionary is passed between all modules, ensuring efficient access to data on the GPU.

Key fields in the `mesh` dictionary include:

- **Geometry**:
  - `xj, yj`: Centroid coordinates of each triangle $[1, N_{elem}]$.
  - `nx, ny`: Outward unit normal components at each side $[3, N_{elem}]$.
  - `ljk`: Length of each triangle side $[3, N_{elem}]$.
  - `Tj`: Area of each triangle $[1, N_{elem}]$.
- **Flow Variables** (Cell-Average):
  - `Wj`: Water surface elevation w $[1, N_{elem}]$.
  - `HUj, HVj`: Momentum components hu, hv $[1, N_{elem}]$.
  - `Bj`: Bathymetry elevation B $[1, N_{elem}]$.
- **Reconstruction Data** (Vertex/Edge):
  - `Bmj`: Bathymetry at edge midpoints $[3, N_{elem}]$.
  - `Coordinates`: Vertex coordinates (x, y) of the mesh $[2, N_{nodes}]$.
  - `Elements`: Connectivity array $[3, N_{elem}]$.
- **Topology**:
  - `Neighboors`: Adjacency list $[3, N_{elem}]$.
  - `jk`: Global indices of neighbor elements, optimized for gathering operations.

Extended Mesh for High-Order Schemes: When using quadratic reconstruction (e.g., WENO), the `mesh` dictionary is augmented with additional precomputed fields during the initialization phase (via 'cuUtilities.precomp_weno2'):

- **Extended Topology**: `Neighs2` stores the second-layer neighbors $[3, 2, N_{elem}]$ required for the larger stencil.
- **Geometric Moments**: `Ix, Iy, Ixy` store the second moments of inertia for each triangle, used in the least-squares reconstruction.
- **Reconstruction Coefficients**:
  - `lsq_coefs`: Coefficients for the quadratic least-squares polynomial.
  - `lin_coefs`: Coefficients for the linear reconstruction stencils.

S4.2 Codebase Overview

The library is organized into distinct Python modules, each handling a specific aspect of the simulation pipeline, located in the `src/` directory:

- `cuShallowWater.py`: The main entry point of the software.
  - `run()`: Orchestrates the main simulation loop, handling memory transfer between host and device, and managing IO operations.
  - `run_with_TS()`: Runs the simulation while recording high-frequency time-series data at specified gauge points.
  - `choose_timestep()`: Selects the appropriate time-integration function and spatial reconstruction method based on user input.
- `cuSolver.py`: Contains the time-integration algorithms and stability controls.
  - `runge_kutta3() / forward_euler()`: Standard integration sequences using linear spatial reconstruction.
  - `constant_runge_kutta3() / constant_forward_euler()`: simplified first-order spatial reconstruction (piecewise constant) combined with RK3 or Forward Euler time stepping.
  - `runge_kutta3_weno() / forward_euler_weno()`: High-order integration sequences using quadratic WENO spatial reconstruction.
  - `max_time_step()`: Computes the adaptive time step Δt based on the Courant-Friedrichs-Lewy (CFL) condition, considering ghost cells.
  - `max_time_step_gl()`: Computes the adaptive time step Δt based on the Courant-Friedrichs-Lewy (CFL) condition, without considering ghost cells.

- `cuCentralUpwindMethod.py`: The numerical kernel implementing the Central-Upwind scheme. It differentiates between standard (constant/linear) and high-order (quadratic) implementations since the latter requires values at 2 gaussian points per triangle side (rather than at the midpoint of each side):
  - **Standard Functions** (used for Constant and Linear schemes):
    - `flux_function_x() / _y()`: Computes numerical fluxes at cell interfaces.
    - `one_sided_speed()`: Calculates propagation speeds (a_{in}, a_{out}).
    - `source_term()`: Evaluates topography source terms.
  - **Higher-Order Functions** (used for Quadratic schemes, suffixed with `2`):
    - `flux_function_x2() / _y2()`: Computes fluxes incorporating higher-order reconstruction data.
    - `one_sided_speed2()`: Calculates propagation speeds for quadratic reconstruction.
    - `source_term2()`: Evaluates bathymetry source terms with higher-order accuracy.
  - **Common Utilities**:
    - `friction_term()`: Implements bottom friction (Manning's formulation).
    - `coriolis()`: Computes Coriolis force contributions.
    - `midvelocity()`: Performs velocity desingularization.
- `cuPieceWiseReconstruction.py`: Handles high-order spatial reconstruction and wet/dry front treatment.
  - **Spatial Reconstruction**:
    - `weno2()`: Implements a quadratic WENO reconstruction for high-order accuracy. Constructs lambda functions representing reconstructions across triangle cells.
    - `minmod()`: Implements a linear minmod reconstruction. Calculates gradients at each cell.
    - `set_constant/linear/weno_midpoint_values()`: Uses the pertinent reconstruction to calculate the midpoint values of the variables.
    - `set_constant/linear/weno_vertices_values()`: Uses the pertinent reconstruction to calculate the vertices values of the variables. Used in plotting and wet/dry reconstruction.
  - **Wet/Dry Reconstruction** (Positivity Preserving):
    - `set_linear_well_balanced_wet_dry()`: Reconstructs water surface levels handling wet/dry interfaces for linear schemes.
    - `set_constant_well_balanced_wet_dry()`: Handles wet/dry interfaces for first-order (constant) schemes.
    - `set_linear_well_balanced_wet_dry2()`: Variant for handling wet/dry interfaces in conjunction with WENO reconstruction.
- `cuBoundaryConditions.py`: Manages boundary enforcement.
  - `impose()`: Applies boundary conditions (e.g., reflective wall, transmissive) by modifying ghost cell values at the end of each time step.
- `cuFileLoader.py`: Handles I/O operations for initialization.
  - `load_from_files()`: The master function that initializes the `mesh` dictionary by calling specific loaders.
  - `load_configuration_file()`: Parses user parameters.
  - `load_mesh_from_file()`: Reads grid topology and coordinates.
- `cuFileSaver.py`: Manages data output for visualization and analysis. Performs explicit array GPU-CPU memory transfers (`cupy.asnumpy`) to avoid syncronization issues caused by sequential (line by line) file writing.
  - `save_animation()`: Writes solution snapshots to VTK Unstructured Grid (`.vtu`) format for Paraview.
  - `save_bathymetry()`: Exports the static bathymetry mesh.
  - `save_TS()`: Records time-series data at specific gauges.

S4.3 Execution Flows

The execution pipeline in `SWEpy` differs slightly depending on the chosen spatial reconstruction method (Standard vs. High-Order).
Standard Execution Flow (Constant/Linear): For standard schemes (e.g., `FE`, `RK3`), the workflow is streamlined:

1. `choose_timestep` selects the appropriate integration function (e.g., `forward_euler`).
2. The simulation loop (`run`) starts immediately.
3. Inside the loop, the solver calls standard reconstruction methods (e.g., `set_linear_vertices_values`) on-the-fly at each time step using only immediate neighbor data.

High-Order Execution Flow (WENO): When a high-order scheme is selected (e.g., `RK3WENO`), an additional pre-processing step is required:

1. `choose_timestep` detects the high-order option.
2. It triggers `cuUtilities.precomp_weno2(mesh)`. This critical step prepares the mesh for quadratic reconstruction by:
   - Computing extended stencils (including second-layer neighbors).
   - Calculating second moments of area (I_x, I_y, I_{xy}) for each cell.
   - Pre-inverting matrices to obtain the coefficients for the linear and least-squares reconstruction polynomials.
3. The simulation loop proceeds, passing this augmented `mesh` to the high-order solver functions (e.g., `runge_kutta3_weno`), which utilize the precomputed data for quadratic reconstruction.

S4.4 GPU Implementation Strategies

A core feature of `SWEpy` is its "GPU-native" design, where array-based operations replace traditional loops to exploit the massive parallelism of CUDA devices. This approach requires rethinking standard sequential algorithms into vectorized forms.

S4.4.1 Vectorized Spatial Reconstruction

Spatial reconstruction operators are fully optimized to maximize GPU throughput. Instead of iterating over cells, the solver operates on global arrays where data for each cell and its neighbors are accessed simultaneously. This Single Instruction, Multiple Data (SIMD) approach ensures thread coherence. For instance, in minmod-type reconstructions, gradient limiting is performed not by branching logic but by computing potential slopes in all directions and selecting the minimum modulus via element-wise operations (e.g., `cupy.minimum` and `cupy.choose`).

S4.4.2 WENO Precomputation and Neighbor Finding

The higher-order WENO scheme involves complex stencil operations that are computationally expensive if evaluated naively. `SWEpy` mitigates this by offloading geometric analysis to a pre-processing stage (function `cuUtilities.precomp_weno2`):

- **Parallel Matrix Inversion**: The coefficients for the quadratic reconstruction are derived from a local least-squares problem $M\mathbf{a} = \Delta\mathbf{q}$. The geometry matrix M depends solely on the mesh. During initialization, the pseudo-inverse $(M^TM)^{-1}M^T$ is computed for all cells simultaneously using `cupy.linalg.inv` on the stacked arrays. The resulting weights are stored, reducing the runtime reconstruction to a fast sequence of matrix-vector multiplications.
- **Mask-Based Neighbor Search**: The extended stencils (neighbors of neighbors) are identified using boolean masks rather than graph traversal loops. The `neighbor2` function constructs a logical mask of the grid connectivity to filter and map second-order neighbors into a fixed-structure lookup table (`Neighs2`), ensuring coalesced memory access during the time-stepping loop.

S4.4.3 Wet/Dry Fronts Algorithm

Handling wet/dry boundaries traditionally involves `if-else` branching that causes thread divergence on GPUs. `SWEpy` implements a sorting-free, mask-based strategy in functions like `set_linear_well_balanced_wet_dry`:

1. **Classification**: A tolerance mask is created to identify cells at the interface. Using `cupy.count_nonzero`, cells are classified in parallel into cases with 1, 2, or 3 dry vertices.
2. **Gather**: Data for identifying cells is extracted into dense "blocks" (e.g., `block1`, `block2`) using advanced indexing. This segregates the workload so that threads processing a block execute identical instructions.
3. **Compute & Scatter**: The geometric correction (fitting a plane to preserve mass while enforcing positivity) is applied efficiently to each block. The corrected values are then scattered back into the global state arrays using the pre-calculated indices, ensuring the global solution remains consistent without serializing the wavefront execution.

S4.5 Solver Structure and Abstraction

Despite the variety of available time-integration schemes (e.g., RK3, Forward Euler, with different spatial reconstructions), `SWEpy` abstracts this complexity through a unified interface.

S4.5.1 The `timestep` Function Pointer

The `cuShallowWater.py` module does not hard-code the integration method in the main loop. Instead, the `choose_timestep` function returns a function pointer (callable) referred to as `timestep`. All solver functions in `cuSolver.py` adhere to a strictly consistent signature:

```
1    timestep(mesh, t, rgc, rkdt, cm)
```

This allows the main simulation loop to remain agnostic to the specific numerical method being used.

S4.5.2 Internal Structure of Solver Functions

Whether using a first-order single-stage Forward Euler or a high-order multi-stage Runge-Kutta scheme, the internal logic within `cuSolver.py` functions follows a standardized pipeline per stage:

1. **Unpack**: Extract essential fields from the `mesh` dictionary.
2. **Reconstruction**: Call the appropriate method from `cuPieceWiseReconstruction` (e.g., `minmod`, `weno2`) to obtain values at cell interfaces.
3. **Correction**: Apply corrections to the reconstructed values to ensure positivity preservation.
4. **Physics Evaluation**: Compute desingularized velocities, wave speeds (a_{in}, a_{out}), source terms, and friction.
5. **Numerical Fluxes**: Calculate fluxes using the methods in `cuCentralUpwindMethod`.
6. **Update**: Advance the conserved variables (w, hu, hv) using the computed fluxes and time step Δt.

S4.5.3 Relation between Schemes

There is a tight structural coupling between the schemes. The Runge-Kutta implementations (e.g., `runge_kutta3`) are often constructed as wrappers or multi-stage sequences that utilize the core update logic found in the Forward Euler implementation. For example, the SSP-RK3 scheme makes three calls to a base Euler-like update step (or explicitly repeats the spatial discretization logic) to achieve third-order temporal accuracy, combining the stages via convex combination to preserve stability.

S4.6 Module Linking and Extensibility

SWEpy uses direct Python imports to link these modules, ensuring that the entire execution flow remains visible, interpretable, and easy to modify for the user. There are no compiled "black box" extensions masking the core logic.

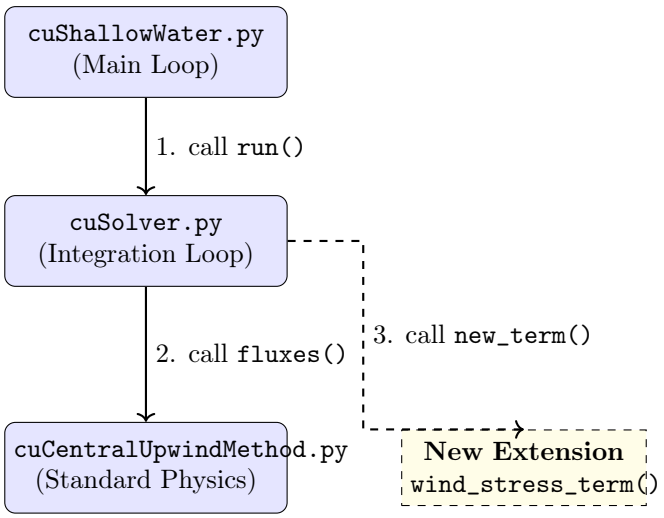


Fig. S10: Module interaction diagram. The solid lines represent the standard execution flow, while the dashed line illustrates how a user-defined extension (e.g., wind stress) is linked into the solver loop.

S4.6.1 Workflow for Adding New Physical Terms

One of the most common user extensions is the addition of new source terms, such as wind stress, rain intensity, or infiltration. The following workflow should be adopted:

1. **Definition**: Define the new mathematical function in `cuCentralUpwindMethod.py`. This function should accept `CuPy` arrays representing the flow variables (e.g., H, U, V) and return the computed term array.
2. **Integration**: Build a new time-stepping function in `cuSolver.py`, possibly replicating an existing one, that incorporates the new term. Or modify one of interest.
3. **Linking**: Call the new function within the integration loop. If the term is a source term S, it should be added to the flux balance equation.

Example: Adding a Wind Stress Term Suppose we want to add a wind stress term W_x, W_y.
Step 1: Define the function in `src/cuCentralUpwindMethod.py`:

```
def wind_stress_term(H, Wparams):
    # Calculate wind stress based on depth H and parameters
    Wx = ...
    Wy = ...
    return Wx, Wy
```

Step 2 & 3: Link in Solver in `src/cuSolver.py`: Inside the chosen time-stepping function (e.g., inside the `forward_euler` sequence), add the call:

```
    # ... existing source term calls ...
    Sx, Sy = CentralUpwindMethod.source_term(...)

    # [NEW] Calculate Wind Stress
    Wx, Wy = CentralUpwindMethod.wind_stress_term(mesh["Hj"], mesh["Constants"]["Wind"])

    # ... water level update ...

    # [MODIFY] Flux update along X-direction (add Wx)
    mesh["HUj"] = mesh["HUj"] + (mesh["dt"]*( ... )/mesh["Tj"]  + mesh["dt"]*Sx + mesh["dt"]*fx + mesh["dt"]*Wx)/rkc

    # [MODIFY] Flux update along Y-direction (add Wy)
    mesh["HVj"] = mesh["HVj"] + (mesh["dt"]*( ... )/mesh["Tj"]  + mesh["dt"]*Sy + mesh["dt"]*fy + mesh["dt"]*Wy)/rkc
```

This structure ensures that changes to the physics (in `cuCentralUpwindMethod`) remain decoupled from the time-marching logic (in `cuSolver`), facilitating rapid experimentation and development.

S4.7 Optimization Strategy: Custom Element-wise Kernels

To further reduce computational overhead, future development considers replacing sequences of array operations with custom-compiled CUDA kernels using `cupy.ElementwiseKernel`. This strategy targets specific bottlenecks such as flux computations, friction terms, and spatial reconstructions.

By defining these operations as C-like kernels, `SWEpy` would achieve two critical performance gains:

1. **Kernel Fusion**: Multiple arithmetic steps (e.g., computing a_{in} and a_{out} wave speeds) are fused into a single kernel launch. This eliminates the latency of launching separate kernels for each intermediate calculation and significantly reduces memory bandwidth usage by keeping temporary values in GPU registers.
2. **Complex Logic Handling**: Operations that traditionally require multi-pass masking in Python (such as the `minmod` limiter or depth checks for friction) can be implemented with efficient C-level branching within the kernel. This avoids allocating large boolean mask arrays and traversing memory multiple times.

For example, the `minmod` reconstruction, originally a sequence of array slicings and logical comparisons, each performed by its own individual kernel, could be optimized into a single kernel that computes slopes, applies the theta-limiter, and enforces monotonicity in one pass. This hybrid approach would retain the Pythonic modularity of the solver structure while delegating the most intensive "hot loops" to highly optimized, compiled machine code.

S5 Simulation Setup & Input Files

Setting up a numerical simulation in `SWEpy` involves more than just providing a geometry; it requires a complete topological description of the domain to leverage the parallel processing power of the GPU. This chapter details the standardized input system, designed to be both human-readable in ASCII format and optimized for fast memory loading into the `CuPy` environment.

S5.1 Directory Layout

To ensure reproducibility and ease of use, `SWEpy` adopts a modular directory structure. This organization strictly separates the static source engine (`src/`) from the user-defined simulation cases (`input_example/`). This layout allows researchers to maintain a single core codebase while managing multiple projects, each with its own configuration and mesh files. The following tree illustrates the standard repository structure required for a successful execution.

```
SWEpy/ ..... Root directory
├── src/ ..... Source code modules
│   ├── cuShallowWater.py ..... Main driver & initialization
│   ├── cuSolver.py ..... Numerical schemes (RW, FE, etc.)
│   ├── cuCentralUpwindMethod.py ..... Flux calculation schemes
│   ├── cuPieceWiseReconstruction.py ..... Data reconstruction (WENO, Linear)
│   ├── cuFileLoader.py ..... Input parsing and mesh loading
│   ├── cuFileSaver.py ..... Output generation (VTK, txt)
│   ├── cuBoundaryConditions.py ..... Boundary condition implementation
│   ├── cuAnalyticSolutions.py ..... Verification benchmarks
│   └── cuUtilities.py ..... Helper functions and GPU kernels
├── input_example/ ..... Template simulation case
│   ├── Config.swe ..... Main configuration file
│   └── *.txt ..... Mesh and topology files
├── run_sim.py ..... Command-line execution script
├── check_installation.py ..... Dependency verification
├── README.md
└── LICENSE
```

S5.2 Configuration and Runtime Parameters

The `Config.swe` file acts as the primary orchestrator of the simulation. It is a plain ASCII text file where the user defines the physical constants, numerical tolerances, time-stepping constraints, and the mapping of all input data files.
One of the key advantages of this format is its simplicity: it uses a key-value pair structure that allows the solver to initialize the `mesh` dictionary without requiring hard-coded paths. This decoupling ensures that the same `SWEpy` engine can run entirely different scenarios simply by swapping this configuration file.

S5.2.1 Parameter Classification

The parameters within `Config.swe` can be grouped into three functional categories: Execution control, physical/numerical constants, and file pointers.

Category	Key	Description
Time Control	`Tmax`	Total simulation duration (s).
	`dt_save`	Temporal interval for saving output files and gauge snapshots (s).
Numerical	`CFL`	Courant-Friedrichs-Lewy stability number ($0 < CFL < 1$).
	`Dry`	Threshold depth (m) to distinguish between wet and dry cells.
	`Tolerance`	Numerical epsilon for floating-point comparisons and convergence.
Physics	`Gravity`	Acceleration due to gravity (g, usually $9.81\,m/s^2$).
	`Roughness`	Global Manning's n coefficient for bed shear stress calculation.
Data Mapping	`WaterLevel`	Path to the initial free surface elevation w at cell centroids.
	`Discharge`	Path to the initial fluxes (hu, hv) at cell centroids.
	`Bathymetry`	Path to the bed elevation B relative to a vertical datum.
	`Coordinates`	Path to the spatial (x, y) vertex coordinates of the mesh.
	`Elements`	Path to the connectivity list (Vertex IDs forming each triangle).
	`Neighboors`	Path to the adjacency list (IDs of adjacent triangular elements).
	`Sides`	Path to the side identification file (mapping local to global edges).
	`GhostCells`	Path to the boundary condition mapping and factor file.

