Programming Homework 04

❗ Submission Deadline: 17.06.2026 - 23:55

- The homework solution has to be handed in as a group solution via Moodle.
- The notebooks must run without errors in our environment on the RWTH-Cluster. See here for more information on how to get started.

❗ Participant list

Please add your names and student ID (Matrikelnummer) here:

1. Name, ID
2. Name, ID
3. Name, ID
4. Name, ID

1 Learning goals

- Understand the shallow water equations (SWE) and their extension to the discontinuous Galerkin (DG) method
- Understand how the SWE weak formulation is a natural extension of the scalar advection case from Homework 01
- Understand class-based design in Python: abstract base classes, inheritance, and method overriding
- Understand stiff source terms and why explicit friction treatment leads to instability
- Understand the lake-at-rest well-balancing property and its failure in a naive scheme
- Understand the Audusse hydrostatic reconstruction as a fix for well-balancing
- Perform our first debris flow simulation on real topography data

Before we start

1. Make sure to update the `library` folder in the `CMM` directory. Download link: library.tar.gz

2. Make sure to update the `dolfinx_latest.sif` container image.

The following commands can be used on the RWTH cluster / Linux:

```
wget https://mbd.pages.rwth-aachen.de/courses/cmm/content/exercises/notebooks/homework0
rm -rf library
wget https://mbd.pages.rwth-aachen.de/courses/cmm/content/exercises/notebooks/library.t
tar -xvzf library.tar.gz
rm library.tar.gz
rm dolfinx_latest.sif
apptainer pull oras://registry.git.rwth-aachen.de/mbd/courses/containers/dolfinx:latest
```

Batch job execution

To run the notebook on the backend of the HPC Cluster, run

```
sbatch < library/batch_hw04.sh
```

To query the current status of the batch job, run

```
squeue --me
```

This will show the status (`ST`) of the job. If it is `R`, it is running; if it is `PD`, it is pending / waiting for resources.

2 Todo's

Make sure that all tests in the notebook run without errors. The following tasks need to be completed:

- ☐ Update the participant list above
- ☐ **Task 1** (P1 — Naive SWE): implement the physical flux tensor `flux(Q)`
- ☐ **Task 2** (P1 — Naive SWE): implement the Lax-Friedrichs numerical flux `numerical_flux(...)`
- ☐ **Task 3** (P1 — Naive SWE): implement the gravity body force source `source_gravity(Q)`

- ☐ **Task 4** (P1 — Naive SWE): implement the topography source term `source_topography(Q, b)`
- ☐ **Task 5** (P1 — Naive SWE): implement the Voellmy friction coefficient `friction_coefficient(Q)`
- ☐ **Task 6** (P1 — Naive SWE): implement the explicit friction source `source_friction(q, trial_q)`
- ☐ **Task 7** (P2 — Semi-implicit friction): override `source_friction` to use semi-implicit treatment
- ☐ **Task 8** (P3 — Well-balancing): implement the hydrostatic reconstruction flux with Audusse correction
- ☐ **Task 9** (P3 — Well-balancing): override `source_topography` to return zero (topography is now in the flux)
- ☐ **Task 10** (P4 — Cascadia): run the debris flow simulation and complete the analysis

3 Imports

```
import os
import sys
import numpy as np
import rasterio

from mpi4py import MPI
from petsc4py import PETSc
import dolfinx
from dolfinx import fem, mesh
import basix
import ufl
import pyvista
import tqdm
from dolfinx.fem import petsc

CMM_DIR = os.getcwd()
os.environ["PYTHONPATH"] = f"{CMM_DIR}:{os.environ.get('PYTHONPATH', '')}"

import library.plot
import library.helpers
import library.tests

def print_rank0(*args, **kwargs):
    if MPI.COMM_WORLD.rank == 0:
        print(*args, **kwargs)
```

```
output_dir = ""
if MPI.COMM_WORLD.rank == 0:
    output_dir = library.helpers.make_unique_dir(os.path.join(CMM_DIR, 'results_ex_04'))
output_dir = MPI.COMM_WORLD.bcast(output_dir, root=0)

interactive_mode_is_on = hasattr(sys, 'ps1')

if os.getenv('DEVCONTAINER') and interactive_mode_is_on:
    pyvista.start_xvfb()
```

4 Background: Shallow Water Equations

From scalar advection to a system of conservation laws

In Homework 01 we solved the **scalar** advection equation

$$\partial_t q + \nabla \cdot (q\mathbf{v}) = 0,$$

where q was a single transported quantity and $\mathbf{v}$ was a prescribed velocity. The shallow water equations (SWE) generalise this to a **system**: the unknowns are the water height h and the depth-averaged momentum vector $(hu, hv)^T$, and the velocity $\mathbf{u} = (u, v)^T$ is itself part of the solution. Everything from Homework 01 (the DG weak form, the Lax-Friedrichs numerical flux, the explicit time stepping) carries over to this system unchanged. The only new ingredient is that the flux $\underline{\underline{F}}$ is now a 3×2 tensor rather than a scalar–vector product.

Conservative form

We work with the conservative state $\mathbf{Q} = [h,\, hu,\, hv]^T$. The bed elevation $b(x, y)$ is an **auxiliary** scalar that does not evolve in time and is carried separately.

$$\begin{aligned}
&\partial_t h + \partial_x(hu) + \partial_y(hv) = 0 \\
&\partial_t(hu) + \partial_x(\alpha hu^2 + \tfrac{g_z}{2}h^2) + \partial_y(\alpha huv) = -g_z h\,\partial_x b + g_x h + S_x^{\text{friction}} \\
&\partial_t(hv) + \partial_x(\alpha huv) + \partial_y(\alpha hv^2 + \tfrac{g_z}{2}h^2) = -g_z h\,\partial_y b + g_y h + S_y^{\text{friction}}
\end{aligned}$$

or in compact divergence form

$$\partial_t \mathbf{Q} + \nabla \cdot \underline{\underline{F}}(\mathbf{Q}) = \mathbf{S}_{\text{gravity}} + \mathbf{S}_{\text{topography}} + \mathbf{S}_{\text{friction}}.$$

Here g_x, g_y, g_z are the components of gravity projected onto the bed-parallel (x, y) and bed-normal (z) directions. For a flat horizontal bed, $g_x = g_y = 0$ and $g_z = 9.81$ m/s^2; for a tilted terrain (P4) we decompose g accordingly.

Flux tensor

Reading off the x- and y-columns from the equations above, the physical flux tensor is

$$\underline{\underline{F}}(\mathbf{Q}) = \begin{bmatrix} hu & hv \\ \alpha hu^2 + \frac{g_z}{2}h^2 & \alpha huv \\ \alpha huv & \alpha hv^2 + \frac{g_z}{2}h^2 \end{bmatrix},$$

where α is the shape factor arising from depth-averaging a non-uniform velocity profile. For a uniform profile $\alpha = 1$, which we use throughout this homework.

Source terms

- $\mathbf{S}_{\text{gravity}} = [0,\ g_x h,\ g_y h]^T$: gravity components along the bed
- $\mathbf{S}_{\text{topography}} = [0,\ -g_z h\, \partial_x b,\ -g_z h\, \partial_y b]^T$: bed slope pressure
- $\mathbf{S}_{\text{friction}} = -c(|\mathbf{u}|)\,[0,\ hu,\ hv]^T$: friction deceleration

Friction model: Voellmy

The Voellmy law combines a dry-Coulomb part and a turbulent Chezy part:

$$c(|\mathbf{u}|) = \frac{\mu\, g_z}{|\mathbf{u}|} + \frac{g\,|\mathbf{u}|}{\xi},$$

with dry-friction coefficient μ (dimensionless) and turbulence coefficient ξ (m/s^2). Note that $c \to \infty$ as $|\mathbf{u}| \to 0$. This singularity is the root cause of the instability we study in P1 and fix in P2. In the implementation the denominator is regularised by a small ε:

$$c(|\mathbf{u}|) \approx \frac{\mu\, g_z}{|\mathbf{u}| + \varepsilon} + \frac{g\,|\mathbf{u}|}{\xi}.$$

5 Select simulation to run

The workflow for this homework is:

1. **Implement** the TODOs below (flux, source terms, implicit friction, well-balancing)
2. **Test locally**: run the smoke tests in the notebook to verify the implementation
3. **Submit to cluster**: enable the friction model runs and submit via:

   `sbatch library/batch_hw04.sh`

   Each run saves its result as a `.npy` file in the output directory.
4. **Analyse**: after the cluster job finishes, re-run the analysis cells to load and compare results.

We do **not** need to wait for a previous job to finish, as each simulation writes to a unique output directory.

```
run_p1       = True   # P1
run_p2       = True   # P2
run_p3       = True   # P3
run_cascadia = False  # P4: Cascadia debris flow (long run - submit to cluster)
```

6 Solver utilities

The cells below contain shared infrastructure used by all problems. They are **provided**; there is no need to modify them. It is however worth reading through to understand the interface that the concrete SWE classes build on.

