Solving the Shallow Water Equations using Finite Volumes and Lax-Friedrichs

Fundamentals of Wave Simulation Seminar

Nathan W Brei
Fakultät für Informatik
Technische Universität München
Email: brei@in.tum.de

***Abstract*— Lax-Friedrichs and Local Lax-Friedrichs are two numerical methods for solving the shallow water equations. This paper introduces both and provides a concise implementation in Julia. The Gaussian perturbation and breaking dam model problems are solved for each method over a range of parameters. The cause and effects of artificial viscosity are explored analytically and experimentally.**

***Keywords*— Shallow water equations, Finite volumes, Lax-Friedrichs method, Rusanov, artificial viscosity, Julia**

I. Introduction

This paper introduces two finite volume methods for solving the shallow water equations. These methods are applied to solving two model problem: the Gaussian perturbation and the breaking dam. The simplest convergent method, Lax-Friedrichs, captures the desired dynamics but exhibits non-physical viscous behavior. An analysis shows that this artificial viscosity arises from the method itself. A second method, Local Lax-Friedrichs (Rusanov), is shown via analysis and experiment to improve on the former method. A concise Julia implementation is provided and described.

II. Shallow Water Equations

The shallow water equations are a system of hyperbolic partial differential equations which are effective at describing the behavior of tsunamis and overland flows. [1] They are derived from the general Navier-Stokes equations by making the following assumptions:

- In the 1-D case, the flow is in a channel of unit width.
- Horizontal velocity $u(x)$ across any cross section x is constant.
- Vertical velocity $v(x)$ is small (but not zero) and falls out of the equations.
- Pressure is dominated by hydrostatics: $p = \frac{1}{2}\rho g h^2$

The 1D equations are as follows:

$$\begin{bmatrix} h \\ hu \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \end{bmatrix}_x = 0$$

The shallow water equations may be defined for a state vector $q := (h, hu)$ in an abstract way, permitting a decoupling between the underlying equation and the numerical method. Because the shallow-water equtions match the pattern of a general conservation law,

$$q_t(x,t) + f(q(x,t))_x = 0$$

the dynamics are captured by the flux function f, where

$$f : (q_1, q_2) \mapsto \begin{bmatrix} q_2 \\ q_2^2/q_1 + \frac{1}{2}g q_1^2 \end{bmatrix}$$

This flux function is coded in lines 27–30. The shallow water equations enter the numerical code in only one other place, the max wave speed function. The system's wavespeeds are given by the eigenvalues of the flux Jacobian,

$$f'(q) = \begin{bmatrix} 0 & 1 \\ -u^2 + gh & 2u \end{bmatrix}$$

and the absolute value of the dominating eigenvalue,

$$\lambda_{max} = \max |u \pm \sqrt{gh}|$$

will be shown to play an important role in ensuring numerical stability. The corresponding implementation is provided on lines 32–37.

III. Finite Volumes

Finite volumes are a family of numerical methods which are well suited for computing an approximate solution to hyperbolic systems such as the shallow water equations. [2] They have two main advantages over methods such as finite elements. Firstly, they automatically conserve physical quantities q. Secondly, they can capture discontinuities, such as shocks, which is considerably simpler than explicitly tracking them. One downside is that the profile of the shock will be smeared by the spatial discretization.

The starting point is to recast the conservation equations in integral form:

$$\frac{d}{dt}\int_{C_i} q(x,t)dx = f(q(x_{i-1/2},t)) - f(q(x_{1+1/2},t))$$

Since these equations hold for an arbitrary control volume, we partition the spatial domain into cells, rather than points. For simplicity these cells are assumed to be regular, although the equations hold just as well over an unstructured mesh.

$$C_i := (x_{i-1/2}, x_{i+1/2})$$

After integrating with respect to time, each term takes on a physical significance directly relevant to our model:

$$\int_{C_i} q(x, t_{n+1})\,dx - \int_{C_i} q(x, t_n)\,dx = \int_{t_n}^{t_{n+1}} f(q(x_{i-1/2}, t))\,dt - \int_{t_n}^{t_{n+1}} f(q(x_{i+1/2}, t))\,dt$$

- Let $Q_i^n \approx \frac{1}{\Delta x}\int_{C_i} q(x, t_n)\,dx$ be the approximate average q in cell i at time n.
- Let $F_{i+1/2}^n \approx \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} f(q(x_{i+1/2}, t))\,dt$ be the approximate average flux f from cell i into $i+1$.