Tab. S1: Detailed description of `config.swe` parameters and their corresponding physical/topological entities.

S5.2.2 Formatting Rules

To ensure the `cuFileLoader` parses the data correctly, the user must adhere to the following rules:

- **Case Sensitivity:** Keys are case-sensitive and must match the solver's expected strings (e.g., `Tmax`, not `tmax`).
- **Spacing:** A single tab or space must separate the key from its assigned value.
- **Comments:** While not explicitly shown in the example, the parser ignores lines starting with `#`.

An example of a standard configuration for a 10-second simulation is shown below:

```
Tmax            10
Dry             1e-06
Tolerance       1e-06
Gravity         9.81
Roughness       0
CFL             0.25
dt_save         1
WaterLevel      WaterLevel.txt
Discharge       Discharge.txt
Bathymetry      Bathymetry.txt
Coordinates     NodeCoords.txt
Sides           ElemNeighSides.txt
Elements        ElemNodes.txt
Neighboors      ElemNeighs.txt
GhostCells      GhostCells.txt
```

Listing 2: Example of a config.swe file content.

S5.3 Mesh and Topology Definition

Unlike structured grid solvers where neighbor indices are implicit, `SWEpy` operates on unstructured triangular meshes. This flexibility allows for better representation of complex coastlines and irregular bathymetry but requires an explicit definition of how nodes and elements are connected. The following files establish the 'skeleton' of the simulation, providing the spatial coordinates and the connectivity graph necessary for calculating numerical fluxes at each interface.

S5.3.1 Geometric Files

The geometry is established by two primary files (*NodeCoords.txt* and *ElemNodes.txt*) that link discrete points in space to the triangular elements that form the computational domain. These files use a standardized 0-indexed system to identify every entity uniquely.

The **NodeCoords.txt** text file is a standard two column format. The first character # means that the line is a comment. The first number – in this case 17587 – represents the number of nodes in the mesh. The second number – in this case 2 – represents the number of coordinates per node.

The other values are the coordinate of the i-th node along x-axis and the y-axis.

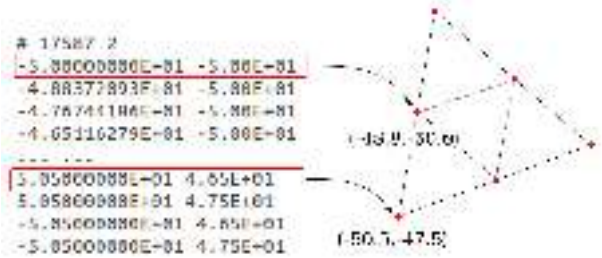


Fig. S11: NodeCoords.txt file format.

The **ElemNodes.txt** text file is a standard three column format. The first character # means that the line is a comment. The first number – in this case 34600 – represents the number of elements in the mesh. The second number – in this case 3 – represents the number of nodes per element.

The other values are the node identifiers that define the i-th element. For instance, the line on the red rectangle represents the i-th element which is formed by the nodes 8764, 17365, and 8765 respectively.

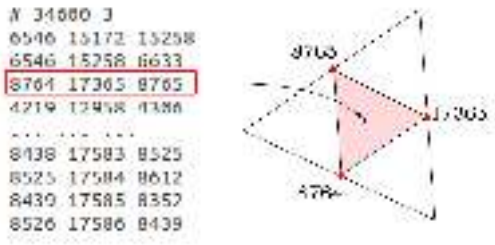


Fig. S12: ElemNodes.txt file format.

S5.3.2 Connectivity and Boundaries

To maximize GPU throughput, `SWEpy` avoids on-the-fly neighbor searching. Instead, it relies on pre-computed adjacency lists (*ElemNeighs.txt* and *GhostCells.txt*). These files tell each triangle which elements are its immediate neighbors and how to handle the physical limits of the domain through the ghost cell method."

The **ElemNeighs.txt** is a standard three column format. The first character # means that the line is a comment. The first number – in this case 34600 – represents the number of neighbor elements. The second number – in this case 3 – represents the number of columns.

The other values are the elements identifiers that surrounds the i-th element. For instance, the line on the red rectangle represents the i-th elements which is surrounded by the elements 220, 1620, and 1 respectively. Now if one of these values is -1, it means that there is no a neighbor element, this case is seen in the second red rectangle.

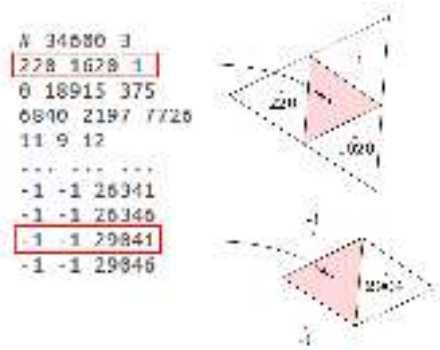


Fig. S13: ElemNeighs.txt file format.

The **GhostCells.txt** text file is a standard three column format. The first character # means that the line is a comment. The first number – in this case 200 – represents the number of ghost elements in the mesh. The second number – in this case 3 – represents the number of nodes per ghost elements.

The other values are the ghost cell identifiers, and the second represents the element identifier which is next to the cell element. The third column is a factor that depends on the type of cell. If the cell is soft then a +1 factor should appear, if the cell is wall a −1 factor should appear, and if the cell is periodic it is not included.

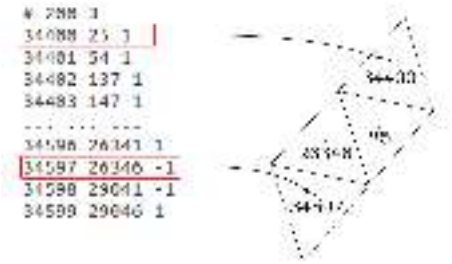


Fig. S14: GhostCells.txt file format.

S5.4 Physical Fields Initialization

Once the geometric and topological framework is established, the solver requires the initial physical state of the domain at $t = 0$. In the Finite Volume framework used here, all physical variables are cell-centered, meaning a single value is assigned to the centroid of each triangle. These fields include the static terrain (bathymetry) and the dynamic fluid properties (water level and fluxes).

The **Bathymetry.txt** text file is a standard one column format. The first character # means that the line is a comment. The first number – in this case 34600 – represents the number of elements in the mesh. The second number – in this case 1 – represents the number of cell-centered elements to be consider.

The other numbers are the bathymetry values at the center of mass for the i-th elements.

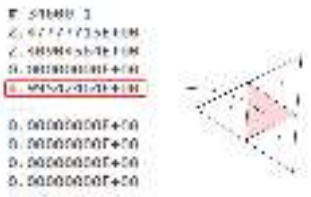


Fig. S15: Bathymetry.txt file format.

The **Discharge.txt** text file is a standard two-column format. The first line represents the number of elements in the mesh and the number of columns. The other values are the discharges values along the x- and y-axis at the center of mass for the i-th elements.

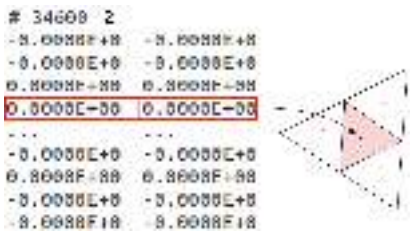


Fig. S16: Discharge.txt file format.

The **WaterLevel.txt** is a standard one column format. The first line represents the number of elements in the mesh and the number of columns. The other values are the water level surface values at the center of mass for the i-th elements.

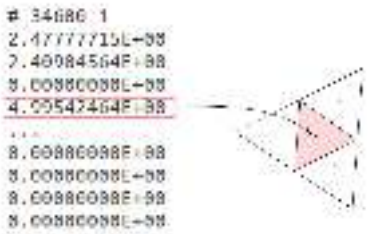


Fig. S17: WaterLevel.txt file format.

S6 Outputs & Post-Processing:

Once a simulation is executed, `SWEpy` manages the storage of massive amounts of GPU data by transferring and writing it to the CPU disk in standardized formats. This chapter describes the output directory structure, the technical specifications of the files, and a guide for professional visualization.

S6.1 Output Directory and Data Management

To maintain a clean workspace and prevent the accidental overwriting of input assets, `SWEpy` enforces a strict separation between source data and simulation results. Upon execution of the main driver script, the solver automatically creates a dedicated directory named `Paraview/` within the current working folder. This action is performed entirely by the software; the user does not need to manually create folders prior to the run.
This `Paraview/` directory serves as the centralized repository for all simulation deliverables, consolidating both the 2D spatial field snapshots (in .vtu format) and the high-frequency time-series text files (gauges). The directory name reflects the recommended open-source visualization platform, ParaView, although the standardized VTK XML formats employed are compatible with various other post-processing tools.

Balancing Temporal Resolution and Storage. The frequency at which data is written to disk is governed by the `dt_save` parameter defined in the `config.swe` file (refer back to Table S1 in Subsection S5.2). The user must strike a balance when setting this parameter

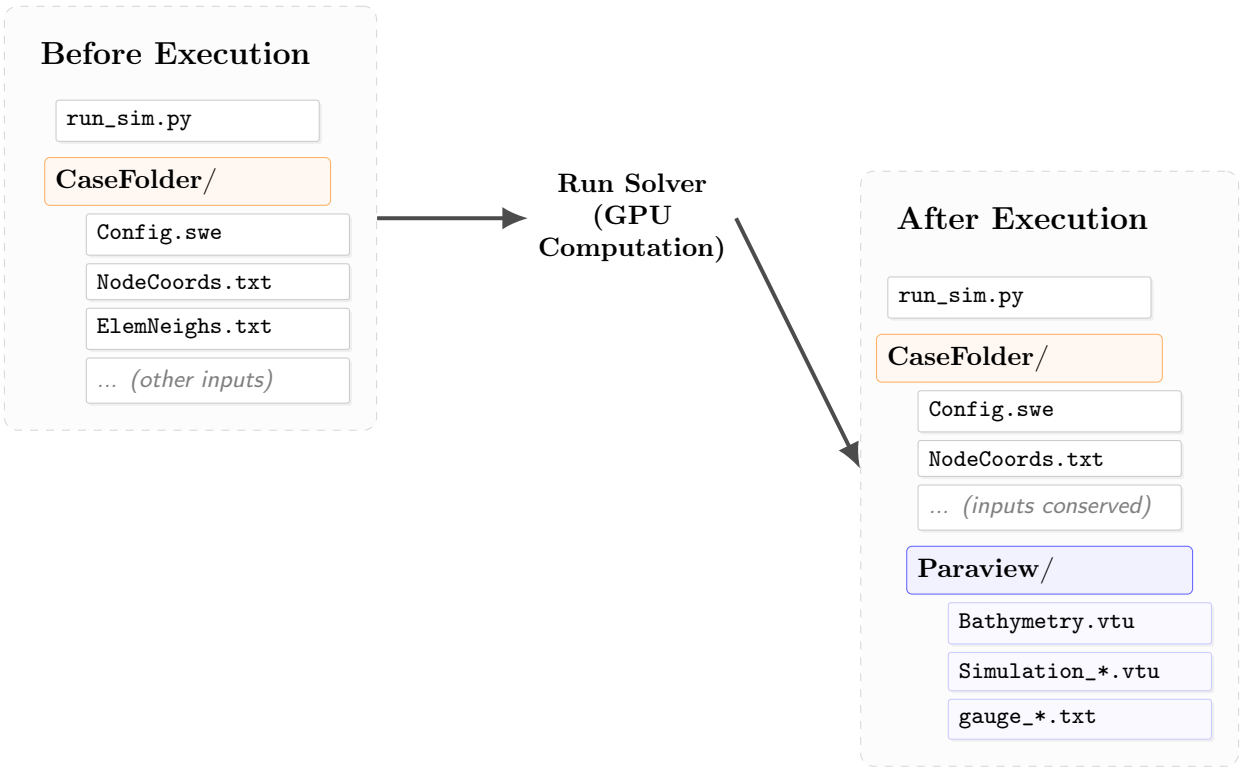


Fig. S18: Workflow of data management in SWEpy. The execution preserves all input files and isolates all simulation products into a dedicated `Paraview` directory to prevent clutter.

S6.2 Simulation Results: VTK Unstructured Grids (.vtu)

For spatial analysis, `SWEpy` adopts the **VTK XML Unstructured Grid (.vtu)** format. Unlike legacy VTK formats, the XML-based structure allows for high-fidelity representation of triangular meshes, supporting compressed data blocks and clear separation between geometry (points and cells) and physical attributes (scalars and vectors).
This format is particularly suited for the Finite Volume Method because it explicitly defines the connectivity of each triangular element, ensuring that gradients and shocks are visualized without the interpolation artifacts common in raster-based formats.

S6.2.1 Static Output: Bathymetry.vtu

The `Bathymetry.vtu` file is a cornerstone of the post-processing workflow, generated at $t = 0$ by the `cuFileSaver.save_bathymetry()` function. It establishes the topographic "master" of the simulation.

Following the VTK XML specification, the file is organized into a single `<Piece>` that defines the global geometry and static attributes:

The `Bathymetry.vtu` file is a cornerstone of the post-processing workflow, generated once at the initialization stage ($t = 0$) by the `cuFileSaver.save_bathymet` function. This file encapsulates the fixed geographic and topographic context of the simulation. Following the VTK XML specification, the file is organized into a single `<Piece>` that defines the global geometry and the static attributes of the domain:

- **Points (Nodes):** Stores the spatial coordinates of the mesh vertices. Although the solver operates in a 2D Cartesian plane, the file records nodes as 3-component vectors $(x, y, 0)$ to comply with the 3D rendering pipeline of visualization engines.
- **Cells (Connectivity):** Defines the triangular topology by indexing the vertices. This section includes the `offsets` and `types` (specifically VTK type 5 for triangles) necessary to reconstruct the unstructured grid.
- **CellData (Bathymetry):** The bed elevation B is stored as a scalar field mapped to the centroids of the triangles. By using `CellData` instead of `PointData`, `SWEpy` ensures that the topographic representation is perfectly consistent with the Finite Volume discretization, where bathymetry is assumed constant within each control volume.

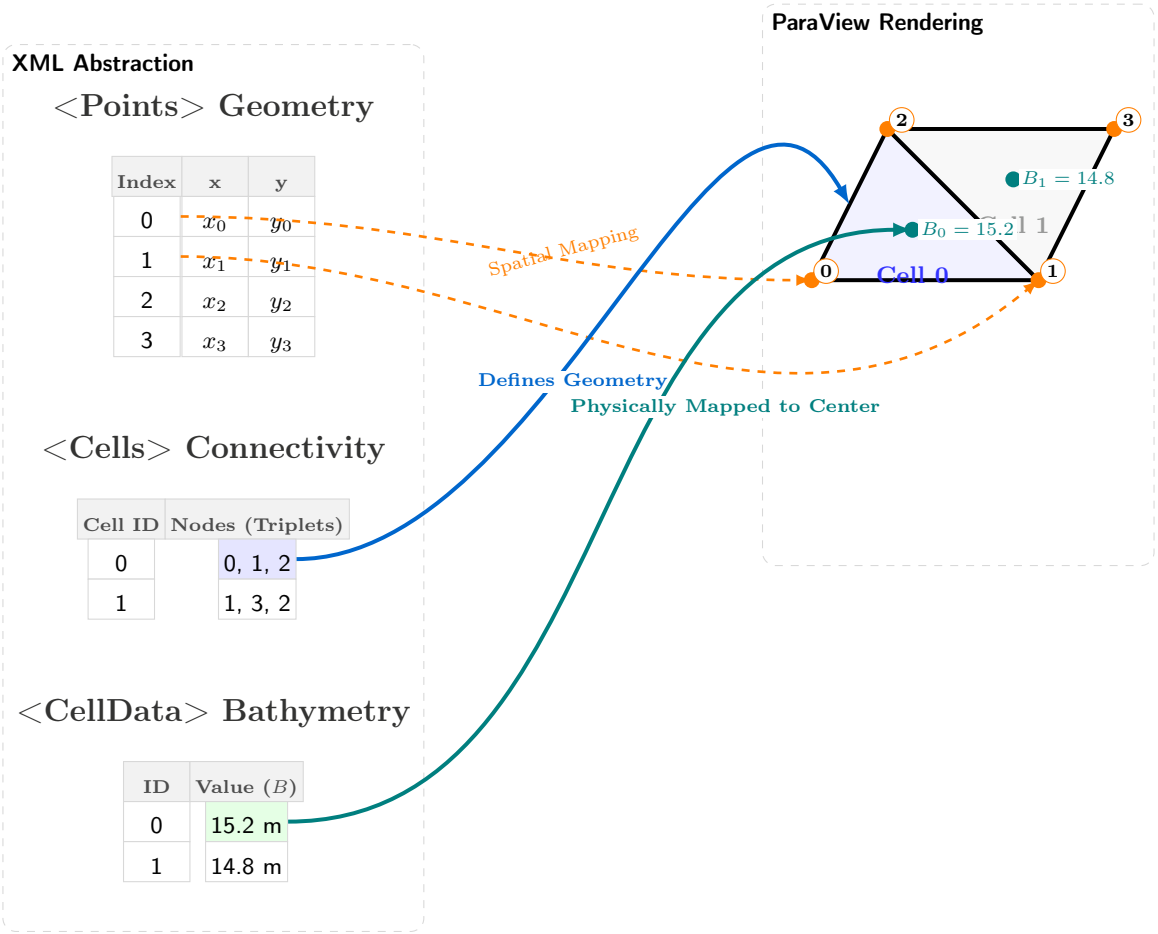


Fig. S19: Structural mapping and geometric materialization of the `Bathymetry.vtu` file. The left panel (*XML Abstraction*) represents the internal data arrays: *Points* define the spatial nodes, *Cells* establish the connectivity triplets, and *CellData* stores the physical scalars. The right panel (*ParaView* View) illustrates how the solver assembles these arrays into an unstructured mesh, specifically highlighting how bathymetry values (B) are centered at the triangle centroids to maintain consistency with the Finite Volume discretization.

This file acts as the topographic reference. In ParaView, it serves as a static background layer or as the geometry input for the *Warp By Scalar* filter to create a fixed 3D relief of the terrain before overlaying the dynamic water surface.

S6.2.2 Dynamic Output: Solution_*.vtu

The `Solution_*.vtu` files represent the evolution of the hydraulic state over time. Generated at each `dt_save` interval, these files provide a snapshot of the water surface and flow dynamics. While they share the same topological "skeleton" as the `Bathymetry.vtu` file, they update

the `CellData` field with the dynamic variables of the system. Following the VTK XML specification, each solution file is organized as follows:

- **Points (Nodes):** Identical to the bathymetry file, storing the $(x, y, 0)$ coordinates. This consistency allows visualization software to overlay flow data perfectly on top of the static terrain without spatial offsets.
- **Cells (Connectivity):** Reuses the triangular indexing (type 5). By maintaining a constant mesh throughout the simulation, the solver optimizes I/O operations and facilitates temporal tracking of specific elements.
- **CellData (Flow Variables):** This is the core of the dynamic file. It stores the state vector **U** at the centroids, interpreted as follows:
  - **WaterSurface (w):** Absolute elevation of the free surface (m).
  - **Depth (h):** Local thickness of the water column ($h = w - B$).
  - **Fluxes (q):** Stored as a 3-component vector array $[hu, hv, 0]$ representing the discharge per unit width (m^2/s). This formatting allows for the native generation of arrows and streamlines.

The files follow a rigid naming convention `Solution_[ID].vtu`, where the suffix is an integer. This allows visualization engines to recognize the files as a single temporal sequence. By loading this series, the user can:

- Compute temporal statistics (maximum water levels reached per cell).
- Visualize the movement of waves through the "Play" controls.
- Export high-quality animations of the flooding process.

S6.3 Visualization Workflow in ParaView

ParaView is the recommended open-source platform for the multi-dimensional analysis of `SWEpy` results. Due to its modular architecture based on the Visualization Toolkit (VTK), it allows users to build complex "filter pipelines" to transform raw numerical data into physical insights. While `SWEpy` solves the shallow water equations in a 2D depth-averaged framework, the outputs are specifically tailored with a 3D-ready structure (e.g., null Z-coordinates and 3-component vectors) to facilitate high-impact 3D rendering. For detailed information on advanced ParaView features, users are encouraged to consult the official ParaView User Documentation.

S6.3.1 Data Ingestion and Temporal Series

The first step involves loading the spatial data into the ParaView *Pipeline Browser*.

1. Terrain Initialization: Load `Bathymetry.vtu`. This file provides the static geometric reference.
2. Temporal Recognition: When loading the simulation snapshots, select the `Solution_*vtu` group. ParaView's XML reader automatically recognizes the integer suffix as a temporal sequence, enabling the VCR-style animation controls in the main toolbar.

