S. Mishra
L. Scarabosio
J. Šukys

Spring Term 2014

Numerical Methods for Partial Differential Equations

ETH Zürich
D-MATH

Homework Problem Sheet 6

Introduction. This exercise sheet focuses on the stable FVM discretizations of the *non-linear* conservation laws. The first problem is the implementation of various numerical flux functions for the approximate Riemann solver and application of the methods for the one-dimensional Burgers' equation. The second problem shows how to apply many of the FVM schemes that you have already learned to a more practical setting. In particular, you will solve an example of a *non-linear multi-dimensional system* of hyperbolic conservation laws - the so-called shallow water equations. Such equations are used to model dynamics in fluids, where depth is very small, compared to the width and length of a given physical domain. In particular, flows in oceans, rivers, lakes, as well as atmospheric flows of air masses are often modeled using shallow water equations. In this problem, we will use a very simplified model of a "dambreak", in one and two space dimensions, modeling the corresponding physical 2-D and 3-D flows, respectively. Upon multiple requests, the master solution for the second problem will be provided in Python (however, MATLAB users should be able to easily understand the code, as many things are similar).

Problem 6.1 Burgers' equation: Rankine-Hugoniot condition and Riemann solvers (Core problem)

Consider the one-dimensional Burger's equation:

$$U_t + \left(\frac{U^2}{2}\right)_x = 0, \qquad \forall (x,t) \in D \times \mathbb{R}_+, \tag{6.1.1}$$

with domain $D = (-1, 3)$ and following initial conditions:

$$\textbf{(Shock)} \qquad U(x,0) = \begin{cases} 1, & \text{if} \quad x < 0, \\ 0, & \text{otherwise}, \end{cases} \tag{6.1.2}$$

$$\textbf{(Rarefaction)} \qquad U(x,0) = \begin{cases} -1, & \text{if} \quad x < 0, \\ 1, & \text{otherwise}, \end{cases} \tag{6.1.3}$$

$$\textbf{(Rarefaction and Shock)} \qquad U(x,0) = \begin{cases} 1, & \text{if} \quad 0 < x < 1, \\ 0, & \text{otherwise}. \end{cases} \tag{6.1.4}$$

(6.1a) Using *Rankine-Hugoniot conditions* (for shock waves) and *entropy conditions* (to distinguish between shocks and rarefaction waves), derive the expression for *weak entropy* solutions of (6.1.1) and all three initial conditions (6.1.2) - (6.1.4).

Solution: For a convex flux $f(\cdot)$ and Riemann initial data of the general form

$$U(x,0) = \begin{cases} U_l, & \text{if} \quad x < 0, \\ U_r, & \text{otherwise}, \end{cases} \tag{6.1.5}$$

with constant left and right states U_l and U_r.

For $U_l > U_r$ (which is the case for (6.1.2)), the solution is given as a shock

$$U(x,t) = \begin{cases} U_l & x < st, \\ U_r & \text{otherwise}, \end{cases} \tag{6.1.6}$$

moving with the wave speed s given by the Rankine-Hugoniot condition:

$$s = \frac{f(U_r) - f(U_l)}{U_r - U_l} = \frac{1}{2}.$$

For $U_l < U_r$ (which is the case for (6.1.3)), the solution is given as a rarefaction wave

$$U(x,t) = \begin{cases} -1, & \text{if} \quad x < -t, \\ x/t, & \text{if} \quad -t \leq x < t, \\ 1 & \text{otherwise}. \end{cases} \tag{6.1.7}$$

For the case (6.1.4), initially we have rarefaction wave joining state 0 and 1, starting around point $x = 0$, and shock wave separating 1 and 0, starting around point $x = 0$. The solution is given by

$$U(x,t) = \begin{cases} 0, & \text{if} \quad x < 0, \\ x/t, & \text{if} \quad 0 \leq x < t, \\ 1, & \text{if} \quad t \leq x < 1 + \frac{t}{2}, \\ 0 & \text{otherwise}. \end{cases} \tag{6.1.8}$$