Rearranging leads to a general update scheme:

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}(F_{i+1/2}^n - F_{i-1/2}^n)$$

Ideally, the cell-wise flux function can be calculated using only the state in the adjacent cells, which yields a fully discrete method:

$$F_{i+1/2}^n = \mathcal{F}(Q_i^n, Q_{i+1}^n)$$

Taking a step back, it is clear that these also fall under the umbrella of finite difference methods. However, they have the additional property of being conservative: Even if the computed fluxes are rather inaccurate, the sum of fluxes over the spatial domain will vary only according to boundary conditions.

The desired properties of finite volumes hold for any choice of $F_{i+1/2}^n$, as long as the resulting method is consistent and stable. Similarly, they hold for any choice of hyperbolic PDE, which is abstracted away as a state space Q_i, a pointwise flux $f(Q_i)$ and max wavespeed $\lambda_{max}(Q_i)$. This natural decoupling of the mathematical machinery can be directly exploited in the implementation by parameterizing a generic finite volume skeleton with different strategies.

IV. Numerical Stability

In order to ensure numerical stability, a necessary (but not sufficient) condition that must be met is the *CFL condition*. For a hyperbolic system, information propagates through the spatial domain at finite speeds. Since the system evolves by exchanging fluxes between neighboring cells, it is intuitively necessary that the timestep size be small enough that the fluxes don't travel further than one cell per timestep. More precisely, the *numerical domain of dependence* (the set of cells whose old values factor in to the new value of each cell) must contain the *physical domain of dependence*, the region of space from which waves propagated. This is captured as a relation between cell size, timestep size, and wave speed, called the Courant number:

$$\nu := \left|\frac{\bar{u}\Delta t}{\Delta x}\right| = \frac{\Delta t}{\Delta x}|\lambda_{max}| \leq 1$$

Because this code uses the same timestep size for all cells, the maximum wave speed must be calculated across the entire spatial domain for each timestep iteration. The maximum wavespeed calculated from the flux Jacobian is exact only in the small-perturbation case, so an additional correction factor, $\mu \in [0, 1]$, scales the timestep down. Thus the experimenter has two knobs for balancing speed, accuracy, and stability: The spatial discretization Δx and the timestep correction μ.

The CFL condition is encoded on lines 95–98.

V. The Lax-Friedrichs Method

Different finite-volume methods may be derived from a common skeleton by providing a choice of $\mathcal{F}$. The simplest choice is simply the average of the fluxes at the center of either neighboring cell.

$$F_{i+1/2} = \frac{1}{2}\big[f(Q_i) + f(Q_{i+1})\big]$$

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{2\Delta x}\big[f(Q_{i+1}^n - f(Q_{i-1}^n))\big]$$

This yields a forward-time, centered-space finite difference scheme. Unfortunately a von Neumann analysis shows that it is always unstable. [3] The Lax-Friedrichs method simply modifies this scheme by changing the time difference to use a value of Q_i^n averaged from the neighboring cells:

$$Q_i^n \approx \frac{1}{2}(Q_{i-1}^n + Q_{i+1}^n)$$

The resulting update scheme becomes

$$Q_i^{n+1} = \frac{1}{2}\Big[Q_{i-1}^n + Q_{i+1}^n\Big] - \frac{\Delta t}{2\Delta x}\Big[f(Q_{i+1}^n) - f(Q_{i-1}^n)\Big]$$