S6.3.2 Topographic Reconstruction (Warp by Scalar)

Since the mesh is stored as a flat plane, we must "extrude" the geometry using the bathymetry values to visualize the relief.

1. Select the `Bathymetry.vtu` source.
2. Apply the **Warp By Scalar** filter.
3. In the *Properties* panel, ensure the *Scalars* array is set to `Bathymetry`. This filter displaces each node vertically based on the cell-centered elevation value.
4. Note on Consistency: `SWEpy` outputs all points as 3-component vectors $(x, y, 0)$ specifically to allow this filter to operate along the Z-axis by default.

S6.3.3 Vector Field Analysis

The fluxes array in `SWEpy` is stored as a vector $[hu, hv, 0]$. This formatting is optimized for the **Glyph** filter, which populates the mesh with arrows or symbols representing flow direction and intensity. This is particularly useful for identifying high-velocity jets or recirculation zones that are not visible through surface elevation alone.

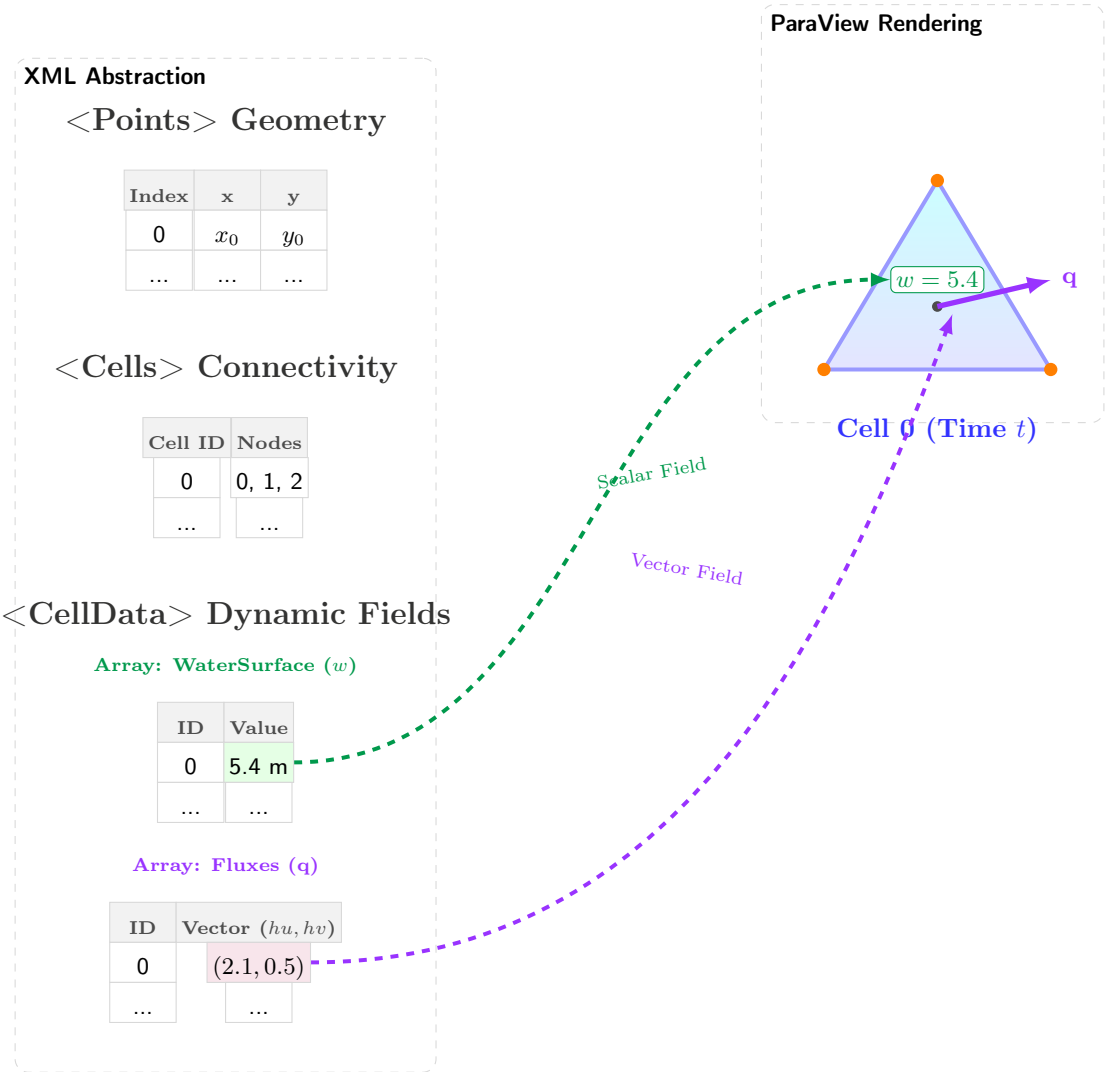


Fig. S20: Structural mapping and dynamic data representation in `Solution_*.vtu` files. The left panel (*XML Abstraction*) details how the state vector $\mathbf{U}$ is stored: *WaterSurface* (w) is recorded as a scalar array, while *Fluxes* ($\mathbf{q}$) are stored as 3-component vectors. The right panel (*ParaView Rendering*) shows the materialization of this data.

S7 Validation and Benchmarks

S7.1 Fundamental Conservation Properties

This section verifies the solver's ability to maintain basic physical balances. It ensures that the numerical scheme is well-balanced (preserving stationarity) and conservative (transporting mass and energy without artificial dissipation).

S7.1.1 Spatial Convergence Order

To validate `SWEpy`'s spatial accuracy and verify the correct implementation of the bathymetry source term discretization (S26), we replicate the convergence test of [7], focusing on scenarios where well-balancing is essential to preserve steady states over non-flat topography without introducing spurious oscillations. The test quantifies the order of convergence for different spatial reconstruction operators (Section S3.4) in the presence of a smooth Gaussian bump.

The computational domain is a 2×1 m rectangle discretized into a regular mesh of equilateral triangles, with triangles along the top and bottom boundaries halved for consistent boundary treatment. The bottom topography is defined as

$$B(x, y) = 0.5 \exp\big(-25(x-1)^2 - 50(y-0.5)^2\big). \tag{S42}$$

The initial conditions are a uniform free-surface elevation $w(x, y, 0) = 1.0$ m and velocity field $u(x, y, 0) = 0.3$ m/s, $v(x, y, 0) = 0$. Fully permeable (zero-gradient) boundary conditions are applied on all sides, with $g = 1$ m/s^2. The flow evolves to a steady, non-uniform state by $t \approx 0.07$ s, at which point temporal errors are negligible and spatial errors dominate.

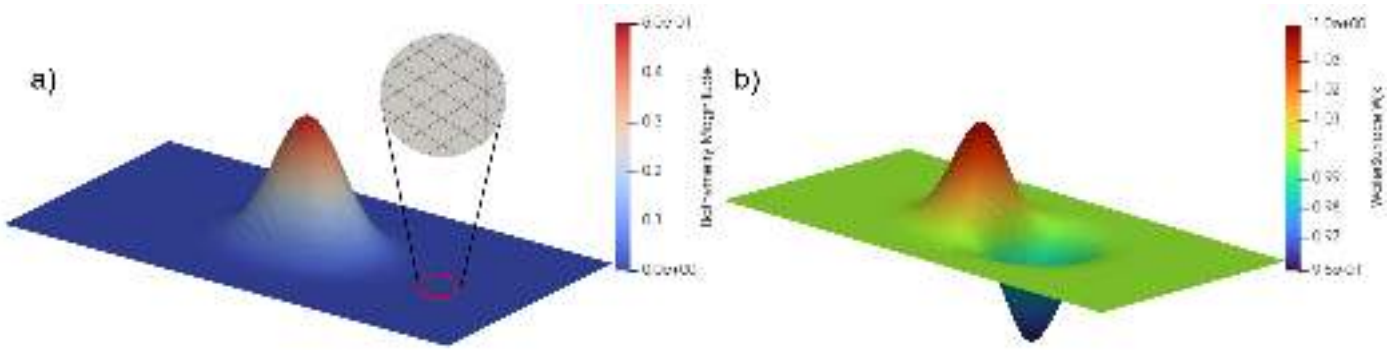


Fig. S21: Bottom topography on a $n_x = 32$ point grid (left) and reference solution (right) for the Gauss bump. The zoomed-in circular window highlights the grid structure pattern.

The reference solution is computed at $t = 0.07$ s on a fine grid with $n_x = 512$ horizontal divisions, corresponding to approximately 1.18×10^6 cells. Figure S21 illustrates the Gaussian bump on a coarse grid ($n_x = 32$) and the corresponding reference solution. The L^2 error is defined as

$$\|e\|_2 = \sqrt{\sum_j |\Omega_j| \left(\overline{w}_j - w_{\mathrm{ref},j}\right)^2},$$

and convergence orders are estimated via successive grid refinements. Table S7.1.1 reports errors and orders for grids $p = 0$ to 3 ($n_x = 32 \cdot 2^p$) across reconstruction–timestepper combinations, while figure S22 shows the log–log relationship between error and effective grid spacing Δx, with fitted power laws confirming the observed slopes.

Grid	Const. - FE		Lin. - FE		Lin. - RK4,3	
	L^2 *Err*	Ord	L^2 *Err*	Ord	L^2 *Err*	Ord
$p = 0$	1.7E-3	-	6.4E-4	-	6.4E-4	-
1	7.6E-4	1.15	1.6E-4	1.91	1.6E-4	1.90
2	3.1E-4	1.28	3.9E-5	1.63	3.8E-5	1.78
3	1.2E-4	1.43	9.6E-6	2.01	8.8E-6	2.12
Grid	Quad. - FE		Quad.- RK4,3		Quad. - RK4	
	L^2 *Err*	Ord	L^2 *Err*	Ord	L^2 *Err*	Ord
$p = 0$	2.9E-4	-	2.9E-4	-	2.9E-4	-
1	6.7E-5	1.90	6.6E-5	1.90	6.6E-5	2.13
2	1.6E-5	1.78	1.5E-5	1.78	1.5E-5	2.11
3	3.9E-6	2.02	3.6E-6	2.12	3.5E-6	2.14

tableL^2 errors and numerical orders of accuracy. Grid number p corresponds to $nx = 32 \cdot 2^p$ horizontal divisions of the domain.

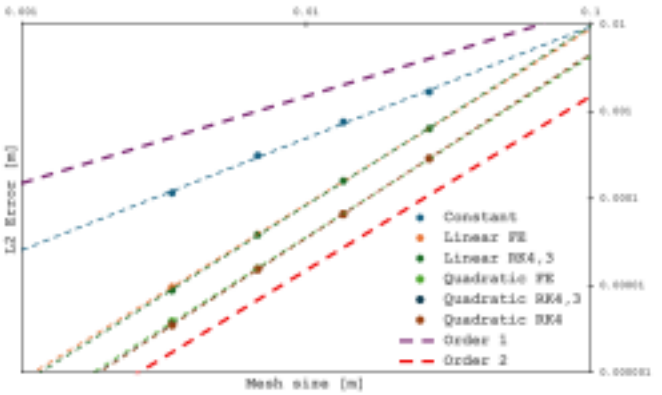


Fig. S22: Log-log plot of L^2 error versus grid size Δx, with fitted power-law curves indicating convergence orders.

The results confirm the robustness of all configurations. Constant reconstruction yields better-than-first-order accuracy, while linear and quadratic reconstructions approach second-order convergence, in agreement with the scheme's formal order for smooth solutions. The theoretically attainable third-order accuracy with quadratic reconstruction is not achieved, suggesting that further refinement of either the numerical flux formulation or the bathymetry source term discretisation may be needed to fully realise higher-degree polynomial benefits. Nevertheless, WENO-based quadratic reconstruction consistently produces the smallest errors, outperforming lower-order approaches across all resolutions. This improved accuracy is particularly relevant in precision-critical scenarios, where error propagation over long timescales can be significant. From a performance standpoint, generating the fine-grid reference solution with forward Euler time-stepping and linear reconstruction required approximately 5 minutes wall-clock time on consumer-grade GPU hardware, including high-frequency (0.01 s) output for animation purposes. This demonstrates that `SWEpy` can deliver high-resolution, well-balanced solutions for smooth-bottom flows at modest computational cost.

S7.1.2 Well-balancing test

A simulation of a static flat water surface is performed over white-noise-generated bathymetry to verify the well-balanced property for all reconstruction operators. The domain is a 1×1 [m] box, with a random bathymetry ranging from $-1.1d$ to $-0.9d$ and average depth of $d = 2$, discretized using a right triangles grid with $\Delta x = \Delta y = 0.01$. The initial water surface is set to 0 [m] and the velocity profile is null in both directions. Figure S23 shows the random bathymetry and the maximum water height over time for the three reconstructions, confirming no spurious oscillations arise.

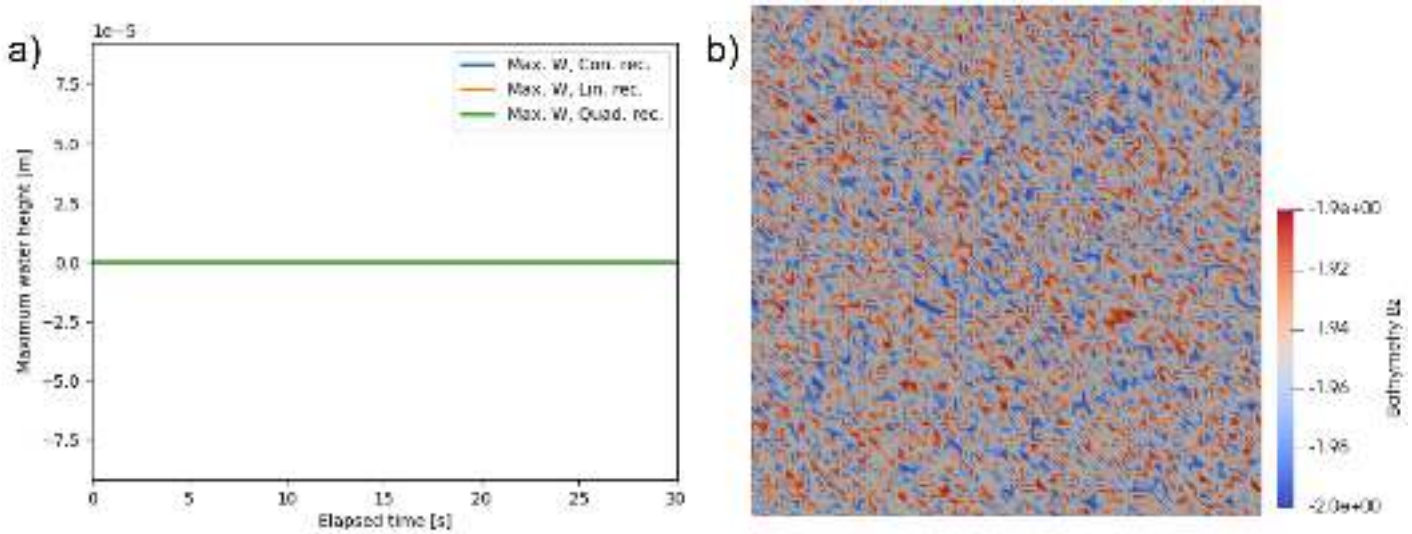


Fig. S23: Randomly generated bathymetry (top). Maximum water height over time (bottom).

These results, while potentially surprising, are in fact anticipated since our bathymetry source term is exactly well balanced. This means that even in the presence of boundary conditions, unphysical oscillations or flows will not appear.

S7.2 Analytical Benchmarks

These tests compare the numerical solution against exact mathematical solutions of the Shallow Water Equations, verifying numerical dissipation and shock speed.

S7.2.1 The Dam Break Problem

Description: An instantaneous release of a wall of water separating two constant states ($h_L > h_R$). This generates a rarefaction wave traveling upstream and a shock wave traveling downstream.
Goal: Validate that the solver captures the correct shock speed and does not introduce excessive numerical diffusion at the discontinuity.

In the numerical experiment taken form [37], we consider a 150 $[m]$ long channel with soft conditions at both ends i.e, $x = -75$ $[m]$, and $x = 75$ $[m]$. Periodic condition were applied for the upper and lower boundaries i.e $y = -7.5$ $[m]$, and $y = 7.5$ $[m]$. The constant of gravity was set to be 9.80 $[m/s^2]$, and the water depth was taken as $h_1 = 1.0$ $[m]$. The computation uses a Manning's roughness coefficient of $n = 0.0$ for the inviscid flow.

Figure(S24) shows a comparison between the analytical profiles with the numerical results for the test case at $t = 0.0$ $[s]$, $t = 1.0$ $[s]$, $t = 2.0$ $[s]$, $t = 5.0$ $[s]$, $t = 7.0$ $[s]$, and $t = 10.0$ $[s]$. It is clear that a parabolic function occurs between the transition of both water levels. In this regard, a good agreements is reached for the studied case.

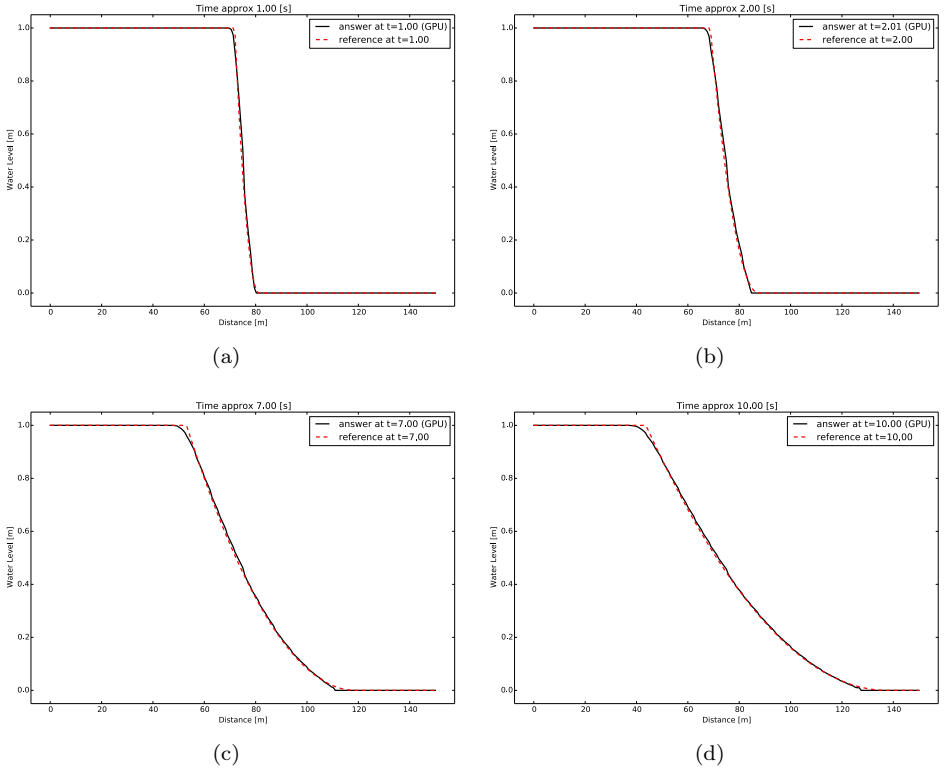
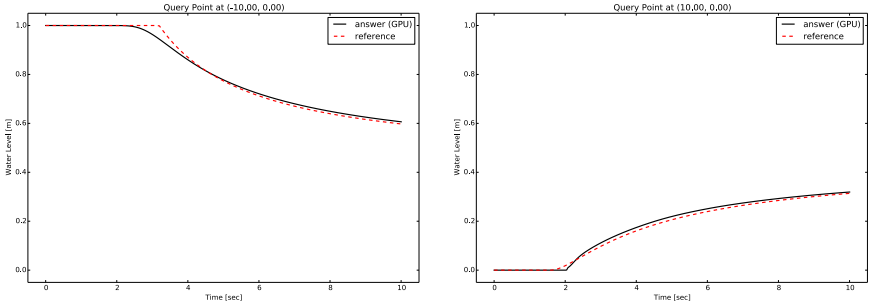


Fig. S24: Spatial profile for the dam break problem at time (a) $t = 1.0$ $[s]$. (b) $t = 2.0$ $[s]$. (c) $t = 7.0$ $[s]$. (d) $t = 10.0$ $[s]$.

Moreover, figure(S25) shows a comparison of the analytical time series for the test case at $x = -50.0$ $[m]$, $x = -10.0$ $[m]$, $x = 10.0$ $[m]$, and $x = 50.0$ $[m]$. It is clear that a good agreement is reached for the studied case.