At $t = 2$ the rarefaction wave meet the shock wave so solution after this time is given by a 0 left state followed by a rarefaction wave that form a shock with 0 on the right state. The shock speed for this shock is given by,

$$\sigma'(t) = \frac{\frac{1}{2}\left(\frac{\sigma(t)}{t}\right)^2}{\frac{\sigma(t)}{t}} = \frac{\sigma(t)}{2t} \qquad (t \geq 2),$$

with initial condition $\sigma(2) = 2$. This results in $\sigma(t) = (2t)^{1/2}$. So finally solution for $t \geq 2$ is given by,

$$U(x,t) = \begin{cases} 0, & \text{if} \quad x < 0, \\ x/t, & \text{if} \quad 0 \leq x < (2t)^{1/2}, \qquad (t \geq 2) \\ 0 & \text{otherwise}. \end{cases} \tag{6.1.9}$$

.

(6.1b) Implement a finite volume code for the Burgers' equation (6.1.1), using Godunov's, Roe, Lax-Friedrichs and Rusanov's numerical flux functions. Compute the numerical solution for the initial conditions (6.1.2), (6.1.3), and (6.1.4), using *outflow boundary conditions*. Plot the solutions at time $T = 1.0$ for 100 mesh cells. Explain the observed results.

HINT: *Outflow boundary conditions* can be easily implemented by extrapolating the values of the variable to the ghost cell i.e. $U_0^n = U_1^n$ and $U_{N+1}^n = U_N^n$.

Solution:

See codes 6.1 - 6.14 and figures 6.1 - 6.3 for the exact and the approximate FVM solutions.

Listing 6.1: Implementation for `solve`

```
function [ x, u, dx ] = solve( f_prime, flux, u0, l, r, T,
   mesh_size )

[ x, dx ] = getMesh( r, l, mesh_size );

u = feval(getCellAverages(u0, dx), x);

t = 0;
while (t < T)
    % calculating time step from CFL condition
    dt = 1/2 * dx / max(abs(f_prime(u)));
    t = min(t + dt, T);
    v = u;
    v = apply_outflow_bc(v);
    for j = 2:length(x)-1
        u(j) = v(j) - dt/dx * (flux(j, v, dx, dt) - flux(j-1,
           v, dx, dt));
    end
end

u = u(2:end-1);
x = x(2:end-1);

end
```

Listing 6.2: Implementation for `solve_all`

```
function [ x, u, dx] = solve_all( f, f_prime, f_type,
   f_extrema, u0, l, r, T, mesh_size )

% Godunov scheme
flux = Godunov( f, f_prime, f_type, f_extrema);
[x, u(1,:), dx] = solve(f_prime, flux, u0, l, r, T,
   mesh_size);

% Roe scheme
flux = Roe( f, f_prime, f_type, f_extrema);
```

```
[~, u(2,:), ~] = solve(f_prime, flux, u0, l, r, T, mesh_size);

% Lax-Friedrichs scheme
flux = Lax_Friedrichs( f, f_prime, f_type, f_extrema);
[~, u(3,:), ~] = solve(f_prime, flux, u0, l, r, T, mesh_size);

% Rusanov scheme
flux = Rusanov( f, f_prime, f_type, f_extrema);
[~, u(4,:), ~] = solve(f_prime, flux, u0, l, r, T, mesh_size);

end
```

Listing 6.3: Implementation for `Godunov`

```
function [ flux ] = Godunov( f, f_prime, f_type, f_extrema )

flux = @(j, u, dx, dt) flux_eval(j, u, dx, dt, f, f_prime,
   f_type, f_extrema);

end

function [flux] = flux_eval( j, u, ~, ~, f, ~, f_type,
   f_extrema )

switch f_type
    case 'convex',
        flux = max( f(max(u(j), f_extrema)), f(min(u(j+1),
           f_extrema)) );
    case 'concave',
        flux = min( f(min(u(j), f_extrema)), f(max(u(j+1),
           f_extrema)) );
    otherwise
        flux = 0;
end

end
```