This may be rearranged to yield a cell-wise flux function:

$$F_{i+1/2} = \frac{1}{2}\Big[f(Q_{i+1}) + f(Q_i) - \frac{\Delta x}{\Delta t}(Q_{i+1} - Q_i)\Big]$$

VI. Lax-Friedrichs Results

The Lax-Friedrichs flux function is implemented on lines 62–74. It was run against two model problems. The first used a Gaussian perturbation for its initial condition with reflecting boundary conditions. The second used a breaking dam initial condition with outflow boundary conditions. Two experiments were run in order to examine the behavior of the knobs n_{cells} and μ independently. These are depicted in Figures 1, 2, and 3.

The experiment depicted in Figure 1 is rather intuitive. The Gaussian perturbation propagates symmetrically across the spatial domain. As the spatial discretization is refined, a shock wave gradually emerges from each (previously symmetric) ripple. It should be noted that these shock waves are smeared across tens or hundreds of cells, far more than the finite volume method presumably requires. Furthermore, the crest is reduced and the troughs are filled in: coarsening the mesh increases dissipation.

Applying the same experiment to the breaking dam scenario, as shown in Figure 2, yields an additional, unexpected twist. As the mesh becomes coarser, a nonphysical stairstep pattern emerges. This can be directly traced back to the update rule:

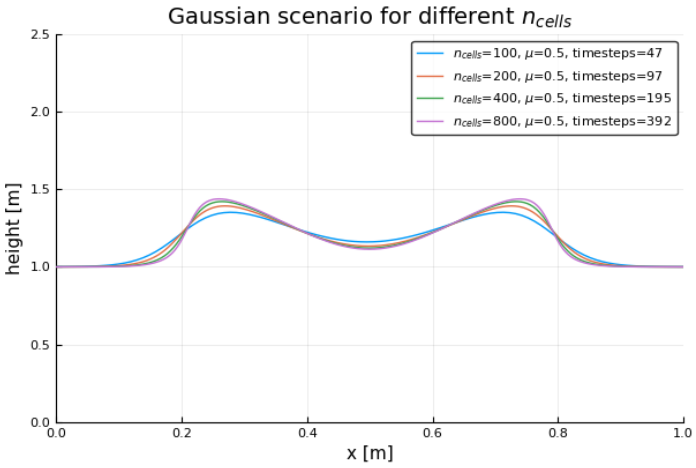

Fig. 1. LxF performance relative to n_{cells}

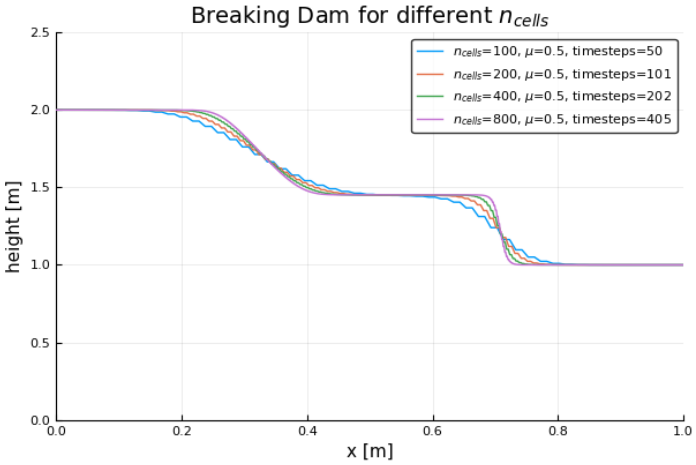

Fig. 2. LxF performance relative to n_{cells}

Q_i^{n+1} depends on Q_{i+1}^n and Q_{i-1}^n, but not Q_i^n. As a result, the odd and even grid cells evolve independently from each other and any disparity between the two oscillates without dampening. Thus the stairstep pattern did not manifest in the Gaussian scenario because the initial condition was smooth.