Lessons learned:

1. A nice agreement is reached for the studied case in which a parabolic function is expected to occurs between the transition of both water levels.
2. The model provides a smooth representation in areas where high water level discontinuities are expected, as it is in time $t = 0$ $[s]$, and dry zone interfaces.



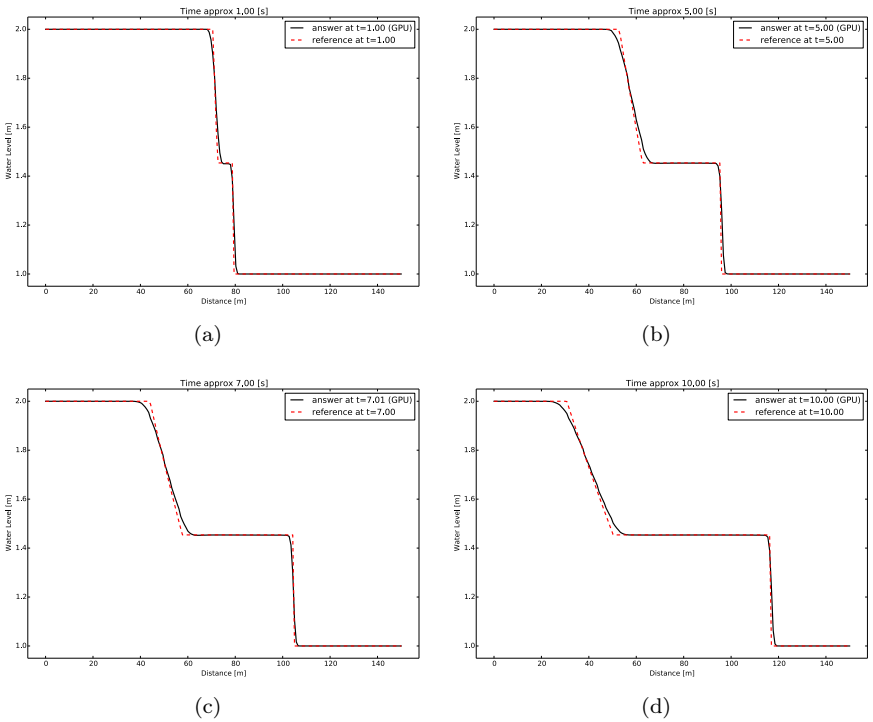
(a) (b)

Fig. S25: Time series values for the dam break problem at (a) $x = -10.0$ $[m]$. (c) $x = 10.0$ $[m]$.

S7.2.2 Hydraulic Jumps Over Constant Depth

In the numerical experiment taken form [37], we consider a 150 $[m]$ long channel with soft conditions at both ends i.e, $x = -75$ $[m]$, and $x = 75$ $[m]$. Periodic condition were applied for the upper and lower boundaries i.e $y = -7.5$ $[m]$, and $y = 7.5$ $[m]$. The constant of gravity was set to be 9.80 $[m/s^2]$, and the water depth were taken as $h_0 = 1.0$ $[m]$, and $h_1 = 2.0$ $[m]$ respectively. Considering the previous values, the constant state $h_2 = 1.4538$ $[m]$. The computation uses a Manning's roughness coefficient of $n = 0.0$ for the inviscid flow.

Figure (S26) shows a comparison between the analytical profiles with the numerical results for the test case at $t = 1.0$ $[s]$, $t = 2.0$ $[s]$, $t = 7.0$ $[s]$, and $t = 10.0$ $[s]$. It is clear that four different regions appear during the simulation. It is evident that the method efficiently captures the shock wave, as well as the refraction wave. In this regard, a good agreements is reached for the studied case.

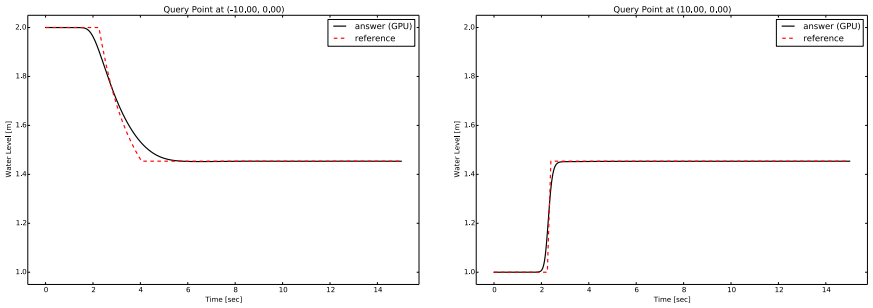


(a) (b)

(c) (d)

Fig. S26: Spatial profiles for the hydraulic jumps at time (a) $t = 1.0$ $[s]$. (b) $t = 3.0$ $[s]$. (c) $t = 10.0$ $[s]$. (d) $t = 15.0$ $[s]$.

Moreover, figure (S27) shows a comparison of the analytical time series for the test case at $x = -10.0$ $[m]$, and $x = 10.0$ $[m]$. It is clear that a good agreement is reached for the studied case.



(a) (b)

Fig. S27: Time series values for the hydraulic jumps at (a) $x = -10.0$ $[m]$. (b) $x = 10.0$ $[m]$.

Lessons learned:

1. A nice agreement is reached for the studied case in which a parabolic and a jump functions are expected to occurs between the transition of both water levels for the shock wave which travels to the right and the depression wave that travels to the left.
2. The model provides a smooth representation in areas where high water level discontinuities are expected, as it is in time $t = 0$ $[s]$, and dry zone interfaces.

S7.2.3 Synolakis' wave runup

We simulate the soliton runup over a sloped beach solved analytically by C.E. Synolakis in his Ph.D. thesis [41]. This numerical solution allows us to test the solver's time integration implementation (cf. sect. S3.6), its wet/dry treatment, and the use of the bathymetry source term. The domain consists of a 160×10 [m] channel, with a $1 : 19.85$ ascending slope at one end, discretized by a regular equilateral triangles mesh, of sidelength 0.666 [m].

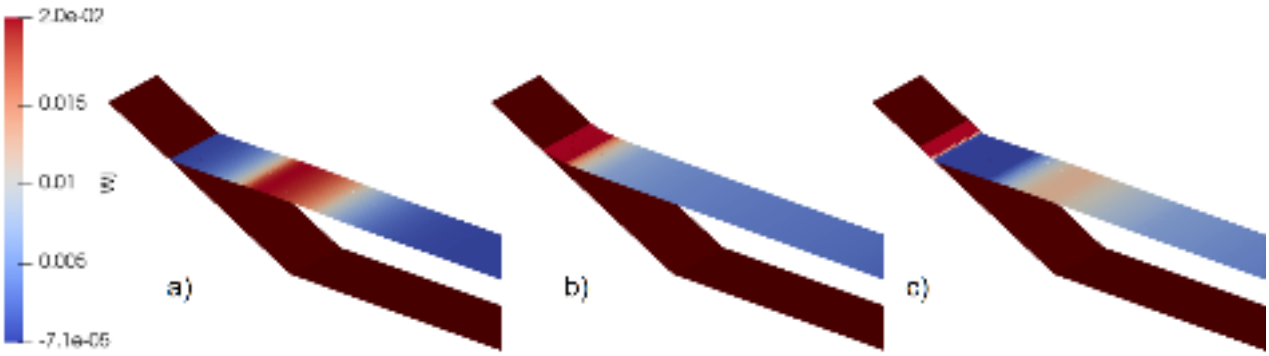


Fig. S28: Wave at the foot of the beach (a), at its maximum runup (b), and reflected (c).

The initial water surface is set as

$$w(x, y, 0) = \frac{H}{d}\text{sech}^2(\gamma(x - X_1)), \tag{S43}$$

where H is the maximum wave height taken as 0.0185 [m], d is the water depth taken as 1 [m], $\gamma := \sqrt{3H/4d}$, and X_1 is the wave's initial position, taken as 93.82 [m]. The initial velocity profile is set as

$$u(x, y, 0) = -\frac{g}{d}w(x, y, 0)\left(1 - 0.25\frac{w(x, y, 0)}{d}\right), \; v(x, y, 0) = 0.$$

A reference time series solution is numerically integrated from Synolakis' solution for the water height at point $(x_0, y_0) = (19.85, 0)$. This time series is plotted together with SWEpy's results for the cell containing the point in the computational domain using the different combinations of spatial reconstruction and temporal integration in figure S29. All solutions were calculated using a timestep size $\Delta t = 0.07$.

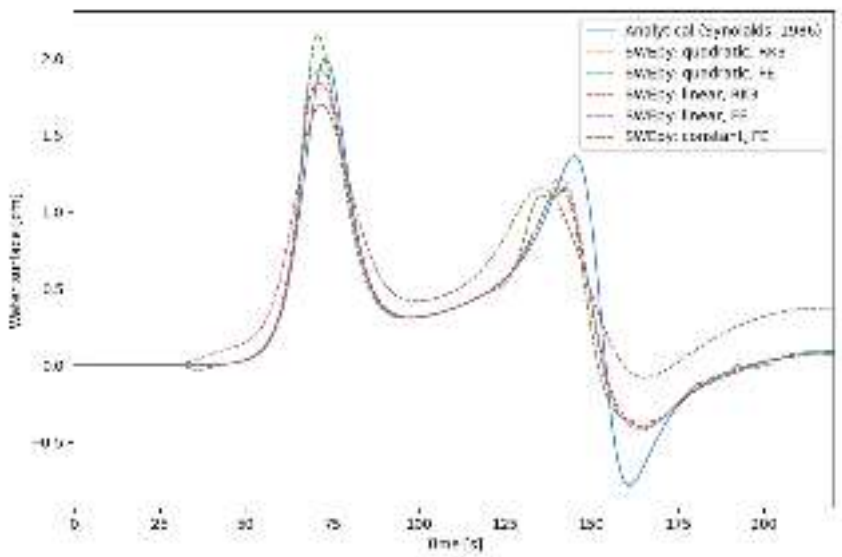


Fig. S29: Time series of water height at point $X_0 = (19.85, 0)$ for analytic solution and SWEpy results.

tableRunup comparison.

Solution	**Runup**
Constant	0.0427
Linear	0.0800
Quadratic	0.0801
Analytical	0.0861

Synolakis defines the *runup law* to calculate the maximum runup of a soliton in this problem as

$$R = 2.831 d (\cot \beta)^{1/2} (H/d)^{5/4}.$$

In our case, the theoretical runup would be $R \approx 0.08605$ [m]. Maximum water heights over time are used to calculate the runup in each of our solutions. This could also be calculated by refining the mesh at the beach and using the subcell resolution given by the reconstruction to look at the highest wet cell's reconstructed water surface.
Additionally, we present the results of a numerical experiment designed to evaluate mass conservation.
Figure S30 shows the curves representing the evolution of mass as a function of time considering different spatial reconstructors. All runs were performed using the Explicit Euler time evolution scheme, and an adaptive Δt with $CFL = 0.25$

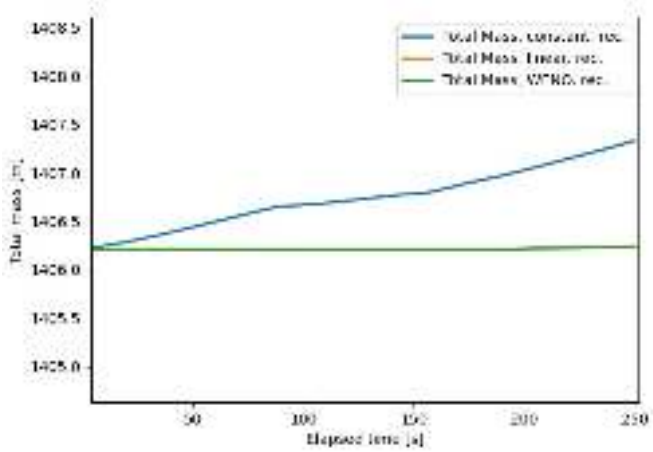


Fig. S30: Mass values as temporal function for different reconstruction method.

Table S2 shows the reference value computed from the same initial condition for each experiment and the maximum value attained during the simulation.

	Initial mass	Constant	Rel. error	Linear	Rel. error	WENO	Rel. error
Mass	1406.210	1407.331	7.97×10^{-4}	1406.242	2.28×10^{-5}	1406.235	1.78×10^{-5}

Tab. S2: Mass values and relative errors with respect to the initial mass for different reconstruction methods.

The results are reported with seven significant figures in order to highlight the differences obtained between the linear and WENO reconstructions. For these reconstruction operator, the relative errors demonstrate that the scheme preserves mass with an error below 10^{-5}.

S7.3 Numerical Configurations

S7.3.1 Grid geometry study: influence of mesh orientation

To investigate the influence of mesh geometry on solution quality, particularly symmetry preservation and oscillatory behavior, we simulate a circular dambreak problem on regular right-triangular (rectangular) and equilateral triangular grids. This test assesses how reconstruction operators (Section 3.4) interact with grid anisotropy, a key factor in unstructured mesh applications where non-equilateral elements may induce directional biases, potentially affecting the accuracy in radial flows like dam breaks or tsunamis **(AGREGAR FUENTE)**.
The domain, as shown in Figure S31, is a flat 400 × 400 m rectangular area with permeable (soft) boundaries to minimize reflections and allow unimpeded radial outflow, simulating open conditions while preserving conservation of mass and momentum. The initial water surface features a cylindrical column of height 1 m within a 20 m radius at the center, superimposed on a uniform 0.5 m depth elsewhere, with zero initial velocity–promoting symmetric expansion driven by hydrostatic pressure gradients. Figure S31(a) shows this setup on the right-triangular (rectangular) grid, comprising 20,000 cells with $\Delta x = 1$ m (formed by bisecting squares along diagonals), while (b) illustrates the equilateral triangular grid with 22,914 cells of edge length ≈ 0.8952 m, ensuring comparable resolution ($\Delta x \sim 1$ m). Zoomed insets highlight the structures: the rectangular grid's diagonals introduce 45° biases in hypotenuse normals, potentially inducing anisotropy, whereas the equilateral grid provides isotropic connectivity for better symmetry preservation.

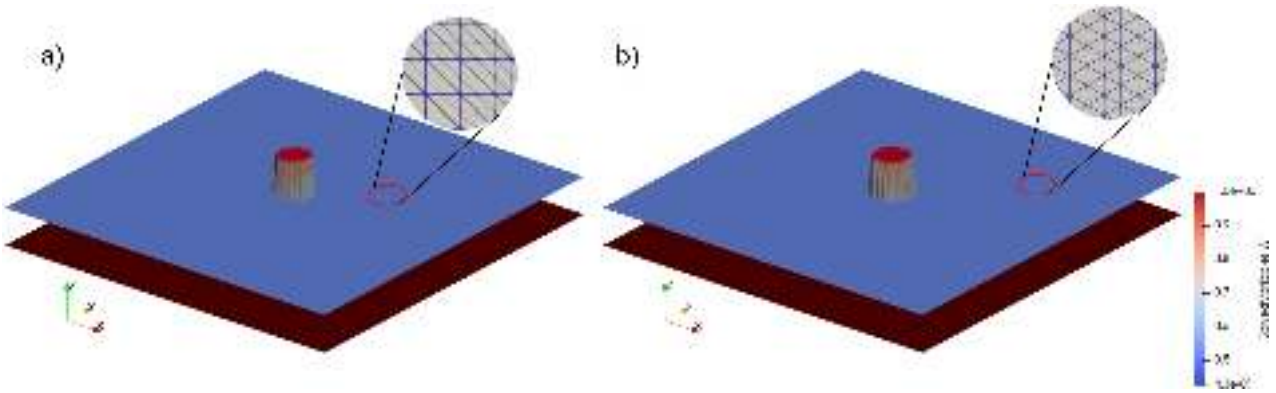


Fig. S31: Initial water surface with a right triangular grid (a) and equilateral grid (b). The zoomed-in view in both figures highlights the grid structure.

Simulations employ the Forward-Euler integrator with both linear (minmod) and quadratic (WENO) reconstructions, evolving to $t = 5$ s—a timescale where radial symmetry should persist but numerical artifacts may emerge. Figure S32 presents the water surface at $t = 5$ s across the four combinations.

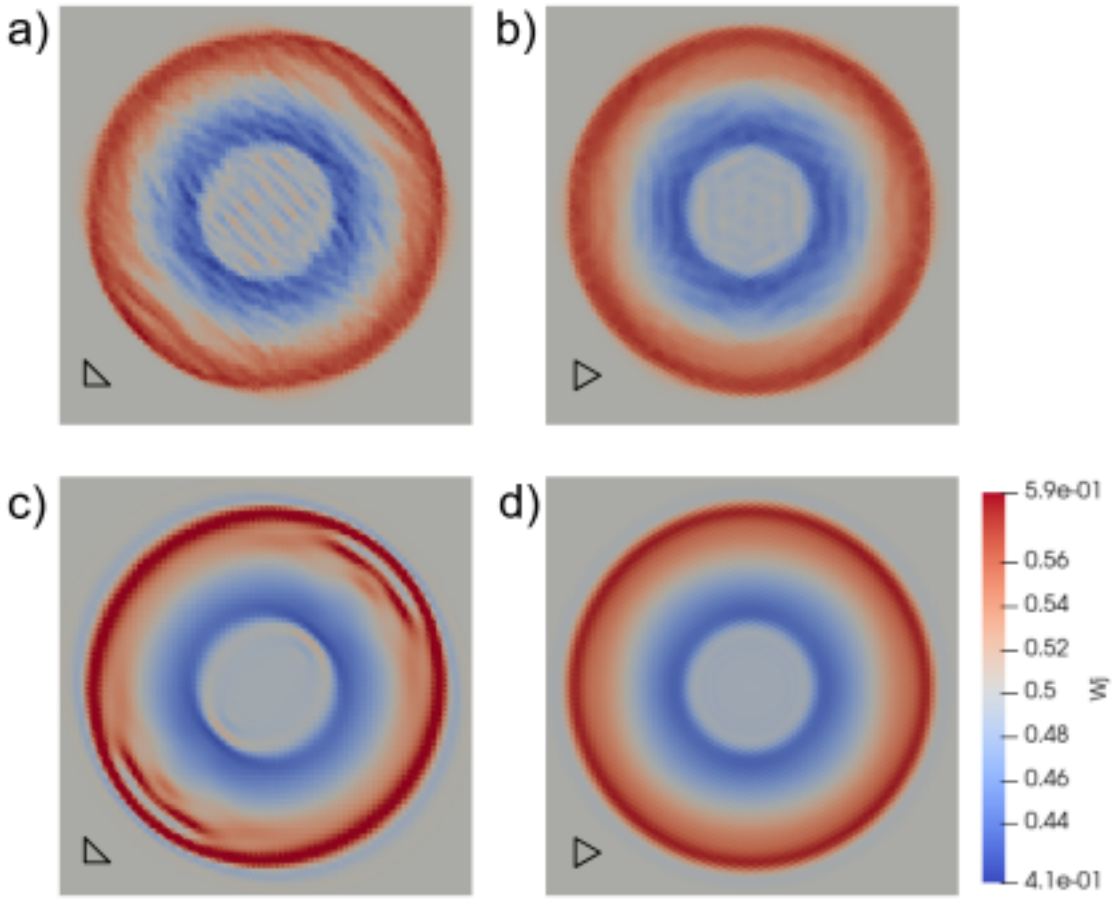


Fig. S32: Grid geometry study: Solution to the circular dambreak problem at $t = 5$ [s], using a rectangular grid (left column) and an equilateral grid (right column). Top row: results obtained with linear reconstruction. Bottom row: results computed with quadratic reconstruction. The black triangles display the grid pattern employed.

On the rectangular grid, solutions exhibit preferential directions, with enhanced oscillations along 45° axes–aligning with hypotenuse normals of the right triangles–leading to symmetry loss and distortion of the circular wavefront. This anisotropy violates radial invariance, inducing preferential fluxes that favor diagonal propagation, resulting in energy leakage and unphysical patterns. In contrast, equilateral grid solutions maintain superior symmetry and circularity, with minimal oscillations under linear reconstruction and near-perfect isotropy with WENO, underscoring the benefits of uniform edge orientations for conserving momentum in axisymmetric flows.
WENO consistently yields less diffusive solutions, preserving height concentration near the wavefront (e.g., sharper radial gradients in bottom panels), even on the suboptimal rectangular grid. To quantify this, we compute conserved energy via numerical integration of

$$\mathcal{E} = \rho g \int_{\Omega_{ring}} \eta^2 \, d\Omega \tag{S44}$$

where η is the water surface displacement with respect to a reference value, taken as $\eta = w - 0.5$ in this case, within an annular ring (outer radius $r_{\text{out}} = 32.5$ m, inner radius $r_{\text{in}} = 28$ m at $t = 5$ s): for rectangular grid, $\mathcal{E}_{\text{minmod}} = 2.9768$ and $\mathcal{E}_{\text{WENO}} = 4.57458$; for equilateral, $\mathcal{E}_{\text{minmod}} = 3.44726$ and $\mathcal{E}_{\text{WENO}} = 3.82919$. These indicate $\sim 54\%$ higher energy retention with WENO on rectangular grids and $\sim 11\%$ on equilateral, confirming its superiority in mitigating dissipation while respecting conservation laws.