Listing 6.4: Implementation for `Lax_Friedrichs`

```
function [ flux ] = Lax_Friedrichs( f, f_prime, f_type,
   f_extrema )

flux = @(j, u, dx, dt) flux_eval(j, u, dx, dt, f, f_prime,
   f_type, f_extrema);

end

function [flux] = flux_eval( j, u, dx, dt, f, ~, ~, ~ )
```

```

flux = ( f(u(j)) + f(u(j+1)) )/2 - dx/(2*dt) * ( u(j+1) - u(j) );

end
```

Listing 6.5: Implementation for `Roe`

```
function [ flux ] = Roe( f, f_prime, f_type, f_extrema )

flux = @(j, u, dx, dt) flux_eval(j, u, dx, dt, f, f_prime, f_type, f_extrema);

end

function [flux] = flux_eval( j, u, ~, ~, f, f_prime, ~, ~ )

% calculating Roe average
if abs(u(j+1) - u(j)) < eps
    Roe_avg = f_prime(u(j));
else
    Roe_avg = ( f(u(j+1)) - f(u(j)) ) / ( u(j+1) - u(j) );
end

% calculating flux
if Roe_avg >= 0
    flux = f(u(j));
else
    flux = f(u(j+1));
end

end
```

Listing 6.6: Implementation for `Rusanov`

```
function [ flux ] = Rusanov( f, f_prime, f_type, f_extrema )

flux = @(j, u, dx, dt) flux_eval(j, u, dx, dt, f, f_prime, f_type, f_extrema);

end

function [flux] = flux_eval( j, u, ~, ~, f, f_prime, ~, ~ )

flux = ( f(u(j)) + f(u(j+1)) )/2 - max( abs(f_prime(u(j))), abs(f_prime(u(j+1))) )/2 * ( u(j+1) - u(j) );

end
```

Listing 6.7: Implementation for `apply_outflow_bc`

```
function [ out ] = apply_outflow_bc( u )

out = u;

for i=1:size(u,1)
    out(i, 1)   = out(i, 2);
    out(i, end) = out(i, end - 1);
end

end
```

Listing 6.8: Implementation for `shock`

```
function [ u0 ] = shock()

u0 = @(x) x < 0;

end
```

Listing 6.9: Implementation for `rarefaction`

```
function [ u0 ] = rarefaction()

u0 = @(x) -2*(x < 0) + 1;

end
```

Listing 6.10: Implementation for `rare_shock`

```
function [ u0 ] = rare_shock()

u0 = @(x) (x > 0) .* (x < 1);

end
```

Listing 6.11: Implementation for `getMesh`

```
function [ x, dx ] = getMesh( r, l, mesh_size )

dx = abs(r-l)/mesh_size;

x = linspace(l + dx/2, r - dx/2, mesh_size);     % space discretization
x = [x(1)-dx x x(end)+dx];                       % ghost cells for bc

end
```

Listing 6.12: Implementation for `getRarefaction`

```
function [ u_rare ] = getRarefaction( x, u_l, u_r, f_prime,
   f_prime_inv, T )

u_rare = x;

for i=1:length(x)

    if (x(i) < (f_prime(u_l)*T))
        u_rare(i) = u_l;
    elseif (x(i) < (f_prime(u_r)*T))
        u_rare(i) = f_prime_inv(x(i)/T);
    else
        u_rare(i) = u_r;
    end

end

end
```