The final experiment considers the effect of the timestep correction μ. It uses a fine mesh in order to reduce the dissipation described previously. As defined earlier, μ is a constant factor which scales down the timestep in order to maintain stability even in the presence of large nonlinearities. However, the behavior shown is remarkably counterintuitive: As the timestep is decreased, the solution becomes more dissipative, and consequently less accurate. When $\mu = 1.0$, the shock is crisp and almost vertical; when $\mu = 0.1$, the shock is effectively lost. The next section shall explore the cause of this phenomenon by showing how viscous behavior can arise as a result of the numerical method used, rather than the underlying physical system.

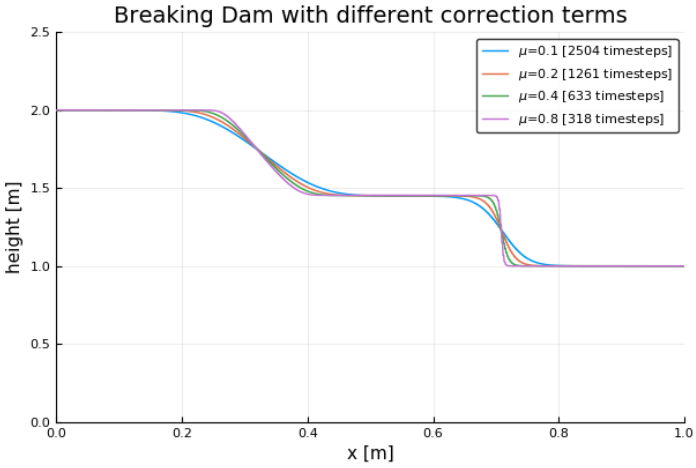

Fig. 3. LxF performance relative to μ

VII. Artificial Viscosity

The key idea behind artificial viscosity is that an approximate solution to a certain problem is oftentimes an exact solution to an approximation of that problem. The trick is to regard simple discretizatons of PDEs as being building blocks which can be assembled into more sophisticated (and correspondingly more approximate) discretizations. One can then map the sophisticated discretization directly back to a modified PDE. In the case of Lax-Friedrichs, the approximation chosen for the flux function is effectively adding a diffusive term to the shallow-water equations.

Suppose that the 'pure' solution to the shallow water equations, $q_t + f(q)_x = 0$, corresponds to the naive flux function

$$F_{i+1/2} = \frac{1}{2}\Big[f(Q_{i+1} + f(Q_i)\Big]$$

even though this is known to be unstable. Similarly, suppose that the 'pure' solution to the diffusion equation, $q_t = \beta q_{xx}$, corresponds to the flux function

$$F_{i+1/2} = -\beta\frac{Q_{i+1} - Q_i}{\Delta x}$$

This can be derived by a straightforward finite difference approximation. The Lax-Friedrichs flux function shown previously can be trivially rearranged as:

$$F_{i+1/2} = \frac{1}{2}\Big[f(Q_i) + f(Q_{i+1})\Big] - \frac{\Delta x}{2\Delta t}\Big[Q_{i+1} - Q_i\Big]$$

It is clear that this flux function pattern-matches the discretization of a different PDE:

$$q_t + f(q)_x = \frac{(\Delta x)^2}{2\Delta t} q_{xx}$$

Thus the Lax-Friedrichs parameter $a = \frac{\Delta x}{\Delta t}$ controls the diffusion coefficient β according to