S7.3.2 Instability study: impact of temporal discretization

In this part, we investigate the impact of time integrators on the development of instabilities in numerical simulations. To this end, we consider the transport of a soliton along a rectangular channel as a benchmark experiment. The primary challenge is to identify an appropriate mesh configuration together with the reconstruction operator that effectively reduces numerical diffusivity. The objective is to provide a basis for the analysis of far-field propagation simulations, with particular relevance to the study of the Mw 8.8 2010 Maule tsunami. Specifically, we are interested in identifying numerical configurations that can be associated with the onset of instabilities, manifested through the generation of oscillations and the amplification of their amplitudes.
The channel is 500×5 m with soft (permeable) end boundaries to absorb waves without reflection, and periodic top/bottom boundaries for transverse uniformity. Discretization uses an equilateral triangular mesh with edge length 0.25 m (94,000 cells) for isotropy, and a rectangular

triangular mesh with $\Delta x = \Delta y = 0.25$ m (80,000 cells) to highlight anisotropy. The initial surface profile considered in this analysis is written as:

$$w(x, y, 0) = \frac{H}{d}\text{sech}^2(\gamma(x - X_1)), \tag{S45}$$

with d=1 m, H=0.25 m, $\gamma := \sqrt{3H/4d^3}$, $X_1 = 50$ m, and zero velocity, inducing symmetric fission. Pre-runs with adaptive Δt yield a fixed $\Delta t = 0.00824$ s for consistent comparisons under EE integrator to isolate spatial effects.
Simulations are first performed using the FE time integrator, to isolate the differences between spatial reconstructors. Three pre-runs are done with variable timestep to identify an appropriate size to force across all simulations, resulting in $\Delta t = 0.00824$ used for the main runs.
The simulations are also carried on a rectangular triangles mesh of $\Delta x = \Delta y = 0.25$ m and 80000 elements, and the same timestep size, to highlight the influence of the mesh geometry. A view of the right-going wave after 10 s and ~ 34 m of traveling is shown in figure S33 for each case.

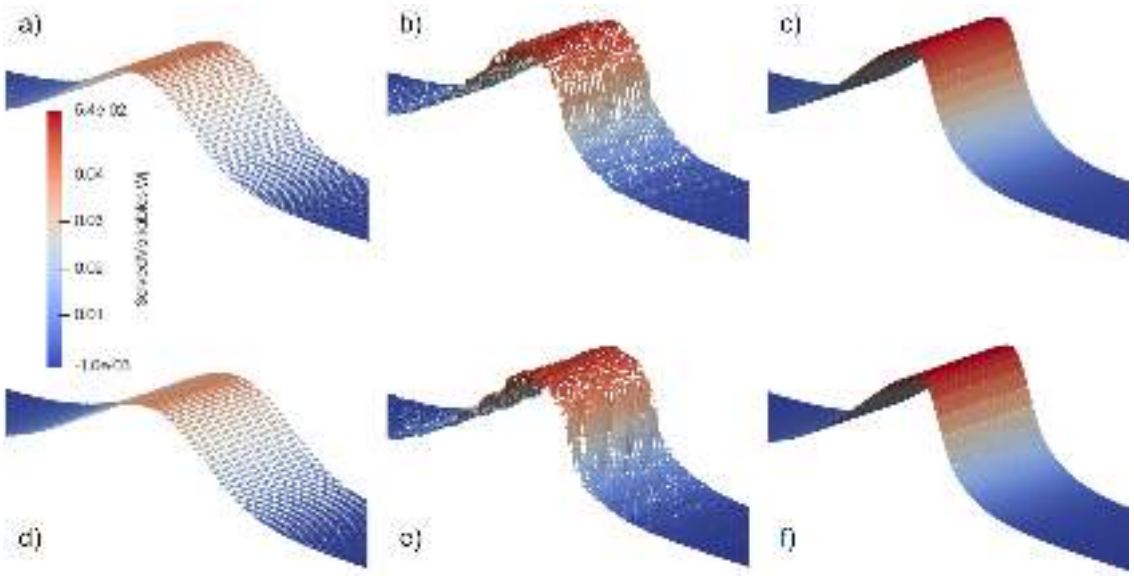


Fig. S33: Wavefront of traveling split wave for constant reconstruction (left), linear reconstruction w/$\vartheta = 1.4$ (middle), and quadratic reconstruction (right) on equilateral mesh (top) and rectangular mesh (bottom).

It can be seen that the WENO reconstruction preserves the wave height the most, while also conserving the smoothness of the wave structure; the linear reconstruction is somewhat more diffusive, but oscillations appear as a result of the usage of a diffusion coefficient $\vartheta > 1$. Finally, the constant reconstruction is also smooth, but its numerical diffusion is the highest.
Based on the previous results, the analysis is continued by considering the WENO reconstruction in combination with a rectangular mesh. The simulations are performed on four different right-triangle mesh sizes, using both the FE and RK(4,3) time integration schemes. The characteristic mesh parameters are described below:

$$(n_x, n_y) \in \big\{(2000, 20),\ (1000, 20),\ (500, 20),\ (100, 5)\big\} \tag{S46}$$

The mesh numbering follows the previous presentation; that is, Mesh 1 corresponds to $(n_x, n_y) = (2000, 20)$, and similarly for the remaining meshes.
The wave profiles calculated using each timestepper over the domain up to the front after 60 seconds and ~ 240 m of travel are shown for each mesh refinement level. Los resultados de las simulaciones se presenatn en la figura S34.

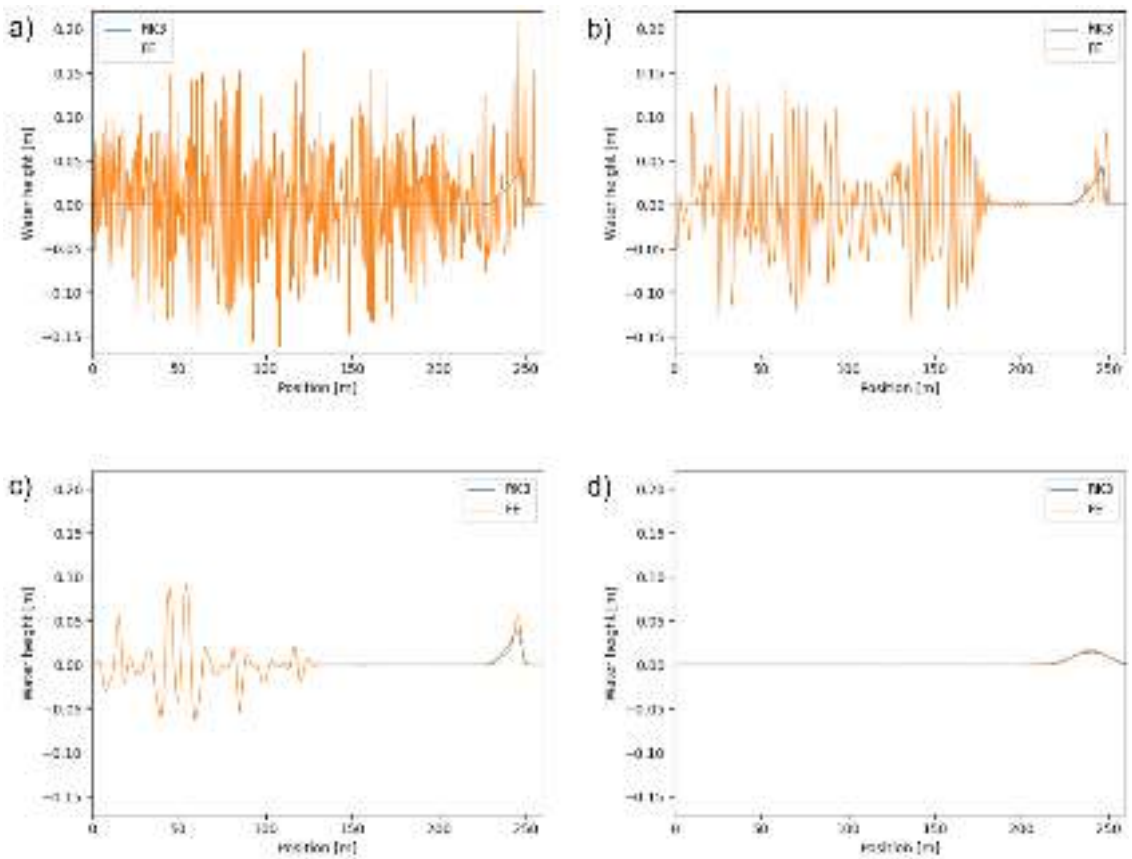


Fig. S34: Water level for the domain after 60 [s] using RK4,3 and FE for meshes 1-4 (a-d).

It can be observed that in all simulations performed with the RK4,3 scheme, the resulting profiles exhibit a partially comparable qualitative behavior, with the most diffusive solution being obtained on mesh-4 with $(n_x, n_y) = (100, 5)$. In turn, the mesh refinement enhances the solution when using the RK4,3 integration, while introducing oscillations that eventually get out of control for the FE scheme. Controlling these oscillations requires lowering the timestep size (or the user-defined CFL constant), which then would result in more diffusion for the solution. This highlights the care needed when using the Explicit-Euler timestep, suggesting that its better (less diffused) solutions may be caused by amplified oscillations (spurious oscillations).
In the plan-view representations of the simulations, the soliton oscillations are generated primarily in the region of the domain associated with periodic boundary conditions, suggesting that their origin may be related to a resonance-type mechanism ya que las oscilaciones no se disipan. A similar behavior, characterized by the generation of high-frequency oscillations, is observed in the tsunami simulations, where the far-field propagation regime promotes the development of oscillatory patterns analogous to those described above con cualidades convectivas en el espacio.

S7.4 Laboratory Benchmark

S7.4.1 Run Up Of Solitary Wave Over A Sloping Beach

In our numerical experiment, taken form [44], we consider a $80[m]$ long channel with soft conditions at both ends i.e., $x = -10$ $[m]$, and $x = 70[m]$. Periodic condition were applied for the upper and lower boundaries i.e $y = -5[m]$, and $y = 5[m]$. We use the parameters $x_0 = 19.85$ $[m]$, $x_1 = 37.35$ $[m]$, $d = 1.00$ $[m]$, the gravitational constant was taken as $g = 9.80$ $[m/s^2]$. Surface roughness becomes important for runup over gentle slopes and a Manning's coefficient $n = 0.01$ describes the surface condition of the smooth glass beach in the laboratory experiments. Figure(S35) provides a schematic of the experiments indicating the parameters employed:

Figure(S36) shows a comparison of the measured profiles for the test case with the solitary wave height ratio of $H/d = 0.0185$. The laboratory data show wave breaking between $t = 20$ $[s]$ and $t = 25$ $[m]$ as the solitary wave reaches the beach and development of a hydraulic jump at $t = 50$ $[s]$ when the water recedes from the beach. Both numerical solutions show very good agreement with the laboratory data as the solitary wave shoals to its maximum height at $t = 20$ $[s]$.

Figure(S37) shows a comparison of the time series between the nonlinear theory model for a $H = 0.0185$ $[m]$ solitary wave climbing up a $1 : 19.85$ beach at $x = 19.85$ $[m]$, $x = 15.71$ $[m]$, $x = 9.95$ $[m]$, $x = 5.10$ $[m]$, $x = 0.74$ $[m]$, and $x = 0.25$ $[m]$. The profiles are plotted as a

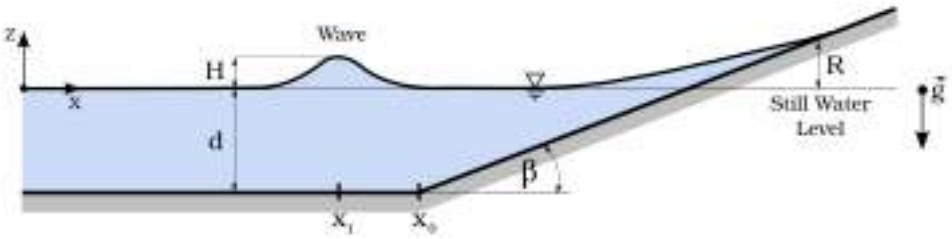
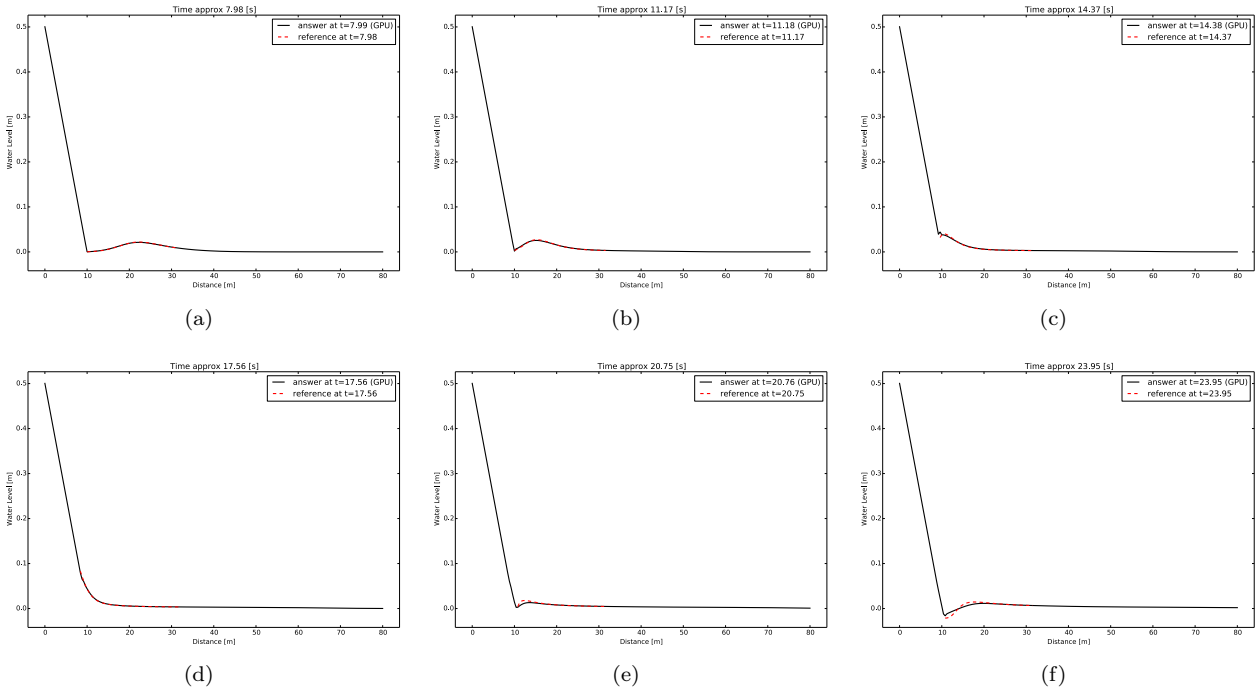


Fig. S35: Definition sketch of solitary wave run-up on a plane beach.



(a) (b) (c)

(d) (e) (f)

Fig. S36: Surface profiles of a solitary wave on a 1:19.85 plane beach at (a) $t = 7.985$ $[s]$, (b) $t = 11.1789$ $[s]$, (c) $t = 14.374$ $[s]$, (d) $t = 17.568$ $[s]$ (e) $t = 20.762$ $[s]$, (f) $t = 13.957$ $[s]$.

function of the time at different locations away from the initial position of the shoreline.

For $x = 19.85$ $[m]$, $x = 15.71$ $[m]$, $x = 9.95$ $[m]$, and $x = 5.10$ $[m]$, there is no significant difference among the three profiles. However, closer to the shoreline, at $x = 0.25[m]$, and $x = 0.74[m]$, the nonlinear effects become important and the linear theory greatly overestimates the wave profile. It is also apparent that dissipation attenuates the wave height close to the shoreline. An interesting feature of these surface profiles is their behavior during the time interval from approximately $t = 125$ $[s]$ to $t = 138$ $[s]$, where the backwash, or rundown of the wave retreats beyond the measurement location, and there is no flow depth to be measured. The shoreline then returns but does not stop moving at its initial position; it continues its motion and behaves as an underdamped oscillator.

Figure(**??**) shows the computed and measured runup ratio R/h as a function of the solitary wave height ratio H/h for beach slopes of $1 : 19.85$. The measured data show a bilinear distribution with the two branches representing the non-breaking and breaking regimes separated by a transition. This result clearly matches the ones presented by Synolakis, see [40].

Lessons learned:

1. This benchmark problem is a good test of the shallow water wave computation against an analytic solution in one dimension.

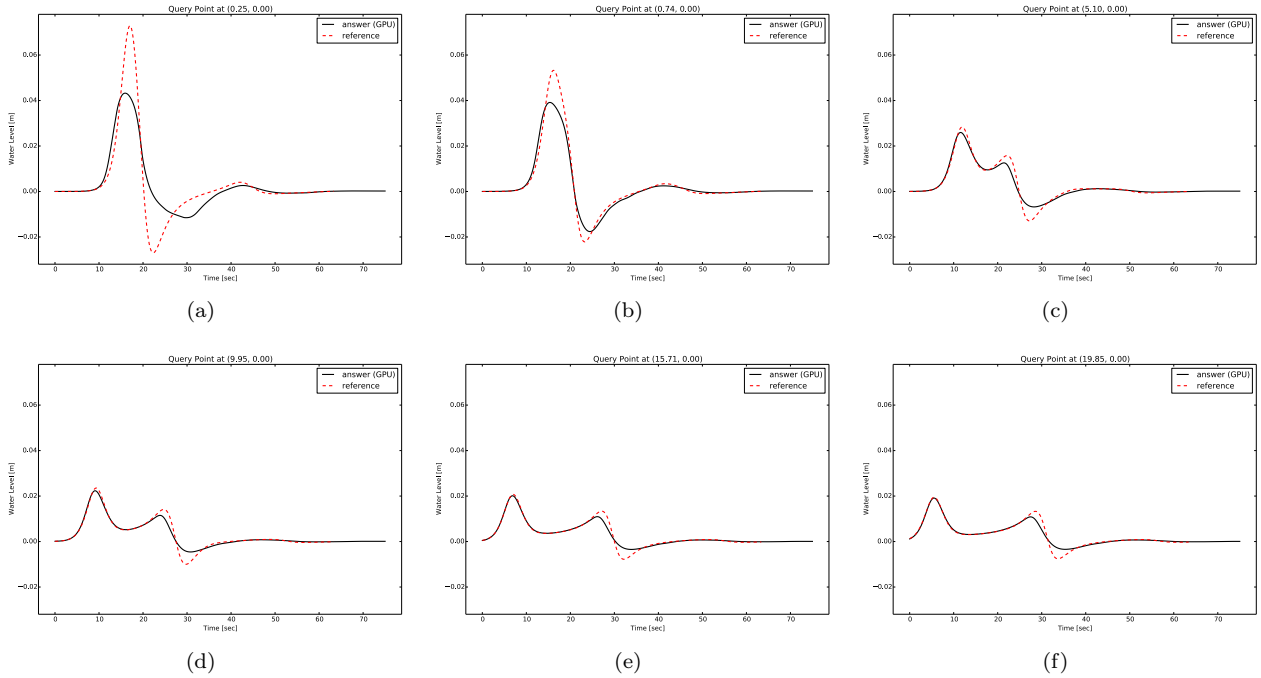


Fig. S37: Time series of a solitary wave on a 1:19.85 plane beach at (a) $x = 0.25$ $[m]$, (b) $x = 0.74$ $[m]$, (c) $x = 5.10$ $[m]$, (d) $x = 9.95$ $[m]$ (e) $x = 15.71$ $[m]$, (f) $x = 18.85$ $[m]$.

2. The time series solution are different at $x = 0.25$ $[m]$, $x = 0.74$ $[m]$ because they are compared with the linear solution. If the non-linear solution is employed, then the results are the same, see [40].
3. Because of its complexity, the analytical solution should be provided in a data file on the benchmark problem website to ensure all participants are solving the same problem.