Mesh creation

Builds a rectangular quadrilateral mesh and returns the domain, a reference edge length (used for CFL time-step estimation), and the corner coordinates `P0`, `P1`.

```
def create_domain(Nx, Ny, P0=[0, 0], P1=[3, 3]):
    """Create a rectangular quadrilateral mesh over [P0, P1] with Nx x Ny elements."""
    domain = mesh.create_rectangle(
        MPI.COMM_WORLD, [P0, P1], [Nx, Ny], cell_type=mesh.CellType.quadrilateral
    )
    reference_edge_length = (P1[0] - P0[0]) / Nx
    return domain, reference_edge_length, P0, P1
```

Function spaces

A 3-component DG(0) space for the conservative state $\mathbf{Q} = [h,\, hu,\, hv]^T$, and a scalar DG(0) space for the bed elevation b.

```
def create_function_space(domain):
    """3-component DG0 space for the conservative state Q = [h, hu, hv]."""
    elem_Q = basix.ufl.element("DG", domain.topology.cell_name(), 0, shape=(3,))
    return fem.functionspace(domain, elem_Q)
```

SWEBase — abstract interface

All SWE solvers in this homework inherit from `SWEBase`. It owns the shared infrastructure (parameters, primitive recovery, time-step control, weak form assembly) and declares the physics interface as **abstract methods** that every concrete subclass must implement.

The code cell below is **provided in full**; there is no need to modify it. The key things to note are:

- `get_primitives` recovers (h, u, v) from the conservative state $\mathbf{Q}$ in a wet-dry safe way.
- `get_weak_form` assembles the linear system (A, b) for one time step. It calls `self.numerical_flux`, `self.source`, etc., which are defined by the subclass.
- `compute_time_step_size` picks Δt from the CFL condition.

Python abstract base classes

An *abstract base class* (ABC) is a class that cannot be instantiated directly. It declares a contract: a set of methods every subclass is required to implement, using the `@abstractmethod` decorator. This is Python's way of expressing the same idea as a pure-virtual function in C++ or an interface in Java.

Further reading: Python docs: `abc` module · Real Python: abstract base classes

```
from abc import ABC, abstractmethod

class SWEBase(ABC):
    """Abstract base for all SWE solvers.

    Infrastructure (provided):
        __init__          -- physics parameter dict
        get_primitives    -- wet-dry stable h, u, v recovery
        max_abs_eigenvalue -- Rusanov wave-speed bound
```

```
    source            -- composite source (gravity + topo + friction)
    compute_time_step_size -- CFL-adaptive dt
    get_weak_form     -- assemble (lhs, rhs) for one time step

Interface (must be overridden):
    flux, numerical_flux,
    source_gravity, source_topography,
    friction_coefficient, source_friction
"""

def __init__(self, domain):
    C = lambda v: dolfinx.fem.Constant(domain, dolfinx.default_scalar_type(float(v)))
    self.param = {
        "eps":          C(1e-6),
        "gx":           C(0.0),
        "gy":           C(0.0),
        "gz":           C(9.81),
        "shape_factor": C(1.0),
        "voellmy_mu":   C(0.0),
        "voellmy_xi":   C(1e30),
    }

# ------------------------------------------------------------------
# Provided helpers
# ------------------------------------------------------------------

def get_primitives(self, Q):
    """Return (h, u, v) with wet-dry safe velocity recovery."""
    eps = self.param["eps"]
    h = Q[0]; h = ufl.conditional(h > 0, h, 0)
    h_inv = (ufl.sqrt(2.) * h) / (
        ufl.sqrt(h**4 + ufl.conditional(h**4 > eps, h**4, eps)))
    return h, Q[1] * h_inv, Q[2] * h_inv

def max_abs_eigenvalue(self, Q, n):
    """Rusanov wave-speed bound in direction n."""
    h, u, v = self.get_primitives(Q)
    return (self.param["shape_factor"]
            * abs(ufl.dot(ufl.as_vector([u, v]), n))
            + ufl.sqrt(ufl.conditional(h > 0, self.param["gz"] * h, 0)))

def source(self, q, b, trial_q):
```

```
        """Composite source: gravity + topography + friction."""
        return (self.source_gravity(q)
                + self.source_topography(q, b)
                + self.source_friction(q, trial_q))

    def compute_time_step_size(self, Q, reference_cell_diameter, CFL=0.45):
        """CFL-adaptive time step. Returns dt >= 1e-6."""
        domain = Q.function_space.mesh
        elem_DG0 = basix.ufl.element("DG", domain.topology.cell_name(), 0)
        space_DG0 = fem.functionspace(domain, elem_DG0)
        ip = space_DG0.element.interpolation_points()
        lam_x = fem.Function(space_DG0); lam_y = fem.Function(space_DG0)
        lam_x.interpolate(fem.Expression(
            self.max_abs_eigenvalue(Q, ufl.as_vector([1.0, 0.0])), ip))
        lam_y.interpolate(fem.Expression(
            self.max_abs_eigenvalue(Q, ufl.as_vector([0.0, 1.0])), ip))
        local_max = max(float(np.max(lam_x.x.array)),
                        float(np.max(lam_y.x.array)))
        gm = MPI.COMM_WORLD.allreduce(local_max, op=MPI.MAX)
        if gm <= 0 or np.isnan(gm) or np.isinf(gm):
            return 1e-6
        dt = CFL * reference_cell_diameter / gm
        return dt if (not np.isnan(dt) and not np.isinf(dt) and dt >= 1e-6) else 1e-6

    def get_weak_form(self, functionspace, q_n, b, dt):
        """Assemble (lhs_form, rhs_form) for one explicit/semi-implicit step.

        Reflecting (slip-wall) BCs are applied on all four boundary sides.
        ``ufl.lhs`` / ``ufl.rhs`` split any trial-function-dependent term
        (e.g. the semi-implicit friction in SWEImplicit) onto the matrix side.
        """
        domain = functionspace.mesh
        n = ufl.FacetNormal(domain)
        dS = ufl.Measure("dS", domain=domain)
        facet_tags = library.helpers.generate_facets_tags(domain, P0, P1)
        ds = ufl.Measure("ds", domain=domain, subdomain_data=facet_tags)
        dx = ufl.dx
        q = q_n
        hu_n = q[1] * n[0] + q[2] * n[1]
        q_bc = ufl.as_vector([q[0],
                              q[1] - 2 * hu_n * n[0],
                              q[2] - 2 * hu_n * n[1]])
```

```
        test_q  = ufl.TestFunction(functionspace)
        trial_q = ufl.TrialFunction(functionspace)

        F_plus, F_minus = self.numerical_flux(
            q("+"), q("-"), b("+"), b("-"), n("+"))
        wf  = ufl.dot(test_q, (trial_q - q) / dt) * dx
        wf += (ufl.dot(test_q("+"), F_plus)
               + ufl.dot(test_q("-"), F_minus)) * dS
        F_bdry, _ = self.numerical_flux(q, q_bc, b, b, n)
        wf += ufl.dot(test_q, F_bdry) * (ds(1) + ds(2) + ds(3) + ds(4))
        wf -= ufl.dot(test_q, self.source(q, b, trial_q)) * dx
        return fem.form(ufl.lhs(wf)), fem.form(ufl.rhs(wf))

    # ------------------------------------------------------------------
    # Abstract interface - every subclass must implement these
    # ------------------------------------------------------------------

    @abstractmethod
    def flux(self, Q):
        """Physical flux tensor F(Q), shape (3, 2)."""

    @abstractmethod
    def numerical_flux(self, Ql, Qr, bl, br, n):
        """Facet numerical flux. Returns (F_plus, F_minus) per side."""

    @abstractmethod
    def source_gravity(self, Q):
        """Gravity body force [0, gx*h, gy*h]."""

    @abstractmethod
    def source_topography(self, Q, b):
        """Topography (bed-slope) source -gz*h*grad(b) (or zero for WB)."""

    @abstractmethod
    def friction_coefficient(self, Q):
        """Voellmy friction coefficient c(|u|)."""

    @abstractmethod
    def source_friction(self, q, trial_q):
        """Friction source. Uses q for explicit, trial_q for semi-implicit."""
```