$$\beta = \frac{(\Delta x)^2}{2\Delta t} = \frac{1}{2} a \Delta x$$

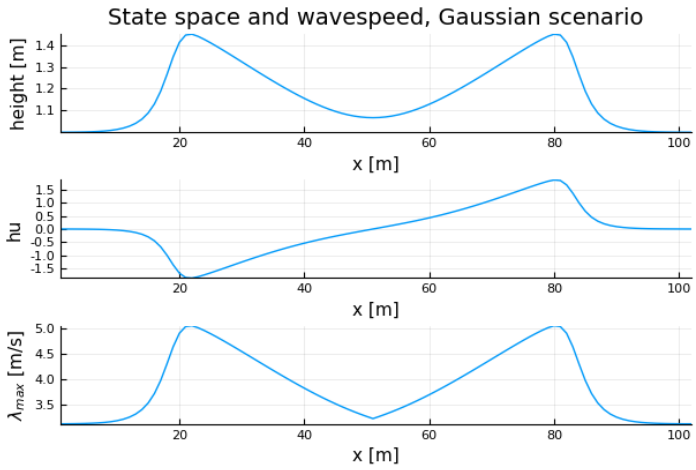

Fig. 4. Wavespeed over the spatial domain

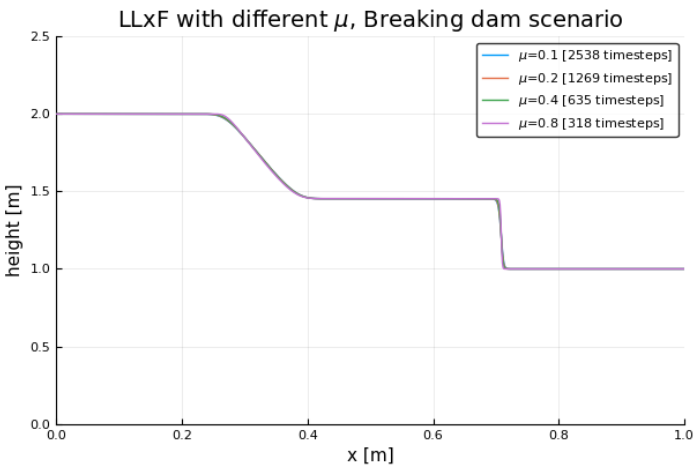

Fig. 5. Local Lax-Friedrichs, breaking dam scenario.

This explains the two results described previously. Firstly, by taking a as constant, and reducing the cell size $\Delta x \to 0$, it is clear that $\beta \to 0$ and the viscous term in the PDE vanishes. Thus the Lax-Friedrichs discretization remains consistent with the pure shallow-water equations.

Secondly, reducing the timestep size Δt increases a, which then increases β. This explains the counterintuitive phenomenon where the solution becomes increasingly dissipative, and hence less accurate, as the timestep size is decreased. This then raises the question of whether it is possible to decouple β from Δt while still maintaining a consistent and stable discretization. The method described in the following section mostly achieves this.

VIII. The Local-Lax-Friedrichs Method

Local-Lax-Friedrichs is a modification which retains the stability benefits of artificial viscosity while decreasing the resulting dissipation. In the basic Lax-Friedrichs, the viscosity parameter a is a constant derived from the timestep Δt, which in turn is derived from the CFL condition. Although the timestep is contrained to be constant over the entire spatial domain (local time stepping is not considered in the scope of this paper), the viscosity is not. Varying the viscosity over the spatial domain is advantageous: in regions with shocks or other discontinuities, the presence of viscosity improves numerical stability, and in smooth regions, the lack of viscosity leads to a less dissipative and hence more accurate solution.

The viscosity parameter a is taken to be the maximum of the wavespeeds of either neighboring cell. As shown in Figure 4, this exhibits the desired behavior of being large in regions which are discontinuous or evolving quickly, and being small elsewhere.

$$a_{i-1/2} = \max(|f'(q)|) \ \forall q \in (Q_{i-1}, Q_i)$$

$$F_{i-1/2} = \frac{1}{2}[f(Q_{i-1}) + f(Q_i) - a_{i-1/2}(Q_i - Q_{i-1})]$$

From the CFL condition, it is clear that this approach is strictly less viscous than basic Lax-Friedrichs:

$$|f'(q)| \leq \frac{\Delta x}{\Delta t}$$

The Local Lax-Friedrichs method does not entirely break the coupling between dissipation and μ, but it strongly reduces it, as depicted in Figure 5. This experiment used the exact same parameters as Figure 3, substituting only the LxF kernel (as implemented in lines 62–74) with the LLxF kernel (as implemented in lines 75–88).