S7.4.2 Solitary Wave On A Composite Beach

Models that perform well with the single beach analytical solutions must still be tested with the composite beach geometry, for which an analytical solution exists, with solitary waves as inputs. Most topographies of engineering interest can be approximated by piecewise–linear segments allowing the use of shallow water equation to determine approximate analytical results for the wave runup in closed form. In principle, fairly complex bathymetries can be represented through a combination of positively/negatively sloping and constant-depth segments.

In this regard, solutions of the shallow water equation at each segment can be matched analytically at the transition points between the segments, and then the overall amplification factor and reflected waves can be determined, analytically.

The definition sketch for revere beach is shown in Figure(S38) which is not to scale. It can be seen that transition points and segments are numbered from shoreward to seaward, i.e., segment numbers 1, 2, 3, and 4 show 1/13, 1/150, 1/53, and a constant-depth segment.

The defined parameters in figure(S38) are, H: the incident solitary wave height, x_1: represents the initial position of the wave-crest, x_0: is the starting point of the sloping beach, d: is the constant bathymetry depth, and g: the gravitational constant, and R: the run-up.

In this particular benchmark case, time series at different locations i.e., $x = 12.64$ $[m]$, $x = 15.04$ $[m]$, $x = 17.22$ $[m]$, $x = 19.40$ $[m]$, $x = 20.86$ $[m]$, $x = 22.33[m]$, and $x = 22.80[m]$ were provided in order to test the model. It is important to mention that a solid wall was provided at $x = 23.23$ $[m]$, in such case a total reflection is expected to occurs at this level.

S7.4.3 Conical island wetting–drying benchmark

To evaluate `SWEpy`'s wet/dry handling and reconstruction detail during wave-obstacle interactions—essential for coastal inundation modeling—we simulate the conical island benchmark, a laboratory experiment by [5] recommended for SWE validation in [42]. This test assesses (i) positivity preservation on sloped topography, where runup/rundown induces dynamic wetting without unphysical negatives, and (ii) compares operator fidelity in capturing complex wavefronts. The domain is a 41 × 30 m tank with permeable (soft) boundaries to absorb reflections, minimizing boundary artifacts. Discretization uses an equilateral triangular grid with edge length 0.5 m (12,000 cells) for isotropy. Bathymetry features

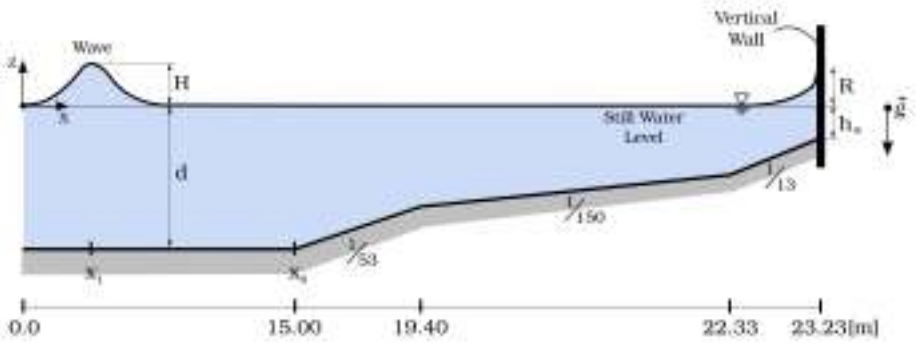


Fig. S38: Definition sketch of solitary on a composite beach

a flat bottom with a truncated cone (toe diameter 7.2 m, crest 2.2 m, height 0.625 m) centered at the origin, simulating an island. The initial free-surface elevation is given by the solitary-wave profile:

$$w(x, y, 0) = \frac{H}{d} \operatorname{sech}^2\big(\gamma\,(x - X_1)\big), \tag{S47}$$

with $d = 0.320$ m, $H = 0.02976$ m, and $\gamma = \sqrt{3H/4d^3}$, positioned at $X_1 = -13$ m. The initial velocity field is

$$u(x, y, 0) = \frac{g}{d} w(x, y, 0) \left(1 - 0.25 \frac{w(x, y, 0)}{d}\right), \quad v(x, y, 0) = 0, \tag{S48}$$

Derived to ensure consistent propagation, and extruded uniformly in the y-direction to simulate a two-dimensional wave front. Simulations are conducted with constant, minmod, and WENO reconstruction operators for cross-comparison, employing a forward Euler time integration scheme with a Courant-Friedrichs-Lewy (CFL) number of 0.25. This choice isolates spatial-reconstruction effects, allowing a focused assessment of how each operator handles wet/dry transitions and wavefront modelling as the soliton interacts with the cone, as visually depicted in the initial setup of Figure S39.

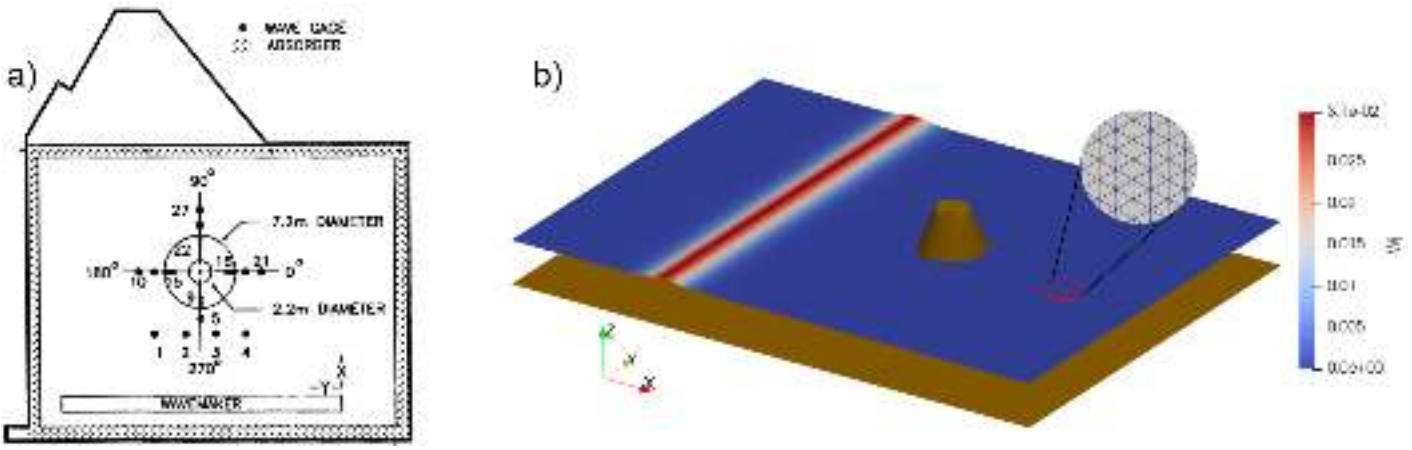


Fig. S39: Configuration of original experiment (digitized from [5]) for the conical island benchmark, illustrating the solitary wave profile approaching the truncated cone.

Figure S40 shows the free-surface elevation for solutions using each reconstruction operator at $t = 7$ s, when the leading wave is passing the location of gauge 16. This snapshot provides a spatial context for the subsequent time-series comparison, highlighting differences in wavefront sharpness and height at the gauge location. The WENO reconstruction exhibits the sharpest and highest wave at gauge 16 ($\eta \approx 0.041$ m), the minmod reconstruction shows moderate attenuation ($\eta \approx 0.03$ m), and the constant reconstruction produces a visibly diffused wavefront ($\eta \approx 0.011$ m). Green markers indicate the positions of gauges 1, 6, 16, and 22 (respectively from left to right), aligning with the experimental setup for direct validation of runup dynamics.

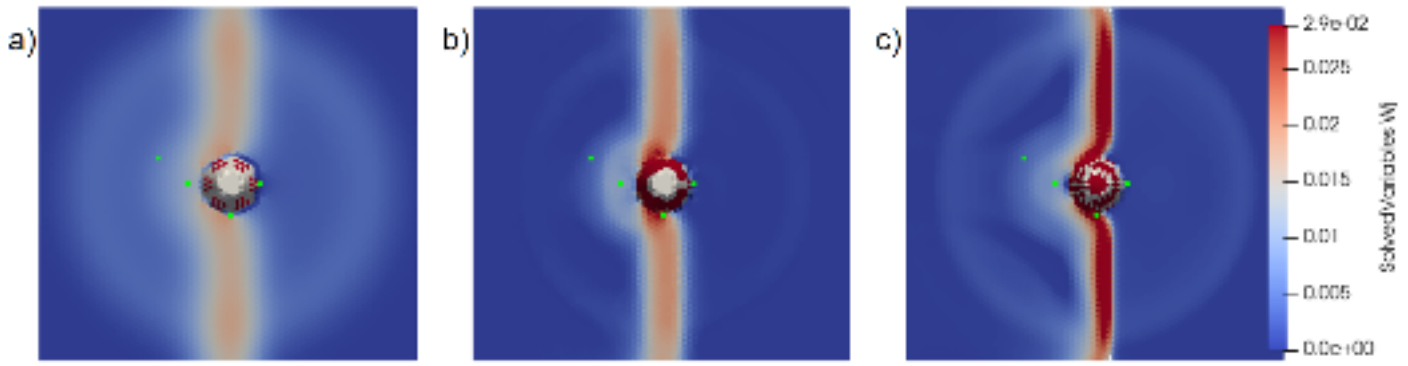


Fig. S40: Comparison of wavefront measurement at gauge 16. Free-surface elevation at $t = 7$ s for (a) constant, (b) minmod, and (c) WENO reconstruction operators. Green markers indicate the positions of gauges 1, 6, 16, and 22 (respectively from left to right).

Figure S41 presents time-series results for gauges 1, 6, 16, and 22 using the constant, minmod, and WENO reconstruction operators. The laboratory records, originally timed from wavemaker initiation, have been shifted by -25 s to align incident wave arrival with simulations, corresponding to the estimated paddle deceleration inferred from signal traces in [5].

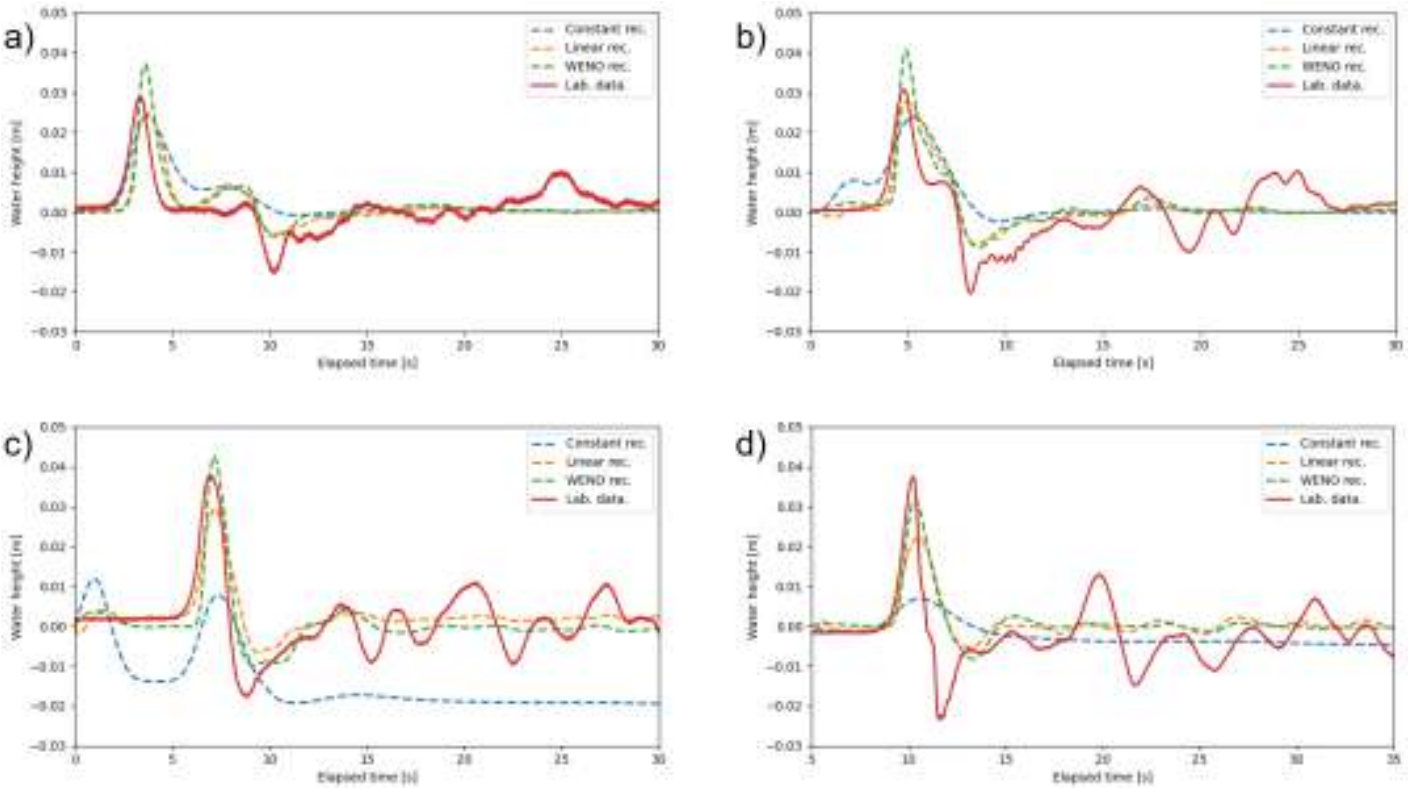


Fig. S41: Time series of water heights at gauges 1 (a), 6 (b), 16 (c), and 22 (d). Solid line is laboratory data, and dashed lines are `SWEpy`'s solutions.

Across gauges, WENO reproduces main crest height and arrival with minimal phase error, while constant and minmod underestimate secondary oscillations and show greater dissipation during rundown. Quantitatively, the L^2 norm of differences between numerical and experimental series (Table S3) confirms that WENO and minmod exhibit similar performance, with WENO showing slightly higher averaged errors over the four gauges ($\overline{|e|}_2^{\text{WENO}} = 0.0321$ vs. $\overline{|e|}_2^{\text{minmod}} = 0.0320$), while both outperform constant by approximately 40% ($\overline{|e|}_2^{\text{constant}} = 0.0533$).

Rec.	Gauge 1	Gauge 6	Gauge 16	Gauge 22
Constant	0.0302	0.0373	0.1014	0.0443
Minmod	0.0254	0.0288	0.0325	0.0412
WENO	0.0262	0.0296	0.0316	0.0409

Tab. S3: L^2 error of time series for each reconstructor at the different gauges.

Since L^2 is phase-sensitive and the shift is estimated, we repeated analysis with uniform shifts from -0.10 to $+0.30$ s on numerical series (Table S4), confirming WENO's robustness (lowest scores across shifts), validating its capacity for dynamic wet/dry reconstruction in coastal flows.

Rec.	+0.30	+0.25	+0.20	+0.15	+0.10	+0.05	-0.05	-0.10
Constant	0.0516	0.0518	0.0520	0.0523	0.0526	0.0529	0.0537	0.0540
Minmod	0.0285	0.0288	0.0292	0.0298	0.0304	0.0312	0.0329	0.0338
WENO	0.0274	0.0277	0.0282	0.0289	0.0298	0.0309	0.0334	0.0348

Tab. S4: L^2 average error for each reconstructor with different time shifts of laboratory data to account for phase error.

Figure S42 illustrates spatial diffusion effects by comparing free-surface elevation at $t = 13$ s across the three reconstruction operators, a stage where the wave has traversed the island and formed the cardioid-shaped crest pattern observed in laboratory measurements. The WENO reconstruction (panel c) preserves sharp gradients, with minimal attenuation and retaining intricate structures. By contrast, the minmod reconstruction (panel b) maintains the overall pattern but introduces noticeable smoothing, while the constant reconstruction (panel a) dissipates most fine-scale features, resulting in a blurred profile. To quantify these differences, we computed the depth-integrated potential energy $\mathcal{E} = \rho g \int_{\Omega_{\text{strip}}} \eta^2, d\Omega$ over a vertical strip from $x_l = 9.5$ m to $x_r = 14.5$ m downstream, where $\eta = w$ given the zero still-water level in the solitary-wave setup. The energies are $\mathcal{E}_{\text{const}} = 0.015$, $\mathcal{E}_{\text{minmod}} = 0.020$, and $\mathcal{E}_{\text{WENO}} = 0.044$, confirming WENO's superior wave energy retention in the post-interaction field. Combined with the L^2 analysis, these findings underscore WENO's optimal balance between low numerical diffusion and faithful waveform reproduction, whereas constant reconstruction underperforms in both aspects.

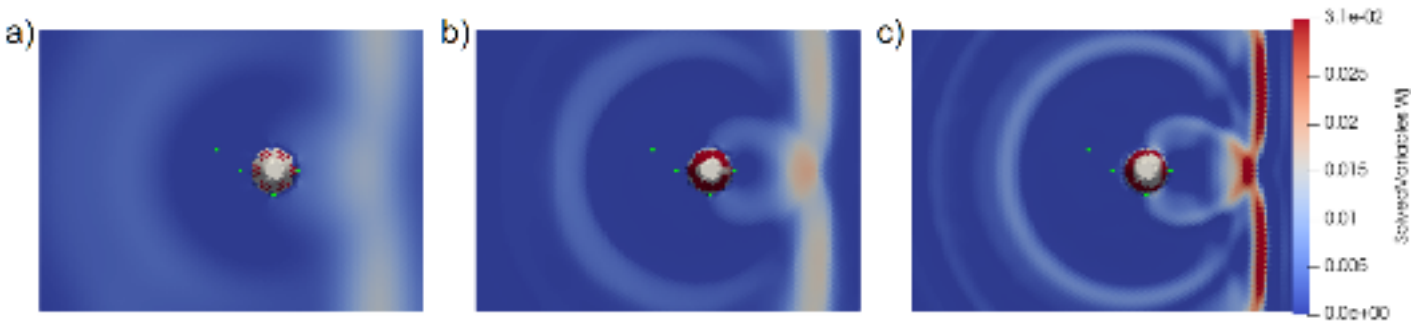


Fig. S42: Diffusion study. Solution at time $t = 13$ s for constant (a), linear (b), and quadratic (c) renconstructors. Green marks are the locations of gauges 1, 6, 16, and 22 (from left to right).

From the previous comments, it can be appreciated that point measurements reflect only one aspect of the numerical result, which can be helpful for specific events such as wave front arrival time, zones of maximum amplitudes, or inundations, among others. In addition, Figure S42 shows a plan view to emphasize the importance of a global approach to the numerical results and their relation to the physical problem analyzed. In that direction, and drawing on the conclusion that the WENO-based approach captures relevant information for global analysis, Figure S43 provides a detailed sequence of the wavefront evolution under WENO reconstruction, illustrating how it maintains sharp gradients and structural integrity throughout the interaction at $t = 7$, 8, and 9 s. The leading crest propagates toward the island, splits, and wraps around its flanks, producing a clear diffraction pattern. In the lee, the opposing wavefronts converge and form a coherent cusp that advances shoreward. A small, nearly circular secondary wave is visible behind the main crest; this is a residual artifact of the wetting–drying correction process, but it decays rapidly and does not trigger further spurious oscillations. For the whole simulation, the shoreline evolves smoothly, and the wavefront retains sharp gradients through the interaction, even in the presence of zeroth-order boundary conditions. Thus, the wavefront sequence in Figure S43 illustrates that `SWEpy`'s implementations incorporate the appropriate numerical foundations, enabling detailed descriptions of more complex real-world phenomena.