Time loop

Adaptive CFL-based explicit time stepping. Takes an `swe` object (any subclass of `SWEBase`) and drives the simulation forward, writing VTK snapshots and collecting diagnostics.

```
def solve_time_loop(name, swe, functionspace, ic_q, b,
                    end_time, output_path, CFL, reference_edge_length,
                    max_timesteps=10000):
    """One ``LinearProblem`` solve per step. Returns
    ``(q_final, diagnostics)`` with per-step ``t``, ``dt``, ``mass``,
    ``min_h``, ``max_mom``."""
    from dolfinx.fem.petsc import LinearProblem
    domain = functionspace.mesh
    t_const = fem.Constant(domain, dolfinx.default_scalar_type(0.0))
    dt = fem.Constant(domain, dolfinx.default_scalar_type(0.1))

    q_n   = fem.Function(functionspace, name="q_n")
    q_np1 = fem.Function(functionspace, name="q_np1")
    q_n.interpolate(ic_q); q_np1.interpolate(ic_q)

    lhs_form, rhs_form = swe.get_weak_form(functionspace, q_n, b, dt)
    # DG0 LHS is diagonal -> Jacobi is an exact one-step solve.
    problem = LinearProblem(lhs_form, rhs_form, u=q_np1,
                            petsc_options={"ksp_type": "preonly", "pc_type": "jacobi"})

    os.makedirs(output_path, exist_ok=True)
    vtk_writer = dolfinx.io.VTKFile(domain.comm, os.path.join(output_path, f"{name}.pvd"), "w
    vtk_writer.write_function(library.helpers.extract_scalar_fields(q_n) + [b], t=0.0)
    n_snapshots = 50
    dt_snapshot = end_time / n_snapshots
    next_snapshot_time = dt_snapshot

    cell_area_form = fem.form(1.0 * ufl.dx(domain=domain))
    cell_area_total = fem.assemble_scalar(cell_area_form)

    def diagnostics_now(q):
        h_arr  = q.x.array[0::3]
        hu_arr = q.x.array[1::3]; hv_arr = q.x.array[2::3]
        mass_local = float(np.sum(h_arr)) * (cell_area_total / len(h_arr))
        mass_g    = MPI.COMM_WORLD.allreduce(mass_local, op=MPI.SUM)
        min_h_g   = MPI.COMM_WORLD.allreduce(float(np.min(h_arr)), op=MPI.MIN)
        max_mom_l = float(np.max(np.sqrt(hu_arr**2 + hv_arr**2)))
        max_mom_g = MPI.COMM_WORLD.allreduce(max_mom_l, op=MPI.MAX)
```

```
        return mass_g, min_h_g, max_mom_g

    t_hist, dt_hist, mass_hist, min_h_hist, max_mom_hist = [], [], [], [], []
    m0, mh0, mm0 = diagnostics_now(q_n)
    t_hist.append(0.0); dt_hist.append(0.0)
    mass_hist.append(m0); min_h_hist.append(mh0); max_mom_hist.append(mm0)

    progress = tqdm.tqdm(desc="Setup " + name + ", solving PDE", total=end_time)
    i = 0
    dt_floor_count = 0
    DT_FLOOR = 1.1e-6
    warned = False
    while t_const.value < end_time and i < max_timesteps:
        q_n.interpolate(q_np1)
        dt.value = swe.compute_time_step_size(q_n, reference_edge_length, CFL=CFL)
        progress.update(dt.value)

        if dt.value <= DT_FLOOR:
            dt_floor_count += 1
            if dt_floor_count == 10 and not warned:
                print_rank0(f"[{name}] WARNING: dt collapsed to {dt.value:.1e} "
                            f"for 10 consecutive steps at t={t_const.value:.4e} "
                            f"(stiff source -- check friction).")
                warned = True
        else:
            dt_floor_count = 0

        problem.solve()

        t_const.value += dt.value; i += 1
        m, mh, mm = diagnostics_now(q_np1)
        t_hist.append(float(t_const.value)); dt_hist.append(float(dt.value))
        mass_hist.append(m); min_h_hist.append(mh); max_mom_hist.append(mm)

        if t_const.value > next_snapshot_time:
            vtk_writer.write_function(library.helpers.extract_scalar_fields(q_np1) + [b], t=t
            next_snapshot_time += dt_snapshot

    progress.close()
    return q_np1, {
        "t":       np.asarray(t_hist),
        "dt":      np.asarray(dt_hist),
```

```
        "mass":     np.asarray(mass_hist),
        "min_h":    np.asarray(min_h_hist),
        "max_mom":  np.asarray(max_mom_hist),
        "n_steps": i,
        "reached_end_time": t_const.value >= end_time,
    }
```

7 P1 — Naive SWE with explicit friction

We build the first concrete SWE solver: standard Rusanov / Lax-Friedrichs flux, topography as a cell source, and Voellmy friction applied **explicitly**, meaning the friction force is computed from $\mathbf{Q}^n$ and placed on the RHS.

All six physics operators (Tasks 1–6) are declared `@abstractmethod` in `SWEBase`; our task is to fill them in here.

1. Mesh and initial conditions

We test the explicit scheme on a **parabolic ramp-bowl**: a bowl-shaped bed centred in the domain with a wet patch released off-centre. The water slides toward the bottom, sloshes, and should eventually settle due to friction. With explicit friction, this run will not reach steady state. That is the point.

```
def setup_ramp_bowl():
    """Parabolic bowl in the centre of a large enough domain that the
    water never reaches the boundary. Water released as a wet patch
    off-centre slides toward the bottom, sloshes, and eventually
    settles. With reflecting walls and water bounded away from the
    boundary, the discrete conservation of the scheme is visible.
    """
    def ic_b(x):
        r2 = (x[0] - 2.0)**2 + (x[1] - 2.0)**2
        return np.minimum(0.3 * r2, 0.5)

    def ic_q(x):
        R = 0.3
        r = np.sqrt((x[0] - 1.3)**2 + (x[1] - 2.0)**2)
        h0 = np.where(r <= R, 0.3, 0.0)
        return np.array([h0, 0.*x[0], 0.*x[0]])
```

```
    domain_rb, ref_rb, P0_rb, P1_rb = create_domain(80, 80, P0=[0, 0], P1=[4, 4])
    space_Q_rb = create_function_space(domain_rb)
    space_b_rb = fem.functionspace(domain_rb, basix.ufl.element("DG", domain_rb.topology.cell
    b_rb = fem.Function(space_b_rb, name="b"); b_rb.interpolate(ic_b)
    return ic_q, ic_b, domain_rb, ref_rb, P0_rb, P1_rb, space_Q_rb, b_rb
```

```
# Build the ramp-bowl test once and share its domain across Blocks A and B
# so we can overlay the cross-sections.
ic_q_rb, ic_b_rb, domain_rb, reference_edge_length_rb, P0_rb, P1_rb, space_Q_rb, b_rb = setup
q_ic_rb = fem.Function(space_Q_rb, name="q_initial"); q_ic_rb.interpolate(ic_q_rb)
```

2. The `SWE` class

`SWE(SWEBase)` inherits all the infrastructure from `SWEBase` and implements the six abstract methods. Complete Tasks 1–6 in the cell below.

> **!** STUDENT TODO — Tasks 1–6
>
> Implement the six methods below inside `SWE`:
> **Task 1** — `flux(Q)`: the 3×2 physical flux tensor $F(Q)$.
> **Task 2** — `numerical_flux(Ql, Qr, bl, br, n)`: Lax-Friedrichs / Rusanov facet flux. Return *(F_+, F_−)* as an anti-symmetric pair.
> **Task 3** — `source_gravity(Q)`: body force $[0,\, g_x h,\, g_y h]^T$.
> **Task 4** — `source_topography(Q, b)`: bed-slope source $[0,\, -g_z h \partial_x b,\, -g_z h \partial_y b]^T$.
> **Task 5** — `friction_coefficient(Q)`: Voellmy coefficient $c(|\mathbf{u}|) = \mu g_z/(|\mathbf{u}|+\varepsilon)+g|\mathbf{u}|/\xi$.
> **Task 6** — `source_friction(q, trial_q)`: **explicit** friction source $[0,\, -c(|\mathbf{u}^n|)(hu)^n,\, -c(|\mathbf{u}^n|)(hv)^n]^T$.

```
class SWE(SWEBase):
    """Naive SWE: standard Rusanov flux, cell-source topography, explicit Voellmy friction."

    ##################TODO 1: Flux tensor#####################################
    # F(Q) is a 3x2 tensor (3 components, 2 space dims).
    # Row 0 (continuity): [hu, hv]
    # Row 1 (x-momentum): [alpha*hu^2 + gz*h^2/2, alpha*hu*hv]
    # Row 2 (y-momentum): [alpha*hu*hv, alpha*hv^2 + gz*h^2/2]
    # alpha = shape_factor (depth-averaging coefficient, usually 1)
    def flux(self, Q):
        h, u, v = self.get_primitives(Q)
        gz = self.param["gz"]; sf = self.param["shape_factor"]
        return ufl.as_tensor([[0, 0], [0, 0], [0, 0]])
```

```
    ########################################################################

    ##################TODO 2: Lax-Friedrichs flux at a facet################
    # Return a per-side tuple (F_plus, F_minus). For the standard
    # scheme they are anti-symmetric (F_minus = -F_plus). The tuple
    # form is kept so the WB version (P3) can carry an asymmetric
    # Audusse correction.
    def numerical_flux(self, Ql, Qr, bl, br, n):
        F = ufl.as_vector([0, 0, 0])
        return F, -F
    ########################################################################

    ##################TODO 3: Gravity body source (g_x h, g_y h)############
    def source_gravity(self, Q):
        h, _, _ = self.get_primitives(Q)
        return ufl.as_vector([0, 0, 0])
    ########################################################################

    ##################TODO 4: Topography source  -g_z h grad(b)#############
    # b is DG0; L2-project to CG1 for a meaningful gradient.
    def source_topography(self, Q, b):
        h, _, _ = self.get_primitives(Q)
        gz = self.param["gz"]
        domain = b.function_space.mesh
        elem_CG1 = basix.ufl.element("CG", domain.topology.cell_name(), 1)
        space_CG1 = fem.functionspace(domain, elem_CG1)
        b_cg1 = library.helpers.project(b, space_CG1)
        grad_b = ufl.nabla_grad(b_cg1)
        return ufl.as_vector([0, 0, 0])
    ########################################################################

    ##################TODO 5: Voellmy friction coefficient c(|u|)###########
    # c(|u|) = mu * g_z / (|u| + eps) + g * |u| / xi
    def friction_coefficient(self, Q):
        h, u, v = self.get_primitives(Q)
        eps = self.param["eps"]
        abs_u = ufl.sqrt(u**2 + v**2)
        return 0.0
    ########################################################################

    ##################TODO 6: EXPLICIT friction source######################
    # S^friction = -c(|u^n|) * [0, (hu)^n, (hv)^n]
```

```
    def source_friction(self, q, trial_q):
        c = self.friction_coefficient(q)
        return ufl.as_vector([0, 0, 0])
    ###############################################################################
```

3. Smoke test

The smoke test runs a set of unit-level checks on the implemented operators (correct tensor shapes, flux anti-symmetry, zero source at rest, etc.). All checks should pass before running the full simulation.

```
library.tests.test_swe_operators(SWE)
```

```
Operator smoke tests for SWE:
  flux                    : pass
  source_gravity          : pass
  source_topography       : pass
  friction_coefficient    : pass
  source_friction         : pass
  max_abs_eigenvalue      : pass
  numerical_flux          : pass
```