Listing 6.13: Implementation for `plotCase`

```
function [] = plotCase(
   ic_str,N,plot_res,axis_limits,T,l,r,f,f_prime,f_type,f_extrema,u0,u_e)

figure;

% solutions
[ x, u, dx ] = solve_all( f, f_prime, f_type, f_extrema, u0,
   l, r, T, plot_res );

subplot(2,2,[1 2]);
plot(x, u, '-o', x, feval(getCellAverages(u_e, dx), x));
title(strcat('Solution for', ic_str, ' initial conditions'));
xlabel('x');
ylabel('u(x,1)');
if strcmp(ic_str, ' rarefaction')
    axis([l r -1.5 1.5]);
else
    axis([l r -0.5 1.5]);
end
legend('Godunov','Roe','Lax-Friedrichs','Rusanov','Exact
   solution');

% convergence rates
[err_L1, err_Li] = geterrors (u_e, f, f_prime, f_type,
   f_extrema, u0, l, r, T, N);
```

```

subplot(2,2,3);
loglog(N, err_L1);
title(strcat('L^1 errors for', ic_str, ' initial conditions'));
legend('Godunov','Roe','Lax-Friedrichs','Rusanov');
xlabel('no. of cells');
ylabel('error');
grid on;
axis(axis_limits);

subplot(2,2,4);
loglog(N, err_Li);
title(strcat('L^\infty errors for', ic_str, ' initial conditions'));
legend('Godunov','Roe','Lax-Friedrichs','Rusanov');
xlabel('no. of cells');
ylabel('error');
grid on;
axis([1e2 2e3 1e-2 1e1]);

end
```

Listing 6.14: Implementation for `main`

```
function [] = main()

%%% === PROBLEM 1 ===

N = 100*(2.^(0:4));
plot_res = 100;
T = 1;
l = -1;
r = 3;

f = @(x) x.^2 / 2;
f_prime  = @(x) x;
f_prime_inv = @(x) x;
f_type = 'convex';
f_extrema = 0;

%%% shock initial conditions

u0 = shock();
s = 1/2;
u_e = @(x) x < s*T;

ic_str = ' shock';
```

```
axis_limits = [1e2 2e3 1e-3 1e-1];
plotCase(ic_str, N, plot_res, axis_limits, T, l, r, f, f_prime, f_type, f_extrema, u0, u_e)

print -depsc '../fig/plot_shock.eps'

%%% rarefaction initial conditions

u0 = rarefaction();
u_l = -1;
u_r = 1;
u_e = @(x) getRarefaction(x, u_l, u_r, f_prime, f_prime_inv, T);

ic_str = ' rarefaction';

axis_limits = [1e2 2e3 1e-2 2e-0];
plotCase(ic_str, N, plot_res, axis_limits, T, l, r, f, f_prime, f_type, f_extrema, u0, u_e)

print -depsc '../fig/plot_rarefaction.eps'

%%% rarefaction and shock initial conditions

u0 = rare_shock();
s = 1/2;
u_l = 0;
u_r = 1;
u_e = @(x) (x>=1).*((x-1) < s*T) + (x<1).*getRarefaction(x, u_l, u_r, f_prime, f_prime_inv, T);

ic_str = ' rarefaction and shock';

axis_limits = [1e2 2e3 0.2e-2 1e-0];
plotCase(ic_str, N, plot_res, axis_limits, T, l, r, f, f_prime, f_type, f_extrema, u0, u_e)

print -depsc '../fig/plot_rarefaction-shock.eps'

end
```

(6.1c) Use the exact solutions derived in sub-problem (6.1a) to calculate $L^1(D)$ and $L^\infty(D)$ errors. Plot $L^1(D)$ and $L^\infty(D)$ errors vs numbers of cells (for no. of cells 100, 200, 400, 800, 1600), for each case of the initial conditions in (6.1.2) - (6.1.4).

HINT: The runs of the FVM solver for meshes with 800 and 1600 cells can take several minutes.