IX. LxF vs LLxF Comparison

A final round of experiments were conducted to compare Lax-Friedrichs to Local Lax Friedrichs. These indicate that Local Lax-Friedrichs is decidedly less dissipative and more accurate. Figure 6 shows the results of running LLxF with a middling $\mu = 0.4$ against LxF with different μ for the breaking dam scenario.

At $\mu = 0.4$, the LLxF method performs only slightly worse than LxF at $\mu = 1.0$. For any given choice of μ, the LLxF reaches a better solution in the same number of timesteps. The quality of the solutions converge as $\mu \to 1.0$. On the other hand, LLxF requires additional work to calculate $a(x)$ for each cell, though the amount is small and linear in the number of cells.

The stairstep oscillations observed in the breaking dam case for LxF are not present for LLxf. This is presumably because the value of Q_i^n enters into the calculation of Q_i^{n+1}, albeit rather indirectly, via the max wavespeed λ_{max} and then the cell-wise flux $F_{i+1/2}^n$.

X. Implementation

This final section describes the design decisions behind the sample code provided in the appendix. The code is broken into chunks corresponding to the theoretical development in the preceding sections. It is written in Julia, a language designed for numerical programming. The choice of Julia enabled a

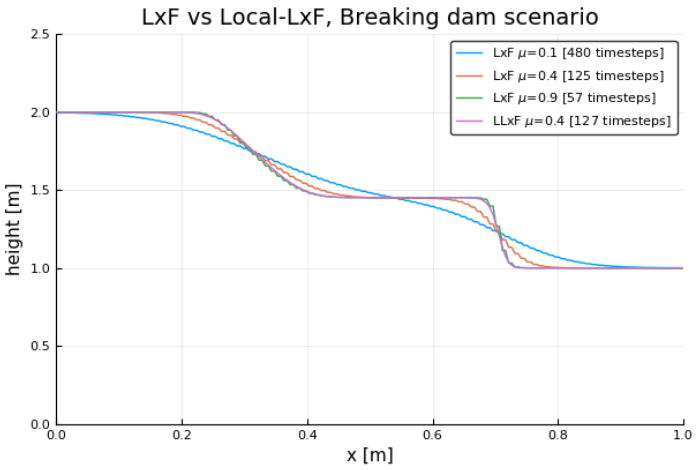

Fig. 6. Comparison of Lax-Friedrichs vs Local Lax-Friedrichs, breakin dam scenario

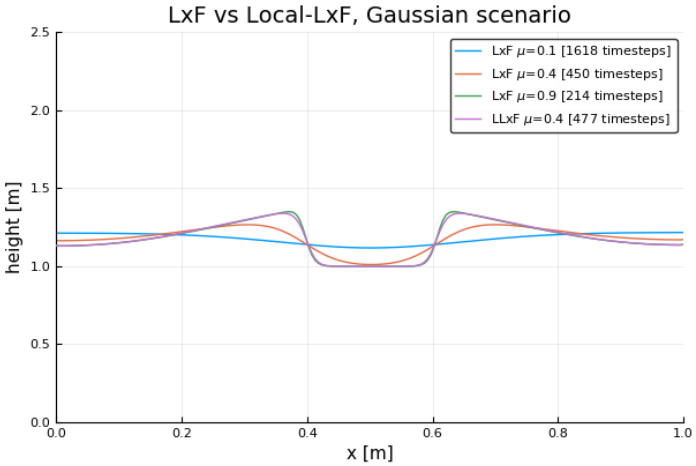

Fig. 7. Comparison of Lax-Friedrichs vs Local Lax-Friedrichs, gaussian perturbation scenario.

particularly terse code representation which still maps closely to the theory.

The first design decision is the choice of data representation. Julia eschews standard object-oriented idioms in favor of a more functional and data-driven approach. The basic composite type is a simple structure, immutable by default, with restricted inheritance and encapsulation.