4. Solve

We run the ramp-bowl simulation with explicit Voellmy friction and observe what happens.

```
if run_p1:
    domain, reference_edge_length, P0, P1, space_Q, b = \
        domain_rb, reference_edge_length_rb, P0_rb, P1_rb, space_Q_rb, b_rb
    ic_q = ic_q_rb

    swe = SWE(domain)  # default: EXPLICIT friction
    swe.param["voellmy_mu"].value = 0.4
    swe.param["voellmy_xi"].value = 50.0

    time_end = 1.0
    CFL = 0.45

    q_e, diag_e = solve_time_loop(
        "explicit_ramp_bowl", swe, space_Q, ic_q, b, time_end,
```

```
        os.path.join(output_dir, "sim"), CFL, reference_edge_length,
        max_timesteps=500,
    )
    print_rank0(f"explicit:   reached t = {diag_e['t'][-1]:.4f} in {diag_e['n_steps']} steps
                f"max |hu| = {diag_e['max_mom'].max():.3e}; min h = {diag_e['min_h'].min():+
                f"mass drift = {diag_e['mass'][-1]-diag_e['mass'][0]:+.3e}")
```

```
Setup explicit_ramp_bowl, solving PDE:   7%|          | 0.07442550804423725/1.0 [00:02<00:17,
```

```
[explicit_ramp_bowl] WARNING: dt collapsed to 1.0e-06 for 10 consecutive steps at t=7.4435e-
02 (stiff source -- check friction).
```

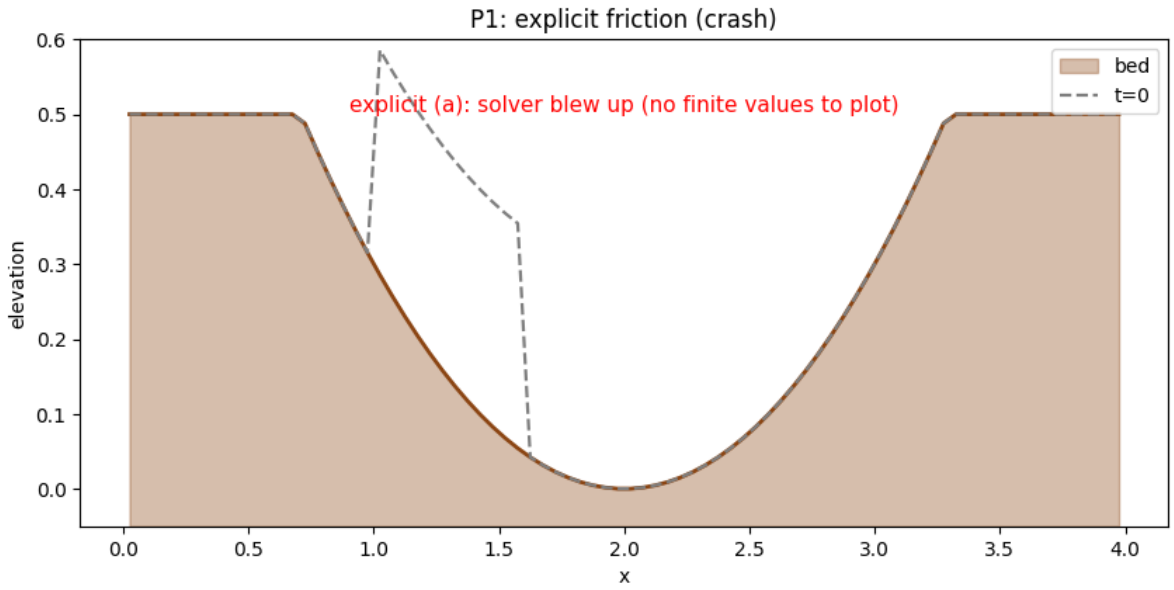
```
Setup explicit_ramp_bowl, solving PDE:   7%|          | 0.07491850804423775/1.0 [00:21<04:21,
```

```
explicit:   reached t = 0.0749 in 500 steps; max |hu| = nan; min h = +nan; mass drift = +nan
```

5. Visualize

Cross-section of the free surface at $y = 2$ m, sampled at several time instants. We expect to see the time step stall and the profile blow up.

```
if run_p1 and interactive_mode_is_on:
    fig, ax, x, b_arr, cells = library.plot.plot_bed_cross_section(
        domain_rb, b_rb, y_target=2.0,
        title="P1: explicit friction (crash)")
    library.plot.add_surface_line(ax, x, b_arr, q_ic_rb, cells, "t=0",
                                  color="tab:gray", ls="--")
    drawn = library.plot.add_surface_line(ax, x, b_arr, q_e, cells,
                                          "explicit (a)", color="tab:blue")
    if not drawn:
        library.plot.add_text_box(ax, "explicit (a): solver blew up "
                                   "(no finite values to plot)")
    ax.legend(loc="upper right")
    library.plot.save_and_show(fig, output_dir, "block_A_cross_section")
```



6. Analysis: stiff source terms

What we observed:

- the adaptive time step collapsed to its floor ($\Delta t = 10^{-6}$),
- a warning was printed about repeated dt-floor hits,
- the run reached only a tiny fraction of the requested end time,
- the maximum $|hu|$ blew up to enormous values.

This is the textbook signature of a **stiff source term**. The Voellmy friction coefficient

$$c(|\mathbf{u}|) = \frac{\mu g_z}{|\mathbf{u}|} + \frac{g|\mathbf{u}|}{\xi}$$

diverges as $|\mathbf{u}| \to 0$. An explicit step on the momentum reads

$$(hu)^{n+1} = (1 - \Delta t\, c(|\mathbf{u}^n|))\,(hu)^n.$$

When $c\Delta t > 2$ we have $|1 - \Delta t\, c| > 1$: friction *amplifies* and *flips* the momentum each step. The adaptive CFL chops Δt but cannot keep up; the run stalls.

The fix is to treat friction **semi-implicitly**: evaluate the coefficient at $\mathbf{Q}^n$ but apply it to the unknown $\mathbf{Q}^{n+1}$. The resulting per-cell factor $1/(1 + \Delta t c) \leq 1$ damps the momentum unconditionally. We implement this in `SWEImplicit` in P2.

7. Analysis: eigenvalue argument

The blowup was not bad luck with a particular initial condition; it was structural. The friction source on the momentum is $\mathbf{S}^{\text{friction}} = -c(|\mathbf{u}|)\,[0, hu, hv]^T$. Linearising around $\mathbf{Q}^n$ and looking at the momentum equation alone, the source contribution to the Jacobian is

$$\frac{\partial \mathbf{S}^{\text{friction}}}{\partial (hu, hv)} \approx -c(|\mathbf{u}|)\,\mathbf{I}_2 \quad \Longrightarrow \quad \text{eigenvalues } = -c(|\mathbf{u}|).$$

For an explicit Euler step in time to be stable, we need $\Delta t\,|c| < 2$. The CFL condition from the flux gives $\Delta t \leq \text{CFL}\, h_{\text{cell}}/\lambda_{\max}$, bounded above by a fixed number for a given mesh. As soon as $c(|\mathbf{u}|)$ exceeds $2/\Delta t_{\text{CFL}}$, the explicit friction destabilises the system *regardless* of what the CFL says about the flux.

The Voellmy coefficient is $c(|\mathbf{u}|) = \mu g_z/(|\mathbf{u}| + \varepsilon) + g\,|\mathbf{u}|/\xi$, which **diverges** as $|\mathbf{u}| \to 0$. The plot below shows c and the corresponding maximum stable explicit time step $2/c$ as a function of $|\mathbf{u}|$, and compares to the flux CFL on our mesh.

8 P2 — Semi-implicit friction

`SWEImplicit` inherits from `SWE` and overrides exactly one method, `source_friction`. Instead of using the current-step state $\mathbf{Q}^n$, the friction force acts on the trial function $\mathbf{Q}^{n+1}$, so `ufl.lhs` moves it onto the matrix:

$$(1 + \Delta t\, c(|\mathbf{u}^n|))\,(hu)^{n+1} = (hu)^n + \Delta t\,(\text{flux divergence} + \text{other sources})$$

This is a *linearised implicit* treatment: c is frozen at time n, but the velocity it multiplies is the unknown at time $n+1$. The system remains linear and the original solver works without modification.

1. Mesh and initial conditions

We reuse the ramp-bowl setup from P1 (`domain_rb`, `space_Q_rb`, `b_rb`, `ic_q_rb` are already defined).

2. The `SWEImplicit` class

`SWEImplicit(SWE)` overrides only `source_friction`. Everything else (flux, numerical flux, topography source) is inherited unchanged from `SWE`.