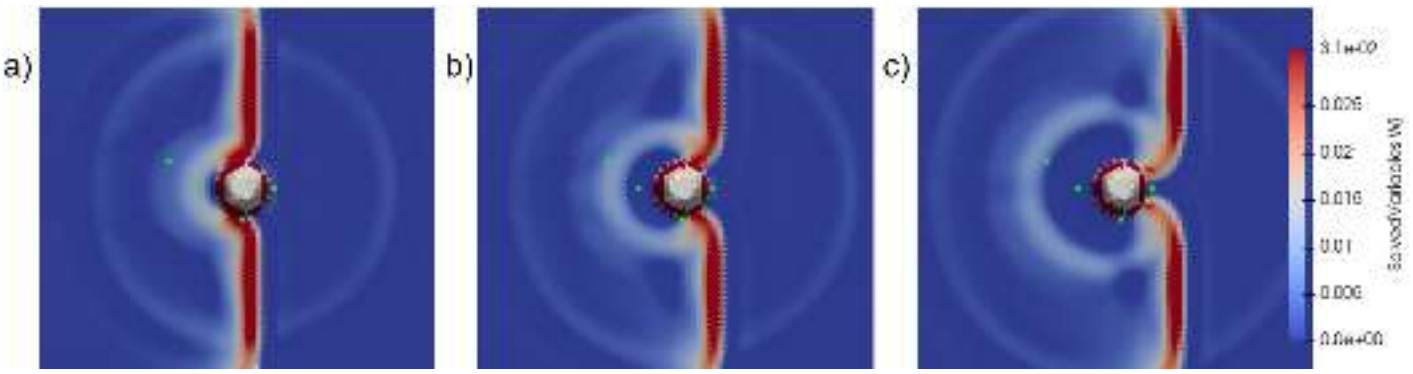


Fig. S43: Wave-obstacle interaction study. Solution at times $t = 7$ s (a), $t = 8$ s (b), and $t = 9$ s (c), using the WENO reconstruction operator. Green marks are the locations of gauges 1, 6, 16, and 22 (from left to right).

Overall, the conical-island benchmark confirms that `SWEpy` reproduces the key hydrodynamic processes involved in wave–obstacle interaction, including runup and rundown on sloping topography, diffraction around an emergent feature, and convergence in the lee. The comparison with laboratory measurements demonstrates that the WENO reconstruction consistently achieves the most faithful representation of height amplitude, phase, and post-interaction structure, while maintaining stability at wetting–drying fronts, ensuring positivity preservation. Although small discrepancies remain–particularly in negative water levels following the first rundown–these can be attributed to physical processes not represented in the depth-averaged SWE framework (e.g., vertical accelerations) and to uncertainties in aligning the laboratory and numerical time series. Taken together with the other validation cases, these results highlight the model's capability to resolve complex nearshore hydrodynamics with high numerical fidelity, especially when paired with high-order reconstruction.

S7.5 Real-World Benchmarks

S7.5.1 Malpasset Dam failure

To evaluate `SWEpy`'s performance in realistic inundation scenarios involving complex topography and moving wet/dry fronts, we reproduce the 1959 Malpasset dam failure on France's Reyran River. This benchmark event is characterized by rapid flooding over highly irregular terrain Moulinec2011. The case serves to validate the model's positivity-preserving reconstruction schemes, treatment of bathymetric source terms, and semi-implicit friction formulation described in Section **??**.

The computational domain is discretized using an unstructured triangular grid adapted from the TELEMAC-2D validation dataset. The mesh contains 26,000 elements, with characteristic triangle heights (measured from a vertex to the opposing side) ranging from $\Delta r_{\min} = 4.01$ m to $\Delta r_{\max} = 401.95$ m, and an average value of $\Delta r_{\text{avg}} = 40.28$ m.

The bathymetric and topographic data are derived from the 1931 IGN Saint-Tropez map, with additional refinement upstream of the dam to better resolve steep gradients (Figure S44a, inset). The initial condition sets the reservoir water surface to an elevation of 100 m upstream of the dam, represented as a vertical plane between coordinates (4701.18, 4143.10) and (4655.50, 4392.10), and 0 m elsewhere, with all cells above sea level initialized as dry (Figure S44b). Boundary conditions are specified as impermeable walls. The Manning roughness coefficient is set to $n = 0.03$ to match the TELEMAC configuration. The simulation employs minmod reconstruction and adaptive time stepping with a Courant–Friedrichs–Lewy (CFL) number of 0.33, and is run until $t_{\max} = 4000$ s.

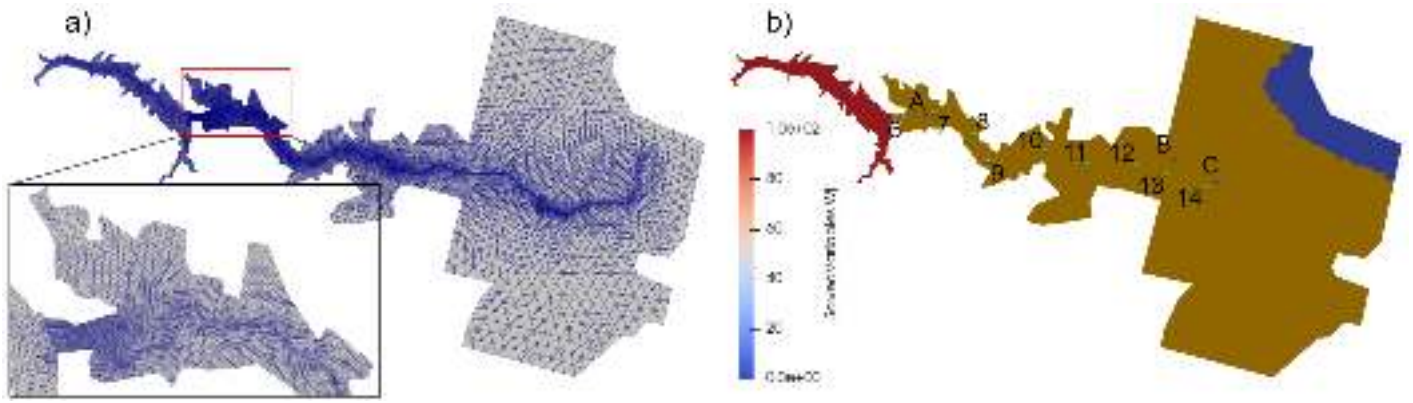


Fig. S44: Grid used in the simulation (a) with zoomed view of the upstream refined part, and initial water height (b). Points are the locations of measured data.

Twelve virtual gauge locations are defined along the river valley to monitor the flood wave progression (Figure S44b). Three gauges (transformers

A, B, and C) are used to measure wave arrival times, while nine gauges (P6–P14) record maximum water heights from 1:400 scale laboratory experiments by Electricité de France. The simulated values of arrival times and peak heights are reported in Table S5 alongside the corresponding experimental data and results from TELEMAC's HLLC solver, which is regarded as the most accurate scheme for this benchmark.

Point	Recorded Data		SWEpy			TELEMAC		
	$h_{\max}$ [m]	t_{ArrA} [s]	$h_{\max}$ [m]	t_{ArrA} [s]	Err [%]	$h_{\max}$ [m]	t_{ArrA} [s]	Err [%]
A	-	-	-	-	-	-	-	-
B	-	1140	-	1071	-6	-	1142.9	0
C	-	1320	-	1088	-18	-	1387.3	5
P6	40.3	-	37.72	-	-6	81.58	-	102
P7	14.6	-	18.13	-	24	55.88	-	283
P8	24.0	-	22.46	-	-6	53.21	-	122
P9	12.8	-	18.6	-	45	48.14	-	276
P10	11.8	-	15.34	-	30	36.88	-	213
P11	8.3	-	6.21	-	-25	25.41	-	206
P12	10.1	-	5.89	-	-42	19.29	-	91
P13	6.8	-	12.21	-	80	17.74	-	161
P14	5.4	-	4.32	-	-20	12.71	-	135

Tab. S5: Simulation results and relative errors, evaluated against recorded data, for both TELEMAC and SWEpy

This benchmark provides a rigorous test of SWEpy's ability to handle complex bathymetry, apply semi-implicit friction for roughness effects, and accurately treat wetting–drying fronts in initially dry cells. While SWEpy exhibits notable discrepancies at certain locations, its relative errors are, in most cases, nearly an order of magnitude smaller than those produced by TELEMAC's finite volume solver. As noted in TELEMAC's validation guide, these discrepancies may stem from several factors: (i) measurement uncertainty in the 1:400 scale physical model, (ii) omission of debris transport and sediment dynamics, and (iii) the likelihood that the dam breach was not truly instantaneous. In addition, the combination of rapidly varying topography and highly curved flow paths may violate the underlying assumptions of the shallow-water equations, further contributing to deviations between simulated and observed results.
TELEMAC's accuracy improves with increasing distance from the dam, suggesting that numerical diffusion is a significant factor in the model and may help explain its closer agreement in arrival-time estimates. Point P13 stands out as a pronounced outlier for both solvers. Its location within a poorly resolved section of the inner riverbank likely contributes to the large discrepancy in the predicted maximum height. To support this hypothesis, nearby points located outside the riverbank record simulated water heights between 4 m and 8 m, values that align more closely with the observed data.
Given the complex bathymetry and the tightly spaced unstructured grid, some numerical artifacts arise in our solutions from the interaction between the wet/dry treatment and the impermeable-wall ghost-cell boundary conditions. These occur in border cells where the bathymetry normal points outward from the domain, i.e., locations where water would naturally exit the computational area. Such cells can act as an artificial source of inflow, as observed in the animations provided in the supplement. In the present case, these artifacts do not significantly affect the solutions presented. However, careful grid construction can help prevent such situations. Figure S45 illustrates the original and corrected bathymetry near domain boundaries, showing a noticeable reduction in spurious water influx after applying the correction.

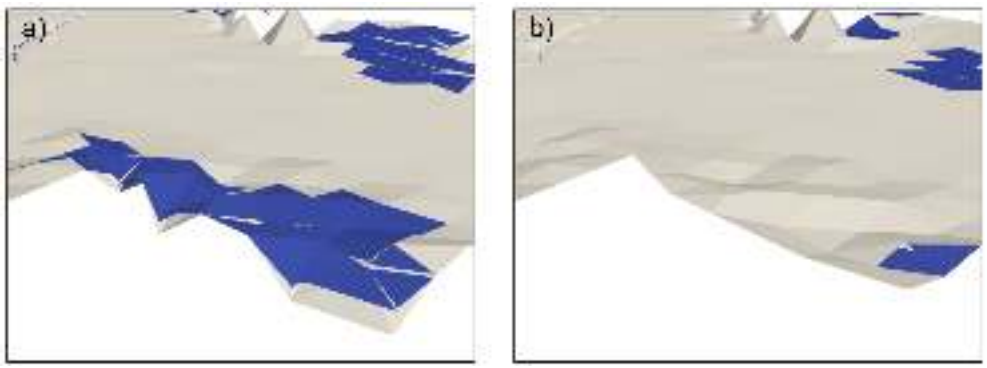


Fig. S45: 3D View of solution before (a) and after (b) grid correction. Artificial water influx is reduced thanks to the bathymetry fix.

However, some artificial inflows remain even after applying this correction procedure. Figure S46 shows the wave arrival at transformers A and C, where the inundation pattern is correctly reproduced but small residual mass contributions from these artifacts are still visible in some cells.

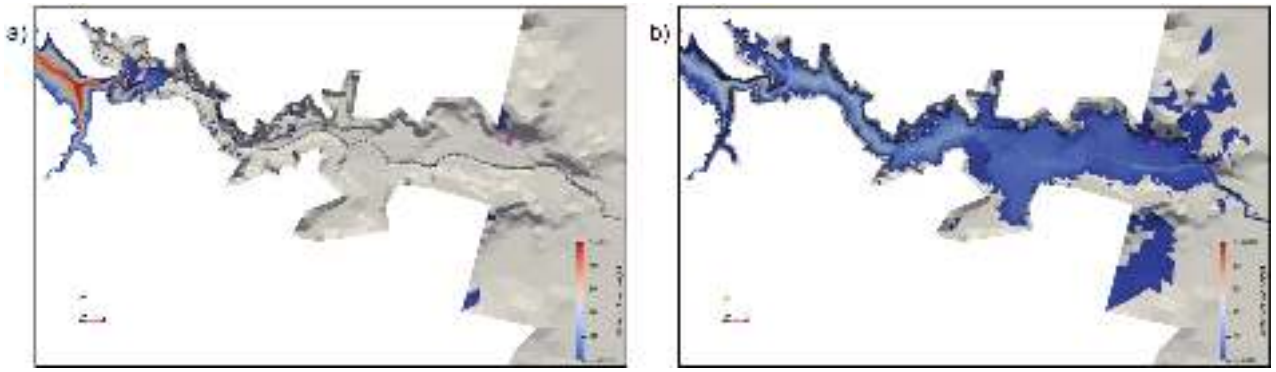


Fig. S46: Top-down view of inundation wave arriving at transformer A (a) and C (b). Transformer locations are marked with magenta boxes. Some water influx is present due to errors of the border-wet/dry interaction.

These results confirm that `SWEpy` can reliably reproduce complex inundation dynamics over irregular terrain while also identifying clear avenues for improvement. Zones with limited topographic resolution would benefit from targeted mesh refinement, and the integration of additional physical processes, like rheology, sediment transport, and the influence of steep bathymetric gradients or strong flow curvature, could further enhance predictive skill. The interaction between ghost-cell boundaries and the wet/dry treatment, identified here as a source of spurious inflow, will be studied in more detail in [17].

S7.5.2 Maule 2010 tsunami

To evaluate `SWEpy`'s capability for far-field tsunami simulation–specifically its ability to propagate long-period waves over transoceanic distances with minimal numerical dissipation–we reproduce the 2010 Maule tsunami, generated by an M_w 8.8 earthquake along the Chilean subduction zone [3]. This event produced measurable signals across the Pacific basin, where Coriolis effects are dynamically relevant and numerical accuracy over very long propagation paths is essential for preserving wave amplitude and shape. This test also enables assessment of the combined impact of WENO spatial reconstruction and SSP RK4,3 time integration, in comparison with minmod and FE methods. The computational mesh is an equilateral triangular grid generated from GEBCO bathymetry [18] via a spherical–Cartesian transformation, containing approximately 10^6 cells (∼536 000 nodes). Because the mesh was constrained to a rectangular bounding box for refinement, additional cells were created at the corners; the effective number of wet cells representing the domain is therefore about 860 000. The initial sea-surface displacement is prescribed from the inversion by [3] using the Okada fault-slip model, with zero initial velocity. Coriolis effects are included by setting $f = 10^{-4}$ s^{-1}. Simulations span 24 h of physical time, using adaptive time stepping with a Courant–Friedrichs–Lewy (CFL) number of 0.25 for FE integration and 0.5 for RK4,3 integration, applied to each reconstruction–integrator configuration.
Two virtual wave gauges are positioned at the locations of NOAA DART buoys 32412 (southwest of Lima, Peru) and 21413 (southeast of Kyoto, Japan), as indicated in figure S47. At each gauge, the water-column height is recorded every 60 s of simulated time, yielding continuous time series for the 24 h simulation period. This duration captures the primary tsunami signal and subsequent wave groups at both stations. The numerical results are compared with quality-controlled, rectified DART measurements noaa, enabling a direct evaluation of amplitude and phase preservation over basin-scale propagation.

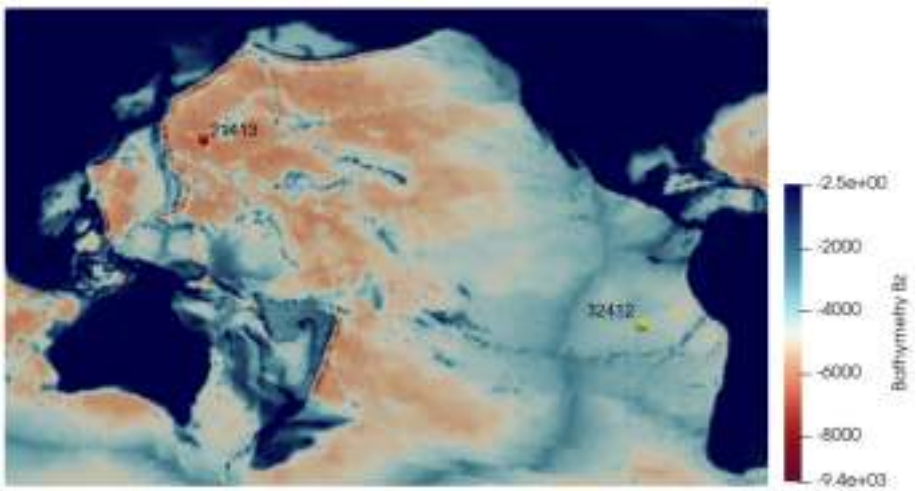


Fig. S47: Bathymetry grid employed for the Maule tsunami simulation.

Figure S48 compares the simulated tsunami waveforms at DART buoys 32412 (panel a) and 21413 (panel b) with quality-controlled observations and reference simulations from the HySEA model. At buoy 32412, the WENO reconstruction with FE integration provides the closest match to the observed primary wave amplitude and timing, and retains secondary oscillations more effectively than the minmod and constant reconstructions. Minmod with a high limiter parameter ($\vartheta = 1.4$) reduces phase drift relative to the constant scheme, but still underestimates the amplitude of later wave groups. At buoy 21413, located in the northwestern Pacific, all SWEpy configurations exhibit greater attenuation of the signal, reflecting the cumulative impact of propagation distance and coarse resolution; here, the WENO scheme again preserves amplitude better than the other reconstructions, with the minmood operator becoming too oscillatory, although the differences are less pronounced. Across both sites, HySEA results at 1.5 M cells show closer agreement with the DART data than `SWEpy`, consistent with the benefits of higher effective resolution. While WENO incurs a modest computational cost increase over minmod (+14 minutes), it remains faster than real time for the domain and resolution tested, and offers a consistent improvement in waveform fidelity.

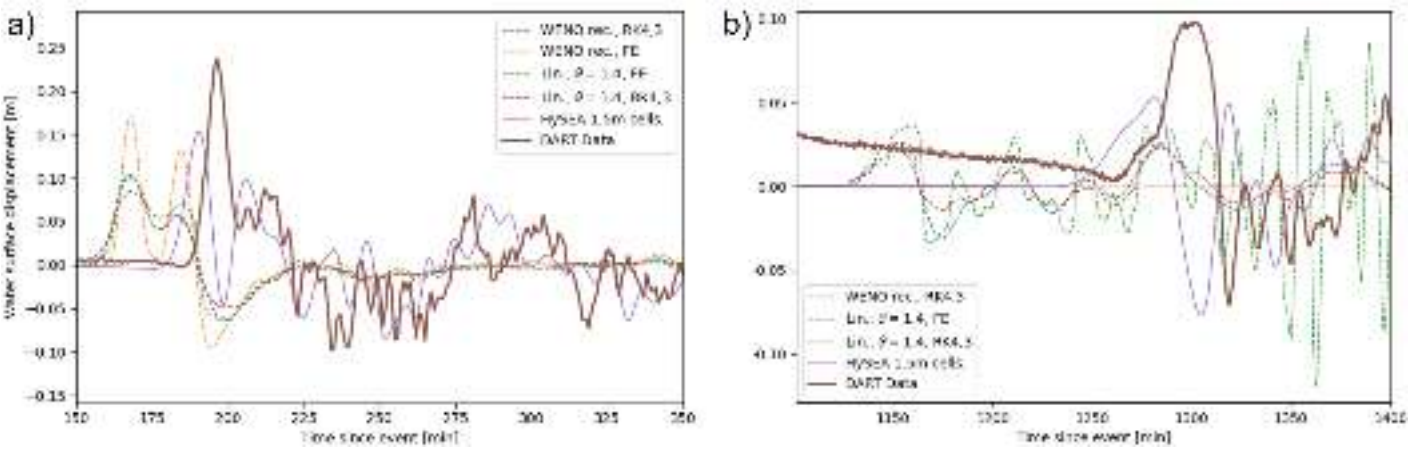


Fig. S48: Tsunami profile comparison at DART buoy 32412 (a) and 21413 (b). Results from the HySEA model are included for reference. The FE+WENO result for buoy 21413 is omitted because of high-frequency oscillations that obscured the primary tsunami signal.

The FE+WENO result is not shown for buoy 21413 because the combination of forward Euler time-stepping and coarse resolution over the long propagation path generated high-frequency oscillations that obscured the primary tsunami signal (figure S49). Table S6 quantifies the performance of each configuration by comparing the maximum wave height $H_{\max}$ and arrival time T_{arr} of the first wave against the DART observations. These metrics complement the full time-series comparison by highlighting differences in amplitude attenuation and phase shift.

Tab. S6: Performance summary of `SWEpy` configurations and HySEA for the Maule 2010 tsunami benchmark. Maximum surface displacement $H_{\max}$ [m] and arrival time T_{arr} [min] are extracted at the first wave crest for DART buoys 32412 (Lima) and 21413 (Kyoto). Relative errors (%) are computed with respect to the DART observations.