The state space Q is a mutable structure. The spatial domain $q(x) \forall x$ is defined as a one-dimensional array with inner type Q. The spatial domain has a ghost cell on either side to enforce boundary conditions, so the physical cell indices range over $[2, n_{cells} + 1]$.

These additional constraints can be expressed within the type system. Firstly, by using `OffsetArrays` instead of regular `Arrays`, the indices can be cheaply translated to $[1, n_{cells}]$. Secondly, one may parameterize the spatial domain by the number of cells, `QS{ncells}`, e.g. by using `StaticArrays`, ensuring that the array size is known at JIT-compile time. This allows for array-size correctness checks and compiler optimizations. Finally, a custom constructor could encapsulate all of this and enforce the existence of the ghost cells. These features may prove helpful on larger projects, though they introduce additional complexity.

The choice of an array-of-structures representation is most consistent with the theory, but disregards the useful fact that Q forms a vector space. This problem is readily resolved thanks to *multiple dispatch*. In Julia, a function is resolved to a method by examining all of the types in the signature, rather than simply the 'owning' object. [4] This means that operations such addition can be extended to custom types in a unified way. Lines 12–26 extend addition and scalar multiplication over the state space type Q.

The basic finite volume skeleton is given on lines 38–61. The skeleton accepts abstract strategies for flux computations, timestep computations, initial conditions, and boundary conditions. These strategies are implemented as first-class functions rather than as an object heirarchy. Lines 91–93 demonstrate a higher-order function which acts like a constructor for timestep strategies. Lines 100–102 demonstrate *currying* to convert a three-argument `cfl_dt()` into a two-argument version. Lines 104–112 demonstrate using the method's *closure* to store arbitrary state, in this case the Δt calculated during the preceding function call. Finally, lines 147–148 ties everything together to run Lax-Friedrichs on a breaking dam scenario.

XI. Conclusion

The Lax-Friedrichs method is a simple finite volume technique for solving the shallow water equations. Starting from the fundamentals, this paper built up a reference implementation which leverages features of Julia to achieve terseness and clarity not available in languages such as Matlab or C++. An abstract finite volume skeleton allows different numerical methods to be swapped in. Two variants were implemented, Lax-Friedrichs and Local Lax-Friedrichs. The former suffers from excessive numerical viscosity, which the latter largely remedies. The causes and consequences of this viscosity were discussed with respect to two model problems, a Gaussian perturbation and a breaking dam. It is the author's hope that this document may serve as a concise guide to students exploring this material in the future.

References

[1] O. Delestre, F. Darboux, F. James, C. Lucas, C. Laguerre, and S. Cordier, "FullSWOF: A free software package for the simulation of shallow water flows," Mapmo, université d'Orléans ; Institut National de la Recherche Agronomique, Research Report, Jan. 2014, 38 pages. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00932234

[2] R. LeVeque, *Finite Volume Methods for Hyperbolic Problems*, ser. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2002. [Online]. Available: https://books.google.de/books?id=O_ZjpMSZiwoC

[3] G. Strang, *Computational Science and Engineering*. Wellesley-Cambridge Press, 2007. [Online]. Available: https://books.google.de/books?id=GQ9pQgAACAAJ

[4] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: https://doi.org/10.1137/141000671