! STUDENT TODO — Task 7

Task 7 — override `source_friction` in `SWEImplicit` to use the semi-implicit treatment:

$$S^{\text{friction}} = -c(|\mathbf{u}^n|)\,[0,\ \text{trial_q}[1],\ \text{trial_q}[2]]^T$$

Use `trial_q` (not `q`) for the momentum components so the friction term is absorbed into the linear system.

```
class SWEImplicit(SWE):
    """Semi-implicit friction: c evaluated at q^n, applied to trial_q."""

    ##################TODO 7: SEMI-IMPLICIT friction source################
    # Same form as TODO 6 but use trial_q instead of q so the friction
    # term lands on the LHS:
    #   S^friction = -c(|u^n|) * [0, trial_q[1], trial_q[2]]
    def source_friction(self, q, trial_q):
        c = self.friction_coefficient(q)
        return ufl.as_vector([0, 0, 0])
    ########################################################################
```

3. Smoke test

Same checks as in P1, now applied to `SWEImplicit`. The smoke test should pass; the key check is that the semi-implicit friction term appears on the left-hand side of the assembled system.

```
library.tests.test_swe_operators(SWEImplicit)
```

```
Operator smoke tests for SWEImplicit:
  flux                    : pass
  source_gravity          : pass
  source_topography       : pass
  friction_coefficient    : pass
  source_friction         : pass
  max_abs_eigenvalue      : pass
  numerical_flux          : pass
```

4. Solve

Same ramp-bowl geometry and friction parameters, only the scheme changes. This time the run should complete.

```
if run_p2:
    domain, reference_edge_length, P0, P1, space_Q, b = \
        domain_rb, reference_edge_length_rb, P0_rb, P1_rb, space_Q_rb, b_rb
    ic_q = ic_q_rb

    swe = SWEImplicit(domain)
    swe.param["voellmy_mu"].value = 0.4
    swe.param["voellmy_xi"].value = 50.0

    time_end = 5.0
    CFL = 0.45

    q_i, diag_i = solve_time_loop(
        "implicit_ramp_bowl", swe, space_Q, ic_q, b, time_end,
        os.path.join(output_dir, "sim"), CFL, reference_edge_length,
    )
    print_rank0(f"implicit:   reached t = {diag_i['t'][-1]:.4f} in {diag_i['n_steps']} steps
                f"max |hu| = {diag_i['max_mom'].max():.3e}; min h = {diag_i['min_h'].min():+
                f"mass drift = {diag_i['mass'][-1]-diag_i['mass'][0]:+.3e}")
```

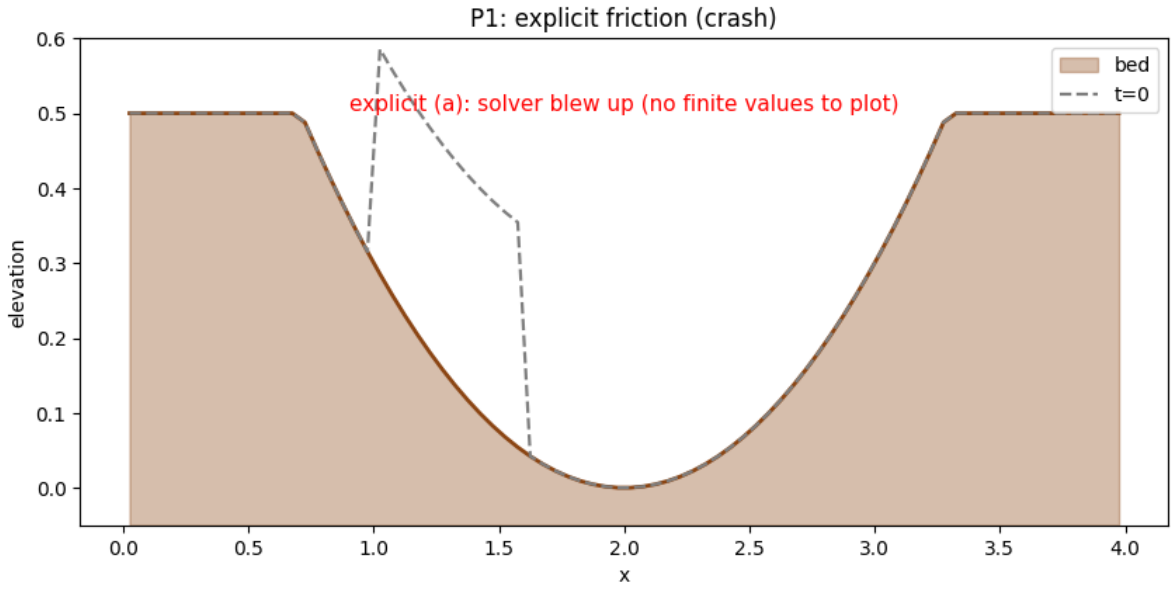
```
Setup implicit_ramp_bowl, solving PDE: 100%|      | 5.012963833521486/5.0 [00:14<00:00,  2.91

implicit:   reached t = 5.0130 in 234 steps; max |hu| = 6.605e-02; min h = +0.000e+00; mass c
1.110e-16
```

5. Visualize

Overlay the explicit (crash) and semi-implicit cross-sections at the same time instants.

```
if run_p2 and interactive_mode_is_on:
    fig, ax, x, b_arr, cells = library.plot.plot_bed_cross_section(
        domain_rb, b_rb, y_target=2.0,
        title="P1: explicit friction (crash)")
    library.plot.add_surface_line(ax, x, b_arr, q_ic_rb, cells, "t=0",
                                  color="tab:gray", ls="--")
    drawn = library.plot.add_surface_line(ax, x, b_arr, q_i, cells,
                                          "explicit (a)", color="tab:blue")
    if not drawn:
```

```
        library.plot.add_text_box(ax, "explicit (a): solver blew up "
                                      "(no finite values to plot)")
    ax.legend(loc="upper right")
    library.plot.save_and_show(fig, output_dir, "block_A_cross_section")
```



6. Analysis: mass and positivity

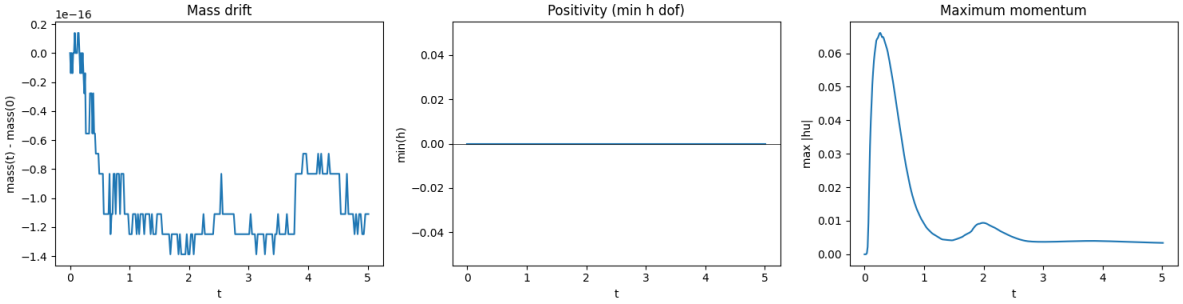
Three diagnostics worth looking at:

- **Mass**: the LF flux is conservative in the discrete sense; the h-equation has no implicit term. With reflecting boundaries the mass should be exactly conserved (up to floating-point round-off in the assembly).
- **min**(h): nothing in the scheme clamps h to be non-negative. The `ufl.conditional(h>0, h, 0)` inside `get_primitives` only protects $1/h$ during primitive recovery; the conservative $Q[0]$ is allowed to dip below zero (loss of positivity).
- **max**$|hu|$: visible bursts here correspond to splashes when the flow hits the bowl.

This test runs fine, but only because the bed is smooth. On a sharper geometry (next) the standard scheme fails.

```
if run_p2 and interactive_mode_is_on:
    from matplotlib import pyplot as plt
    fig, ax = plt.subplots(1, 3, figsize=(15, 4))
    ax[0].plot(diag_i["t"], diag_i["mass"] - diag_i["mass"][0])
```

```
    ax[0].set_xlabel("t"); ax[0].set_ylabel("mass(t) - mass(0)")
    ax[0].set_title("Mass drift")
    ax[1].plot(diag_i["t"], diag_i["min_h"]); ax[1].axhline(0, color="k", lw=0.5)
    ax[1].set_xlabel("t"); ax[1].set_ylabel("min(h)")
    ax[1].set_title("Positivity (min h dof)")
    ax[2].plot(diag_i["t"], diag_i["max_mom"])
    ax[2].set_xlabel("t"); ax[2].set_ylabel("max |hu|")
    ax[2].set_title("Maximum momentum")
    plt.tight_layout(); plt.show()
```



7. Further test: bowl + step + wall

We now run the **same SWEImplicit scheme** on a different geometry:

- left half: a parabolic bowl filled to lake-at-rest,
- middle: a *step up* in the bed (one cell, jump of 0.5), so the lake's rim is in direct contact with a near-vertical wall,
- right half: a tall wall.