Method	Lima				Kyoto			
	$H_{\max}$	Err%	T_{arr}	Err%	$H_{\max}$	Err%	T_{arr}	Err%
DART Data	0.237	-	196	-	0.098	-	1296	-
WENO+RK4,3	0.1034	-56.4	167.263	-14.7	0.022	-77.6	1151.39	-11.2
WENO+FE	0.1703	-28.1	167.098	-14.7	-	-	-	-
minmod+RK4,3	0.0851	-64.1	167.197	-14.7	0.02551	-74.0	1152	-11.1
minmod+FE	0.1005	-57.6	169.068	-13.7	0.03611	-63.2	1158.1	-10.7
HySEA	0.1537	-35.1	190.064	-3.02	0.0534	-45.5	1281	-1.2

These metrics indicate that `SWEpy` underestimates the maximum crest height and predicts earlier arrivals at both stations, whereas HySEA exhibits smaller amplitude bias—particularly at Lima—and reduced phase error, consistent with the visual comparisons in figure S48. The differences likely reflect a combination of: (i) source smoothing introduced during interpolation to the computational grid, (ii) bathymetric resolution, with the HySEA configuration employing nearly twice as many cells, and (iii) projection-related errors from the spherical–Cartesian transformation, given that HySEA solves the shallow-water equations directly on the sphere. This last point is important, since we only converted the grid to cartesian coordinates via a haversine transformation; however, formulas for fluxes, cell side length, cell area, and such were maintained, when they should be calculated considering transformations. Quantifying the contribution of each factor is left for future work, with the aim of guiding targeted improvements to `SWEpy`'s far-field performance by a rigorous treatment of the spherical case.
While initially producing the most accurate waveforms at the near-field gauge, the FE time integrator—particularly in combination with the WENO reconstruction—develops spurious high-frequency oscillations after extended transoceanic propagation. These oscillations overwhelm

the primary tsunami signal at the Kyoto gauge, making the solution unsuitable for quantitative analysis. Figure S49 illustrates the modeled free-surface elevation at the moments when the wave passes the Lima and Kyoto DART buoys, for both FE+WENO and RK4,3+WENO configurations, highlighting the marked difference in solution smoothness.

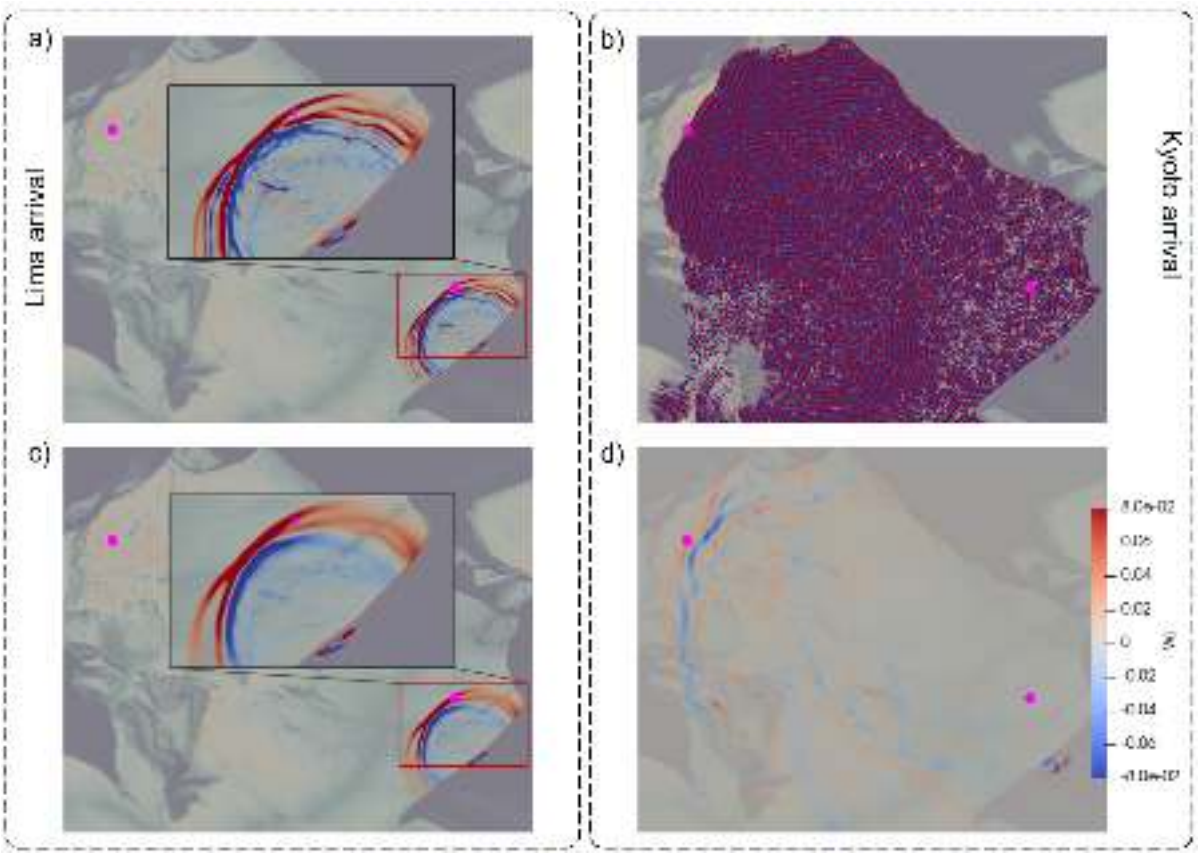


Fig. S49: Snapshots of simulated free-surface elevation for the Maule 2010 tsunami at the times when the leading crest passes the locations of DART buoys 32412 (Lima) and 21413 (Kyoto). Insets show zoomed view of wave at Lima buoy. Panels (a) and (b) correspond to forward Euler (FE) integration, and panels (c) and (d) to SSP RK4,3 integration. Buoy locations are marked with magenta circles. These views provide basin-scale context for the time-series comparisons in figure S48.

Control of these oscillations could, in principle, be achieved by reducing the CFL number; however, this would further increase numerical diffusion. Even with a reduced value of $CFL = 0.15$, the oscillations remain excessive by the time the wave reaches Kyoto, producing spurious amplitudes of approximately 0.3 m, while simultaneously reducing the maximum height at Lima to 0.14 m.

To evaluate computational efficiency and the benefits of GPU acceleration, the Maule 2010 scenario was repeated on a simplified, non linearized, mesh of approximately 2.5×10^5 elements, using different combinations of time integrators and spatial reconstructors. At this reduced resolution, wave-height errors ranged from -98% to -13.4% and arrival-time errors from -2% to 26.5% relative to DART observations, reflecting the expected degradation in solution quality. Despite this, execution times for all configurations were faster than real time (table S7).

Tab. S7: Comparison of methods minmod+FE (fastest) and WENO+RK4,3 (slowest)

Method	CPU (min)	GPU (min)	Speedup (×)
minmod+FE	37.32	2.87	13.0
WENO+RK4,3	360.49	17.31	20.8

Tests were performed on an Intel® Core™ i5-10300H CPU (10th generation) and an NVIDIA GeForce GTX 1650 GPU, both consumer-grade components released more than six years prior to writing. These results underscore `SWEpy`'s ability to deliver high-performance simulations on widely available, non-specialized hardware.

As a whole, the Maule 2010 benchmark results demonstrate `SWEpy`'s ability to reproduce the main features of basin-scale tsunami propagation, including crest arrival timing, amplitude decay, and the modulation of subsequent wave groups. Among the tested schemes, the WENO

reconstruction consistently yields the most accurate far-field waveforms, though in combination with forward Euler integration it can develop basin-scale oscillations that obscure the signal at distant stations (figure S49). These oscillations persist even under reduced CFL numbers, with the trade-off of increased diffusion and degraded amplitudes at nearer gauges. The SSP RK4,3 integrator mitigates this instability while preserving much of WENO's accuracy, making it the most balanced choice for long-range simulations. GPU acceleration enables faster-than-real-time performance even for the most demanding configuration, with speedups exceeding 20× on consumer-grade hardware with respect to serial implementations. Together, these results confirm the model's applicability to large-domain tsunami scenarios, while highlighting clear paths for improvement in projection accuracy, bathymetric resolution, and source initialization to close the remaining gaps with higher-resolution reference models such as HySEA.

S8 Glossary

The following list of variables are extensively used in `ANSWER`:

Variable	:	Description	Dimension
	:		
g	:	Gravity constant	[1,1]
n	:	Manning's roughness coefficient	[1,1]
Tol	:	Cut-off value for dry cell	[1,1]
t_{max}	:	Total simulation time	[1,1]
dt	:	Maximum allowed time step	[1,1]
nElems	:	Total number of elements	[1,1]
nNodes	:	Total number of nodes	[1,1]
nGhostCells	:	Total number of ghost cells	[1,1]
nThreads	:	Number of threads to be used	[1,1]
nBlocks	:	Number of blocks to be used	[1,1]
Elems	:	Conectivity array of elements.	[nElems,3]
Neigh	:	Array of neighbor elements	[nElems,3]
Sides	:	Array of side elements	[nElems,3]
GhostCells	:	Connectivity array of ghost cell elements	[GhostCells,3]
ElemCoord	:	Coordinates of each node (x,y)	[nNodes,2]
x_j	:	Cell-center vector coordinates in X	[nElems,1]
y_j	:	Cell-center vector coordinates in Y	[nElems,2]
T_j	:	Area of the j^{th} triangle	[N_e,1]
B_j	:	Cell-centered bathymetry values	[N_e,1]
W_j	:	Cell-centered water level for j^{th} triangle	[N_e,1]
HU_j	:	Cell-centered Flux over X for j^{th} triangle	[N_e,1]
HV_j	:	Cell-centered Flux over Y for j^{th} triangle	[N_e,1]
S_x	:	Source term along-X	[N_e,1]
S_y	:	Source term along-Y	[N_e,1]
F_x	:	Friction term along-X	[N_e,1]
F_y	:	Friction term along-Y	[N_e,1]
n_x	:	Unit-Normal component over X	[N_e,3]
n_y	:	Unit-Normal component over Y	[N_e,3]
l_{jk}	:	Side lengths for each triangle	[N_e,3]
r_{jk}	:	Side altitudes for each triangle	[N_e,3]
W_{mj}	:	Mid-points values of water surface	[N_e,3]
HU_{mj}	:	Mid-points values of flux along-X	[N_e,3]
HV_{mj}	:	Mid-points values of flux along-Y	[N_e,3]
h_{mj}	:	Mid-point values of water depth	[N_e,3]
u_{mj}	:	Normal Velocity over X-direction at mid-pints.	[N_e,3]
v_{mj}	:	Normal Velocity over Y-direction at mid-pints.	[N_e,3]
a_{in}	:	One-sided inwards velocity	[N_e,3]
a_{out}	:	One-sided outwards velocity	[N_e,3]
U_{in}	:	Inward mid-point values of vector variables	[N_e,3]
U_{out}	:	Outwards mid-point values of vector variables	[N_e,3]
FU_{in}	:	Inward mid-point values of Flux vector in X	[N_e,3]
FU_{out}	:	Outwards mid-point values of Flux vector in X	[N_e,3]
GU_{in}	:	Inward mid-point values of Flux vector in Y	[N_e,3]
GU_{out}	:	Outwards mid-point values of Flux vector in Y	[N_e,3]

Variable	:	Description	Dimension
	:		
Movie_dt	:	Time step sampling to create the movie	[1,1]
Movie_nt	:	Number of total time steps for movie	[1,1]
SaveArrivalTimes	:	Boolean variable to specify if arrival time will be store	[1,1]
SaveMaxWaterHeights	:	Boolean variable to specify if maximum water heigh time will be store	[1,1]
SaveTSeries	:	Boolean variable to specify if time series values will be store	[1,1]
SaveSProfiles	:	Boolean variable to specify if spatial profile values will be store	[1,1]
ArrivalTimes	:	Vector variable that stores the arrival time	[1,1]
MaxWaterHeights	:	Vector variable that stores the maximum water heigh	[1,1]
TSeries_values	:	Vector variable that stores the time series values	[1,1]
SProfiles_values	:	Vector variable that stores the spatial profile series values	[1,1]

References

[1] Arminjon, P., and St-Cyr, A. Nessyahu–tadmor-type central finite volume methods without predictor for 3d cartesian and unstructured tetrahedral grids. *Applied Numerical Mathematics 46*, 2 (2003), 135–155.

[2] Ayachit, U. *The paraview guide (full color version)*. Kitware, Jan. 2015.

[3] Benavente, R., and Cummins, P. R. Simple and reliable finite fault solutions for large earthquakes using the w-phase: The maule (mw = 8.8) and tohoku (mw = 9.0) earthquakes. *Geophysical Research Letters 40*, 14 (July 2013), 3591–3595.

[4] Bollermann, A., Chen, G., Kurganov, A., and Noelle, S. A well-balanced reconstruction of wet/dry fronts for the shallow water equations. *Journal of Scientific Computing 56*, 2 (August 2013), 267–290.

[5] Briggs, M. J., Synolakis, C. E., Harkins, G. S., and Green, D. R. Benchmark problem 2: Run-up of solitary waves on a circular island. In *International Workshop on Long Wave Modeling of Tsunami Runup* (Vicksburg, MS, 1995).

[6] Bryson, S., Epshteyn, Y., Kurganov, A., and Petrova, G. Well-balanced positivity preserving central-upwind scheme on triangular grids for the saint-venant system. *ESAIM: Mathematical Modelling and Numerical Analysis 45*, 3 (May 2011), 423–446.

[7] Bryson, S., Epshteyn, Y., Kurganov, A., and Petrova, G. Well-balanced positivity preserving central-upwind scheme on triangular grids for the Saint-Venant system. *ESAIM: Mathematical Modelling and Numerical Analysis 45*, 3 (May 2011), 423–446. Number: 3 Publisher: EDP Sciences.

[8] Bryson, S., and Levy, D. Balanced central schemes for the shallow water equations on unstructured grids. *SIAM Journal on Scientific Computing 27*, 2 (2005), 532–552.

[9] Bryson, S., and Levy, D. Balanced central schemes for the shallow water equations on unstructured grids. *SIAM Journal on Scientific Computing 27*, 2 (2005), 532–552.

[10] Cao, Y., Kurganov, A., Liu, Y., and Zeitlin, V. Flux globalization-based well-balanced path-conservative central-upwind scheme for two-dimensional two-layer thermal rotating shallow water equations. *Journal of Computational Physics 515* (Oct. 2024), 113273.

[11] Chertock, A., Cui, S., Kurganov, A., and Wu, T. Well-balanced positivity preserving central-upwind scheme for the shallow water system with friction terms. *International Journal for Numerical Methods in Fluids 78*, 6 (2015), 355–383.

[12] Chertock, A., Dudzinski, M., Kurganov, A., and Lukáčová-Medvid'ová, M. Well-balanced schemes for the shallow water equations with coriolis forces. *Numer. Math. (Heidelb.) 138*, 4 (Apr. 2018), 939–973.

[13] Chow, V. *Applied hydrology*. McGraw-hill, 1971.

[14] Chow, V. T. *Open-Channel Hydraulics*. McGraw-Hill, New York, 1959, 2009.

[15] Christov, I., and Popov, B. New non-oscillatory central schemes on unstructured triangulations for hyperbolic systems of conservation laws. *Journal of Computational Physics 227*, 11 (2008), 5736–5757.

[16] Desveaux, V., and Masset, A. A fully well-balanced scheme for shallow water equations with Coriolis force. *Communications in Mathematical Sciences 20*, 7 (2022), 1875–1900.

[17] Fuenzalida A., J. A., Meneses, R., Meza, J., and Kusanovic, D. Adaptive local reconstruction for solving saint-venant equations in irregular domains. Manuscript in preparation, 2025.

[18] GEBCO Bathymetric Compilation Group 2024. The gebco_2024 grid - a continuous terrain model of the global oceans and land., 2024.

[19] Gottlieb, S., Shu, C.-W., and Ketcheson, D. *Strong Stability Preserving Runge-kutta And Multistep Time Discretizations*. World Scientific Publishing, Singapore, Singapore, May 2010.

[20] Gottlieb, S., Shu, C.-W., and Tadmor, E. Strong stability-preserving high-order time discretization methods. *SIAM Review 43*, 1 (2001), 89–112.

[21] Greenberg, J. M., and Leroux, A. Y. A well-balanced scheme for the numerical processing of source terms in hyperbolic equations. *SIAM Journal on Numerical Analysis 33*, 1 (1996), 1–16.

[22] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature 585*, 7825 (Sept. 2020), 357–362.

[23] Jawahar, P., and Kamath, H. A high-resolution procedure for euler and navier–stokes computations on unstructured grids. *Journal of Computational Physics 164*, 1 (2000), 165–203.

[24] Kundu, P., Cohen, I., and Dowling, D. *Fluid Mechanics*. Science Direct e-books. Elsevier Science, 2012.

[25] Kurganov, A., and Petrova, G. Central-upwind schemes on triangular grids for hyperbolic systems of conservation laws. *Numerical Methods for Partial Differential Equations 21*, 3 (2005), 536–552. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/num.20049.

[26] Kurganov, A., and Petrova, G. A Second-Order Well-Balanced Positivity Preserving Central-Upwind Scheme for the Saint-Venant System. *Communications in Mathematical Sciences 5*, 1 (2007), 133 – 160.

[27] LeVeque, R. J. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, Aug. 2002.

[28] Liu, X. A new well-balanced finite-volume scheme on unstructured triangular grids for two-dimensional two-layer shallow water flows with wet-dry fronts. *Journal of Computational Physics 438* (2021), 110380.

[29] Liu, X., Albright, J., Epshteyn, Y., and Kurganov, A. Well-balanced positivity preserving central-upwind scheme with a novel wet/dry reconstruction on triangular grids for the saint-venant system. *Journal of Computational Physics 374* (2018), 213–236.

[30] Moukalled, F., Mangani, L., and Darwish, M. The finite volume method. In *The finite volume method in computational fluid dynamics: An advanced introduction with OpenFOAM® and Matlab*. Springer, 2015, pp. 103–135.

[31] Mungov, G., DOC/NOAA/NWS/NDBC > National Data Buoy Center, National Weather Service, NOAA, U.S. Department Of Commerce, and DOC/NOAA/OAR/PMEL > Pacific Marine Environmental Laboratory, OAR, NOAA, U.S. Department Of Commerce. Deep-ocean assessment and reporting of tsunamis (dart(r)), 2005.

[32] Nessyahu, H., and Tadmor, E. Non-oscillatory central differencing for hyperbolic conservation laws. *Journal of computational physics 87*, 2 (1990), 408–463.

[33] Okuta, R., Unno, Y., Nishino, D., Hido, S., and Loomis, C. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)* (2017).

[34] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 12 (Dec. 1972), 1053–1058.

[35] Persson, P.-O., and Strang, G. A simple mesh generator in matlab. *SIAM Review 46*, 2 (June 2004), 329–345.

[36] Stiernström, V., Lundgren, L., Nazarov, M., and Mattsson, K. A residual-based artificial viscosity finite difference method for scalar conservation laws. *Journal of Computational Physics 430* (Apr. 2021), 110100.

[37] Stoker, J. J. *Water Waves: The mathematical theory with applications*. John Wiley & Sons, Inc., 270 Madison Ave, NY, 1992.

[38] Sunder, D., Vaghani, D., and Shukla, R. *Third-Order WENO Schemes on Unstructured Meshes*. 02 2021, pp. 215–223.

[39] Sweby, P. K. High resolution schemes using flux limiters for hyperbolic conservation laws. *SIAM journal on numerical analysis 21*, 5 (1984), 995–1011.

[40] Synolakis, C. The runup of solitary waves. *Journal of Fluid Mechanics 185* (1987), 523–545.

[41] Synolakis, C. E. The runup of solitary waves. *Journal of Fluid Mechanics 185* (1987), 523–545.

[42] Synolakis, C. E., Bernard, E. N., Titov, V. V., Kânoğlu, U., and González, F. I. Validation and verification of tsunami numerical models. *Pure and Applied Geophysics 165*, 11–12 (Dec. 2008), 2197–2228.

[43] Toro, E. F. *Shock-capturing methods for free-surface shallow flows*. John Wiley, Chichester ; New York, 2001.

[44] V. Titov, C. S. Modeling of breaking and nonbreaking long-wave evolution and runup using vtcs-2. *Journal of Waterway, Port, Coastal, and Ocean Engineering 121*, 6 (1995), 308–316.

[45] Van Leer, B. Towards the ultimate conservative difference scheme. *Journal of computational physics 135*, 2 (1997), 229–248.

[46] Xie, W.-X., Cai, L., Feng, J.-H., and Xu, W. Computations of shallow water equations with high-order central-upwind schemes on triangular meshes. *Applied mathematics and computation 170*, 1 (2005), 296–313.

[47] Zhu, J., and Qiu, J. New finite volume weighted essentially nonoscillatory schemes on triangular meshes. *SIAM Journal on Scientific Computing 40*, 2 (2018), A903–A928.