XII. Appendix: Code Listing

```
# Define precision
const Float = Float64

# Define state space datatype
mutable struct Q
    h :: Float
    hu :: Float
end

# Define spatial domain datatype
QS = Array{Q,1}
```

```
# Make Q a vector space

import Base.+, Base.-, Base.*

function +(lhs::Q, rhs::Q)
    Q(lhs.h + rhs.h, lhs.hu + rhs.hu)
end

function -(lhs::Q, rhs::Q)
    Q(lhs.h - rhs.h, lhs.hu - rhs.hu)
end

function *(a::Real, q::Q)
    Q(a*q.h, a*q.hu)
end
```

```
"Shallow-water point-wise flux"
function f(q :: Q)
    Q(q.hu, q.hu^2/q.h + 0.5*G*q.h^2)
end

"Shallow-water point-wise max wave speed"
function wavespeed(q :: Q)
    u = q.hu / q.h
    c = sqrt(G * q.h)
    max(abs(u-c), abs(u+c))
end
```

```
"Finite volume skeleton"
function run(stoptime::Real, ncells::Int, choose_dt,
             compute_fluxes!, apply_bcs!, apply_ics)

    qs = apply_ics(ncells)
    Fl = QS(ncells+2)
    Fr = QS(ncells+2)

    currenttime = 0
    timesteps = 0
    dx = Float(1000.0/ncells)

    while currenttime < stoptime
        apply_bcs!(qs)
        dt = choose_dt(qs, dx)
        compute_fluxes!(qs, Fl, Fr, ncells, dx, dt)
        for x = 2:ncells+1
            qs[x] -= dt/dx * (Fr[x] + Fl[x-1])
        end
        currenttime += dt
        timesteps += 1
    end
    return (qs, currenttime, timesteps)
end
```

```
"Lax-Friedrichs kernel"
function lxf!(qs::QS, Fl::QS, Fr::QS,
              ncells::Int, dx::Float, dt::Float)

    a = dx/dt
    for x = 2:ncells+2
        ql,qr = qs[x-1], qs[x]
        fl,fr = f(ql), f(qr)

        Fr[x-1] = 0.5*((fr-fl) - a*(qr - ql))
        Fl[x-1] = 0.5*((fr-fl) + a*(qr - ql))
    end
end
```

```
"Local Lax-Friedrichs kernel"
function llxf!(qs::QS, Fl::QS, Fr::QS,
               ncells::Int, dx::Float, dt::Float)

    for x = 2:ncells+2
        ql,qr = qs[x-1], qs[x]
        fl,fr = f(ql), f(qr)

        a = max(wavespeed(ql), wavespeed(qr))

        Fr[x-1] = 0.5*((fr-fl) - a*(qr-ql))
        Fl[x-1] = 0.5*((fr-fl) + a*(qr-ql))
    end
end
```

```
# Timestepping strategies: (QS, dx) -> dt

function make_const_dt(dt::Float)
    return (qs, dx) -> dt
end

function cfl_dt(qs::QS, dx::Real, mu::Real)
    lambda = map(wavespeed,qs) |> maximum
    return Float(dx / lambda * mu)
end

function make_cfl_dt(mu::Real)
    return (qs, dx) -> cfl_dt(qs, dx, mu)
end

function make_lagging_dt(dt0::Real, mu::Real)
    dt = dt0
    function lagging_dt(qs, dx)
        result = dt
        dt = cfl_dt(qs, dx, mu)
        return result
    end
    return lagging_dt
end
```

```
# Initial conditions: ncells -> QS

function gaussian_ics(ncells::Int)
    qs = QS(ncells+2)
    mid = ncells / 2.0 + 1
    for x = 1:ncells+2
        pos_rel = (mid-x) / (ncells/10.0)
        qs[x] = Q(exp(-pos_rel^2) + 1, 0.0)
    end
    return qs
end

function breakingdam_ics(ncells::Int)
    [x < ncells/2 ? Q(2,0) : Q(1,0) for x in 1:ncells+2]
end
```

```
# Boundary conditions: QS -> nothing

function outflow_bcs!(qs::QS)
    qs[1] = qs[2]
    qs[end] = qs[end-1]
end

function periodic_bcs!(qs::QS)
    qs[1] = qs[end-1]
    qs[end] = qs[2]
end

function reflecting_bcs!(qs::QS)
    qs[1].h = qs[2].h
    qs[1].hu = -qs[2].hu
    qs[end].h = qs[end-1].h
    qs[end].hu = -qs[end-1].hu
end
```

```
# Sample experiment run
qs,t,ts = run(50, 1000, make_cfl_dt(0.5),
              lxf!, outflow_bcs!, breakingdam_ics)
```