The lake is at rest at $\eta_0 = 0$. With friction enabled (Voellmy, $\mu = 0.1$, $\xi = 100$), the WB advantage shows up dramatically. The non-WB scheme cannot maintain lake-at-rest across the step facet: the standard Lax-Friedrichs dissipation pumps water uphill across the step regardless of hu, friction cannot damp this because friction acts on hu not on h, and the bowl visibly empties.

```
def setup_lake_at_rest_perturbed():
    """Bowl + step + wall stress test for well-balancing.

    Geometry (cross-section through y = 2):

    - x in [0, 2.5]:       parabolic bowl, b = -0.5 + 0.3 r^2
                           (r is the distance from (1.5, 2)). The bowl
```

```
                            rim reaches b = -0.2 at r = 1.
    - x in [2.5, 4.5]:      flat plateau at b = 0.3. The bed jumps from
                            -0.2 to +0.3 at x = 2.5 -- this is the
                            "step" -- and the discrete imbalance per
                            facet is O(g_z * dh^2). It is the textbook
                            well-balancing failure mode.
    - x in [4.5, 6]:        linear wall rising to b = 1.5 at x = 6
                            ("high wall on the right").

    Lake-at-rest at eta_0 = 0 fills only the bowl. With reflecting
    walls the only mass exchange the non-WB scheme can do is
    *internal* leakage across the step.
    """
    def ic_b(x):
        r2 = (x[0] - 1.5)**2 + (x[1] - 2.0)**2
        bowl    = -0.5 + 0.3 * r2
        plateau = 0.3 * np.ones_like(x[0])
        wall    = 0.3 + 1.2 * np.maximum(x[0] - 4.5, 0.0)
        right = np.where(x[0] >= 4.5, wall, plateau)
        return np.where(x[0] < 2.5, bowl, right)

    def ic_q(x):
        eta_0 = 0.0
        h = np.maximum(eta_0 - ic_b(x), 0.0)
        return np.array([h, 0.*x[0], 0.*x[0]])

    domain_lar, ref_lar, P0_lar, P1_lar = create_domain(120, 80, P0=[0, 0], P1=[6, 4])
    space_Q_lar = create_function_space(domain_lar)
    space_b_lar = fem.functionspace(domain_lar, basix.ufl.element("DG", domain_lar.topology.
    b_lar = fem.Function(space_b_lar, name="b"); b_lar.interpolate(ic_b)
    return ic_q, ic_b, domain_lar, ref_lar, P0_lar, P1_lar, space_Q_lar, b_lar
```

```
# Build the bowl+step+wall test once; P2 and P3 share the same domain
# so the cross-sections can be overlaid.
ic_q_lar, ic_b_lar, domain_lar, reference_edge_length_lar, P0_lar, P1_lar, space_Q_lar, b_la
q_ic_lar = fem.Function(space_Q_lar, name="q_initial"); q_ic_lar.interpolate(ic_q_lar)
```

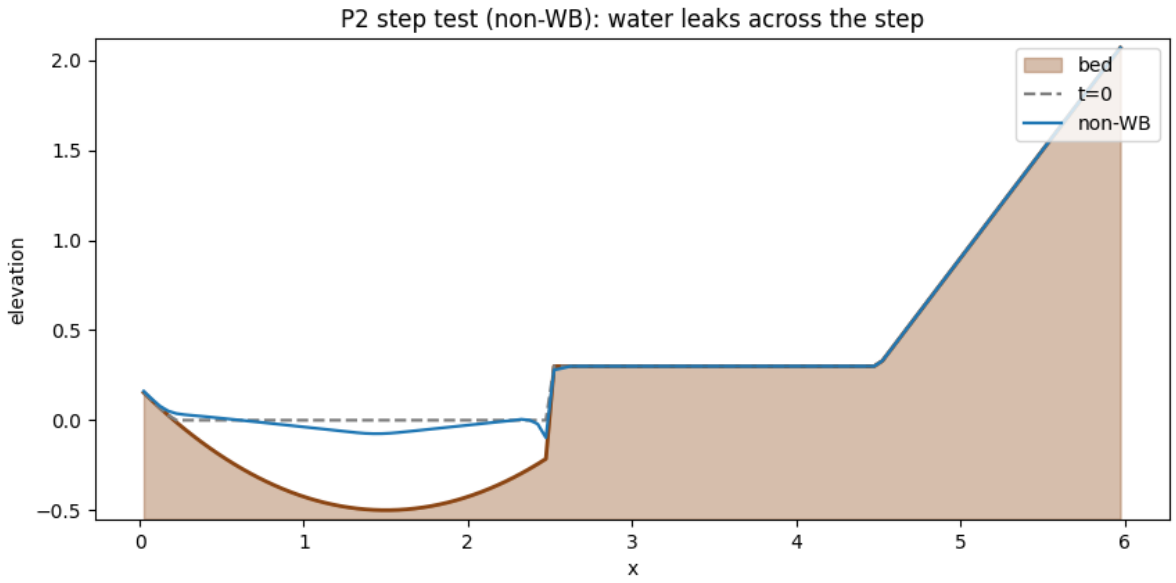
```
if run_p2:
    domain, reference_edge_length, P0, P1, space_Q, b = \
        domain_lar, reference_edge_length_lar, P0_lar, P1_lar, space_Q_lar, b_lar
    ic_q = ic_q_lar
```

```
    swe = SWEImplicit(domain)  # non-WB
    swe.param["voellmy_mu"].value = 0.1
    swe.param["voellmy_xi"].value = 100.0

    time_end = 3.0
    CFL = 0.45

    q_lar_i, diag_lar_i = solve_time_loop(
        "lar_implicit", swe, space_Q, ic_q, b, time_end,
        os.path.join(output_dir, "sim"), CFL, reference_edge_length,
    )
    print_rank0(f"LAR non-WB: reached t = {diag_lar_i['t'][-1]:.4f} in {diag_lar_i['n_steps']
                f"max |hu| = {diag_lar_i['max_mom'].max():.3e}; mass drift = {diag_lar_i['mas
```

```
Setup lar_implicit, solving PDE:   0%|          | 0/3.0 [00:00<?, ?it/s]Setup lar_implicit, s
```

```
LAR non-WB: reached t = 3.0080 in 351 steps; max |hu| = 2.816e-01; mass drift = +0.000e+00
```

```
if run_p2 and interactive_mode_is_on:
    fig, ax, x, b_arr, cells = library.plot.plot_bed_cross_section(
        domain_lar, b_lar, y_target=2.0,
        title="P2 step test (non-WB): water leaks across the step")
    library.plot.add_surface_line(ax, x, b_arr, q_ic_lar, cells, "t=0",
                                  color="tab:gray", ls="--")
    library.plot.add_surface_line(ax, x, b_arr, q_lar_i, cells, "non-WB",
                                  color="tab:blue")
    ax.legend(loc="upper right")
    library.plot.save_and_show(fig, output_dir, "block_B_step_cross_section")
```



Well-balancing: problem statement

The standard scheme is **not well-balanced**. For a flat free surface $h + b = \eta_0$ with $\mathbf{u} = 0$, the continuous SWE has $\partial_t(hu) = 0$: the pressure-flux divergence $\partial_x(g_z h^2/2) = g_z h \partial_x h$ cancels the topography source $-g_z h\, \partial_x b$ because $\partial_x h = -\partial_x b$ on a flat surface.

At the discrete level our standard LF flux uses cell-averaged h_L, h_R in the pressure term while the topography source uses an L2-projected ∇b. The two pieces are computed independently and **do not cancel**: the residual generates a spurious "wind". On a smooth bed it is bounded by the LF dissipation; at a one-cell step it is bounded only by the friction (which does not act on mass), and water leaks across the step.

The fix is **well-balancing** in the sense of Audusse, Bouchut, Bristeau, Klein & Perthame (2004): compute the flux using reconstructed water depths at each facet so that the pressure flux and the topography contribution cancel discretely for a flat free surface. We implement this in P3 as `SWEImplicitWellbalanced`.

9 P3 — Well-balancing: hydrostatic reconstruction

The non-WB scheme's failure on the step is a symptom of a deeper structural issue: the **pressure flux** at a facet and the **topography source** in the adjacent cells are computed **independently** of each other. In the continuous SWE the two are linked:

$$\partial_x(\tfrac{g_z}{2}h^2) \;=\; g_z h\,\partial_x h \;=\; -g_z h\,\partial_x b \quad \text{when } h+b=\text{const},$$

so for a flat free surface they cancel and the steady state is preserved. The discrete versions in our standard scheme do *not* cancel: the Lax-Friedrichs pressure flux uses cell-averaged h_L, h_R that differ across the bed jump, while the topography source uses an L2-projected ∇b, computed in separate code paths.

Hydrostatic reconstruction fixes this by combining the pressure flux *and* the topography contribution into **one operator at each facet**, so they "see each other" by construction. At an interface with bed values b_L, b_R we define the *reconstructed* heights:

$$b^* = \max(b_L, b_R), \qquad h_L^* = \max(0,\; h_L + b_L - b^*), \qquad h_R^* = \max(0,\; h_R + b_R - b^*).$$

These satisfy $h_K^* + b^* = h_K + b_K = \eta_K$ (when $\eta_K > b^*$), so $h_L^* = h_R^*$ **whenever the free surface is flat**, which is exactly the property the standard reconstruction lacks.

We apply the standard Lax-Friedrichs flux to the reconstructed states $(h_K^*, h_K^* u_K, h_K^* v_K)$ instead of the cell-averaged ones. To preserve mass exactly we also add a small **non-conservative facet correction** per side:

$$S_K^{nc} \;=\; -\frac{g_z}{2}((h_K^*)^2 - h_K^2)\,\mathbf{n}_K.$$

This is the second reason `numerical_flux` returns a **tuple** (F_+, F_-): the WB version returns $(F + S_+^{nc}, -F + S_-^{nc})$ where the two corrections are not simple negatives of each other.

Note what has happened: the Audusse HR flux carries **the entire effect of the bed slope**. There is nothing left for the cell source $-g_z h \nabla b$ to compute, so `SWEImplicitWellbalanced.source_topography` returns **0**.

> **i** Riemann problem on a step (optional deeper dive)
>
> Consider a 1D Riemann problem with a vertical bed step at the interface and $\mathbf{u} = 0$ on both sides, with $h_L + b_L = h_R + b_R = \eta_0$ (lake at rest). The HR reconstruction gives $h_L^* = h_R^* = \eta_0 - b^*$ and the LF flux on these *identical* states is
>
> $$F^*(Q_L^*, Q_R^*, n) \;=\; \tfrac{g_z}{2}(h^*)^2\,\mathbf{n} \quad \text{(pressure only, dissipation = 0)}.$$
>
> The per-side Audusse correction adds $-(g_z/2)\,(h_K^*)^2\,\mathbf{n}_K$ to side K. The pressure-flux contribution and the correction therefore **exactly cancel** on each side, and $\partial_t(hu) = 0$ at every cell adjacent to the step. The steady state is preserved.

`SWEImplicitWellbalanced` overrides exactly **two** methods: `numerical_flux` (HR + Audusse correction in the per-side tuple) and `source_topography` (zero, since the topography is now in the flux). Friction is inherited from `SWEImplicit`.

1. Mesh and initial conditions

We reuse `domain_lar`, `space_Q_lar`, `b_lar`, and `ic_q_lar` (bowl + step + wall, built in P2 step 7).

2. The `SWEImplicitWellbalanced` class

`SWEImplicitWellbalanced(SWEImplicit)` overrides two methods. Semi-implicit friction is inherited unchanged.

> ! STUDENT TODO — Tasks 8–9
>
> **Task 8** — `numerical_flux`: implement the Audusse hydrostatic reconstruction flux. At a facet with bed values b_L, b_R: - $b^* = \max(b_L, b_R)$ - $h_L^* = \max(0,\, h_L + b_L - b^*)$ - $h_R^* = \max(0,\, h_R + b_R - b^*)$
> Apply the standard LF flux to (Q_L^*, Q_R^*), then add the Audusse pressure correction:
>
> $$F_+ \;-\!= \frac{g_z}{2}({h_L^*}^2 - h_L^2)\,\mathbf{n}, \qquad F_- \;+\!= \frac{g_z}{2}({h_R^*}^2 - h_R^2)\,(-\mathbf{n})$$
>
> **Task 9** — `source_topography`: the HR flux + Audusse correction carry the entire bed-slope effect, so return the zero vector.

```
class SWEImplicitWellbalanced(SWEImplicit):
    """Audusse hydrostatic reconstruction."""

    ##################TODO 8: HR flux with baked-in Audusse correction#####
    # At a facet with bed values b_L, b_R:
    #   b*    = max(b_L, b_R)
    #   h_L*  = max(0, h_L + b_L - b*)
    #   h_R*  = max(0, h_R + b_R - b*)
    # Apply the standard LF flux to (Q_L*, Q_R*), then add the
    # asymmetric Audusse correction per side:
    #   F_+ -= (g_z/2) (h_L*^2 - h_L^2) * n
    #   F_- += (g_z/2) (h_R*^2 - h_R^2) * n     (n_- = -n)
    def numerical_flux(self, Ql, Qr, bl, br, n):
        zero = ufl.as_vector([0, 0, 0])
        return zero, zero
```

```
    ########################################################################

    ##################TODO 9: Drop the cell topography source###############
    # The HR flux + Audusse correction together carry the entire
    # effect of the bed slope, so the cell source is zero.
    def source_topography(self, Q, b):
        return ufl.as_vector([0, 0, 0])
    ########################################################################
```

3. Smoke test

The key additional check here is the **lake-at-rest test**: for a flat free surface with $\mathbf{u} = 0$, the numerical flux plus the Audusse correction should produce zero net momentum tendency. A passing smoke test means the WB property holds discretely.

```
library.tests.test_swe_operators(SWEImplicitWellbalanced)
```

```
Operator smoke tests for SWEImplicitWellbalanced:
  flux                    : pass
  source_gravity          : pass
  source_topography       : pass
  friction_coefficient    : pass
  source_friction         : pass
  max_abs_eigenvalue      : pass
  numerical_flux          : pass
```

4. Solve

Same bowl + step + wall geometry, only the scheme changes. The lake should remain exactly at rest.

```
if run_p3:
    domain, reference_edge_length, P0, P1, space_Q, b = \
        domain_lar, reference_edge_length_lar, P0_lar, P1_lar, space_Q_lar, b_lar
    ic_q = ic_q_lar

    swe = SWEImplicitWellbalanced(domain)
    swe.param["voellmy_mu"].value = 0.1
    swe.param["voellmy_xi"].value = 100.0
```

```
    time_end = 3.0
    CFL = 0.45

    q_lar_w, diag_lar_w = solve_time_loop(
        "lar_wb", swe, space_Q, ic_q, b, time_end,
        os.path.join(output_dir, "sim"), CFL, reference_edge_length,
    )
    print_rank0(f"LAR WB:     reached t = {diag_lar_w['t'][-1]:.4f} in {diag_lar_w['n_steps']
                f"max |hu| = {diag_lar_w['max_mom'].max():.3e}; mass drift = {diag_lar_w['mas
```

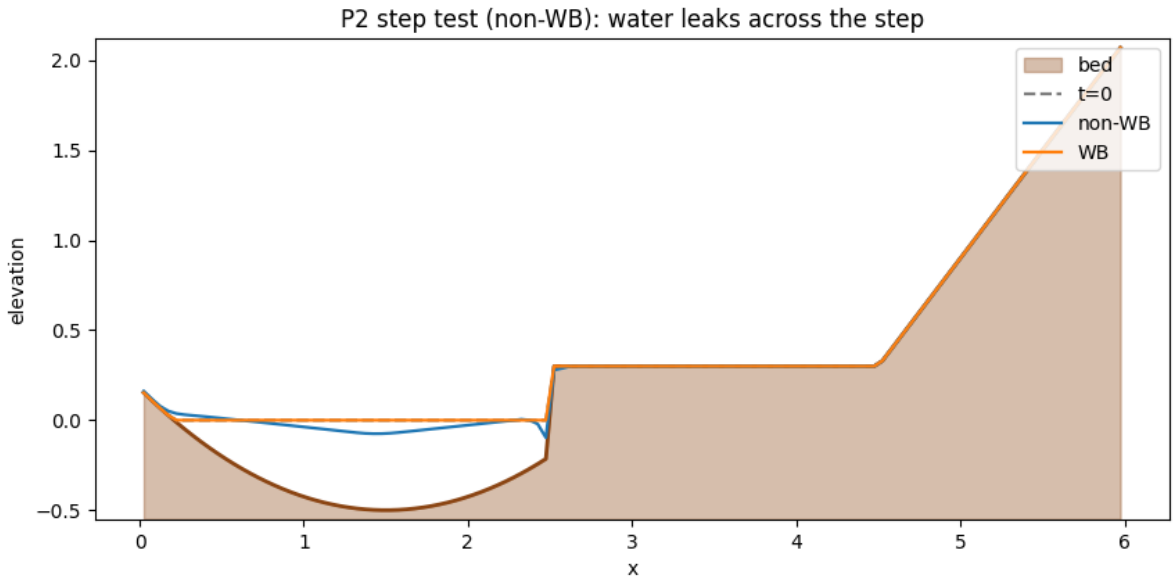
```
Setup lar_wb, solving PDE: 100%|      | 3.0082757639751914/3.0 [00:19<00:00,  6.38s/it]
```

```
LAR WB:     reached t = 3.0083 in 296 steps; max |hu| = 4.309e-20; mass drift = +2.220e-
16
```

5. Visualize

Overlay the non-WB (P2) and WB cross-sections.

```
if run_p2 and interactive_mode_is_on:
    fig, ax, x, b_arr, cells = library.plot.plot_bed_cross_section(
        domain_lar, b_lar, y_target=2.0,
        title="P2 step test (non-WB): water leaks across the step")
    library.plot.add_surface_line(ax, x, b_arr, q_ic_lar, cells, "t=0",
                                  color="tab:gray", ls="--")
    library.plot.add_surface_line(ax, x, b_arr, q_lar_i, cells, "non-WB",
                                  color="tab:blue")
    library.plot.add_surface_line(ax, x, b_arr, q_lar_w, cells, "WB",
                                  color="tab:orange")
    ax.legend(loc="upper right")
    library.plot.save_and_show(fig, output_dir, "block_B_step_cross_section")
```



6. Analysis: well-balancing verification

The WB scheme should hold the lake exactly at rest. We expect $\max|hu| \approx 0$ (machine precision), zero mass drift, and no leakage across the step.

7. Discussion

Write a brief explanation of why the naive scheme is not well-balanced and how the Audusse correction fixes it.

10 P4 — Physical application: Cascadia debris flow

With the well-balanced semi-implicit solver in hand, we apply it to a real debris-flow scenario using DEM topography from the Cascadia mountain area (Graz, Austria). The data set is from the r.avaflow simulation suite.

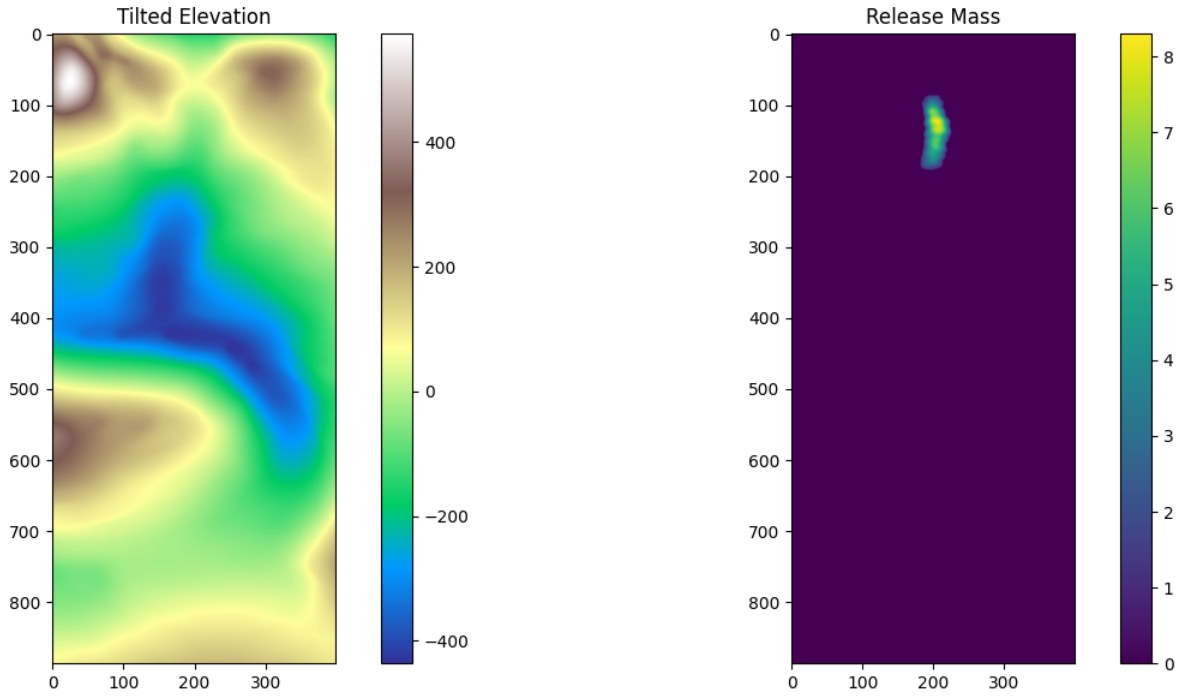
1. Load topography data

We load two raster files from the Cascadia DEM dataset: the bed elevation and the initial debris-flow release mass. Both are provided in the `library/data_hw04/` directory.

```
with rasterio.open(os.path.join(CMM_DIR, "library/data_hw04/ca_elev.tif")) as src:
    elevation = src.read(1)
with rasterio.open(os.path.join(CMM_DIR, "library/data_hw04/ca_debrisflow.tif")) as src:
    release_mass = src.read(1)

dx = 5
(Nx, Ny) = elevation.shape
elevation = elevation[:, 700:1100]
release_mass = release_mass[:, 700:1100]
(Nx, Ny) = elevation.shape

x_arr = np.arange(Nx)*dx
y_arr = np.arange(Ny)*dx
X, Y = np.meshgrid(x_arr, y_arr)
X = X.T; Y = Y.T
A_fit = np.stack([np.ones_like(X.ravel()), X.ravel(), Y.ravel()], axis=1)
coeffs, _, _, _ = np.linalg.lstsq(A_fit, elevation.ravel(), rcond=None)
a0, a1, a2 = coeffs
alpha_rad = -np.arctan(a1)
elevation_tilted = elevation - (a0 + a1 * X)

print_rank0(f"Topography shape: {elevation.shape}, inclination angle: {np.degrees(alpha_rad)
```

```
Topography shape: (887, 400), inclination angle: 25.26 degrees
```

```
if interactive_mode_is_on:
    from matplotlib import pyplot as plt
    fig, ax = plt.subplots(1, 2, figsize=(14, 6))
    im0 = ax[0].imshow(elevation_tilted, cmap="terrain", origin="upper", aspect="equal")
    ax[0].set_title("Tilted Elevation")
    fig.colorbar(im0, ax=ax[0], orientation="vertical")
    im1 = ax[1].imshow(release_mass, cmap="viridis", origin="upper", aspect="equal")
    ax[1].set_title("Release Mass")
    fig.colorbar(im1, ax=ax[1], orientation="vertical")
    plt.tight_layout()
    plt.show()
```



2. Mesh and initial conditions

`setup_cascadia_run()` loads the raster data, builds the mesh from the DEM dimensions, and interpolates both the debris-flow mass (initial `h`) and bed elevation.

```
def setup_cascadia_run():
    """Mesh + 3-component IC + bed IC for the Cascadia debris flow."""
    def ic_q(x):
        i = np.clip(np.array(x[0] / dx, dtype=int), 0, Nx-1)
        j = np.clip(np.array(x[1] / dx, dtype=int), 0, Ny-1)
        return np.array([release_mass[i, j], 0.*x[0], 0.*x[0]])

    def ic_b(x):
        # CG1 interpolation hits the domain boundary exactly, so clip
        # the raster indices to stay in range.
        i = np.clip(np.array(x[0] / dx, dtype=int), 0, Nx-1)
        j = np.clip(np.array(x[1] / dx, dtype=int), 0, Ny-1)
        return elevation_tilted[i, j]

    domain, ref_edge, P0_loc, P1_loc = create_domain(Nx-1, Ny-1, P0=[0,0], P1=[Nx*dx, Ny*dx]
```

```
    space_Q = create_function_space(domain)
    return ic_q, ic_b, domain, ref_edge, P0_loc, P1_loc, space_Q
```

3. Boundary conditions

Reflecting (slip-wall) conditions are applied on all four sides. These are handled automatically inside `SWEBase.get_weak_form` and require no extra setup here.

4. Solve

Run the `SWEImplicitWellbalanced` solver with Voellmy friction. On the cluster use `batch_hw04.sh`; if running interactively set `run_cascadia = True` above.

Voellmy parameters: dry-friction $\mu = 0.05$, turbulence coefficient $\xi = 500\,\mathrm{m/s^2}$, in the tilted coordinate system.

```
if run_cascadia:
    ic_q, ic_b, domain, reference_edge_length, P0, P1, space_Q = setup_cascadia_run()
    space_b = fem.functionspace(domain, basix.ufl.element("DG", domain.topology.cell_name(),
    b = fem.Function(space_b, name="b"); b.interpolate(ic_b)

    swe = SWEImplicitWellbalanced(domain)
    swe.param["gx"].value         = float(np.sin(alpha_rad)*9.81)
    swe.param["gz"].value         = float(np.cos(alpha_rad)*9.81)
    swe.param["voellmy_mu"].value = 0.05
    swe.param["voellmy_xi"].value = 500.0

    time_end = 200.0
    CFL = 0.45

    q_voellmy, diag_voellmy = solve_time_loop(
        "voellmy", swe, space_Q, ic_q, b, time_end,
        os.path.join(output_dir, "sim"), CFL, reference_edge_length,
    )
    library.tests.test_hw04_voellmy(q_voellmy)
    np.save(os.path.join(output_dir, 'q_sol_voellmy.npy'), q_voellmy.x.array)
```

! STUDENT TODO — Task 10

Task 10 — run the debris flow simulation on the Cascadia DEM and complete the analysis below.

1. Submit the batch job (`batch_hw04.sh`).
2. After it finishes, load the results and run the analysis cells.

5. Visualize

We load the final state from the output directory and plot the bed elevation and flow height side by side.

```
import glob

result_dirs = sorted(glob.glob(os.path.join(CMM_DIR, "results_ex_04", "run_*")))
latest_dir = result_dirs[-1] if result_dirs else output_dir
print_rank0(f"Loading from: {latest_dir}")

_domain, _ref, _P0, _P1 = create_domain(Nx-1, Ny-1, P0=[0,0], P1=[Nx*dx, Ny*dx])
_space_Q = create_function_space(_domain)
_space_b = fem.functionspace(_domain, basix.ufl.element("DG", _domain.topology.cell_name(),

fpath = os.path.join(latest_dir, "q_sol_voellmy.npy")
if os.path.exists(fpath) and interactive_mode_is_on:
    q_loaded = fem.Function(_space_Q)
    q_loaded.x.array[:] = np.load(fpath)
    b_loaded = fem.Function(_space_b)
    b_loaded.interpolate(lambda x: elevation_tilted[np.clip(np.array(x[0]/dx, dtype=int), 0,
                                                    np.clip(np.array(x[1]/dx, dtype=int), 0,
    # Bed + final h side-by-side.
    library.plot.scalar_fields_subplots(
        functions=[b_loaded, q_loaded.sub(0).collapse()],
        titles=["Cascadia bed b(x,y)", "h (final, Voellmy + WB)"],
        legend_titles=["b", "h"],
        figurepath=latest_dir, figurename="cascadia_final",
        show_edges=False,
    )
    library.plot.display_image(image_path=os.path.join(latest_dir, "cascadia_final.png"))
else:
    print_rank0("No voellmy result to display.")
```

```
Loading from: /workspaces/cmm/content/exercises/notebooks/results_ex_04
No voellmy result to display.
```

6. Analysis

Examine the run output: where did the debris flow deposit? How does the run time compare to the simulated time? Are there any cells where $h < 0$ (loss of positivity)?

11 Prepare for your presentation

During the presentation, be prepared to answer the following questions. We will ask 1–2 questions per person.

- What was the exercise about?
- Describe the Voellmy friction closure: what does μ model, what does ξ model, and where does each blow up?
- Why did P1 blow up? What is a "stiff source term", and what is the problem in conjunction with an explicit solver?
- What is the *single* method we override between `SWE` and `SWEImplicit`, and what does the change do?
- What does it mean for a scheme to be **well-balanced**? Why does P2 generate a spurious flow across the bed step?
- Describe the Audusse hydrostatic reconstruction. What is the pressure correction term added on every facet, and what physical effect does it account for?
- Which **two** methods does `SWEImplicitWellbalanced` override, and what does each one do?
- Turn the weak formulation of the advection equation from Homework 01

$$\int_\Omega \phi \frac{q^{n+1} - q^n}{\Delta t}\, dx + \int_{\partial\Omega_i} \phi \widehat{F} \cdot \mathbf{n}\, ds = 0$$

into the weak formulation with source terms used in this homework, for an explicit and a semi-implicit source term $S(q) = -c(q)\, q$ where $c > 0$.