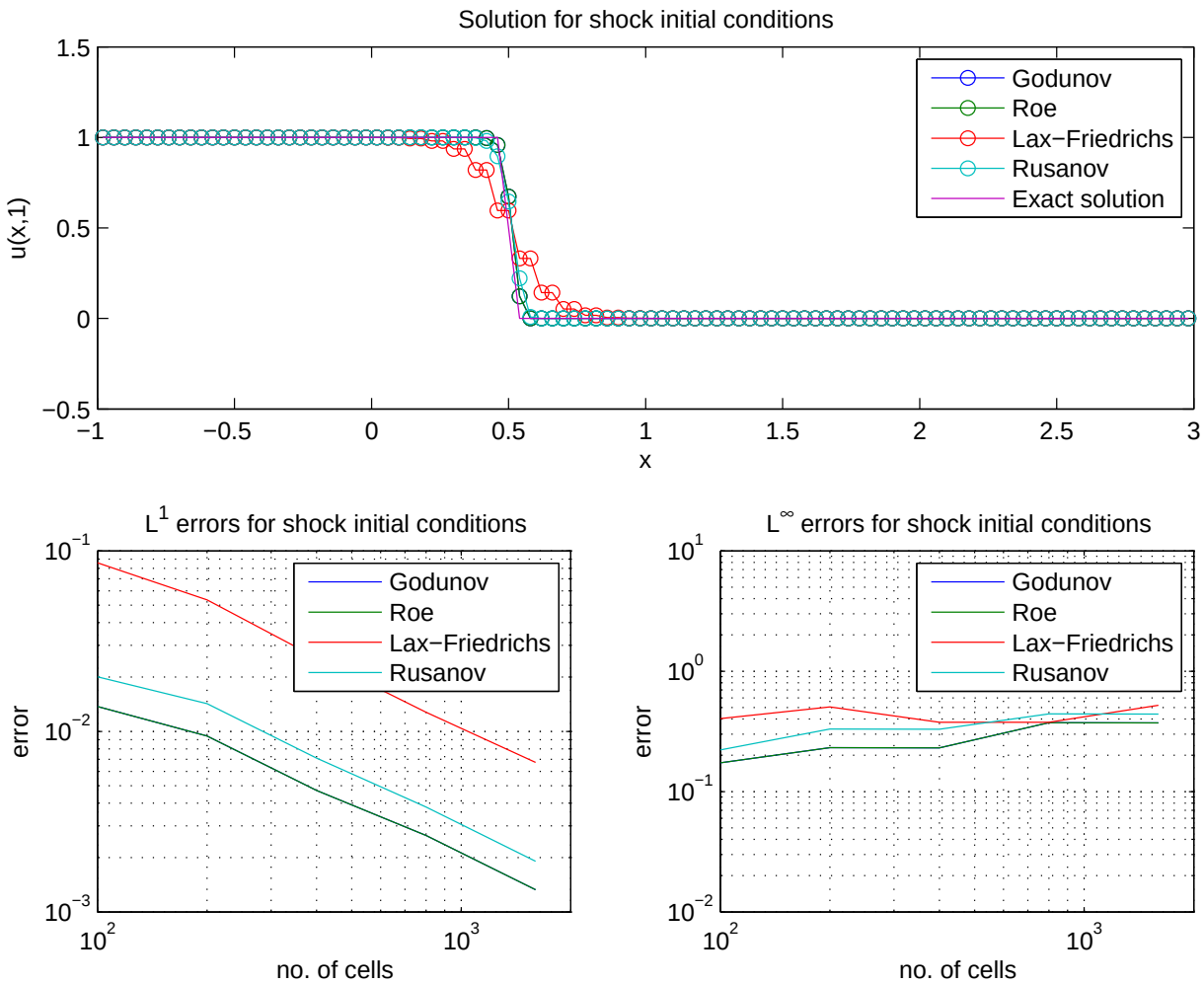

Figure 6.1: Exact solutions and FVM approximations for the shock solution (6.1.2).

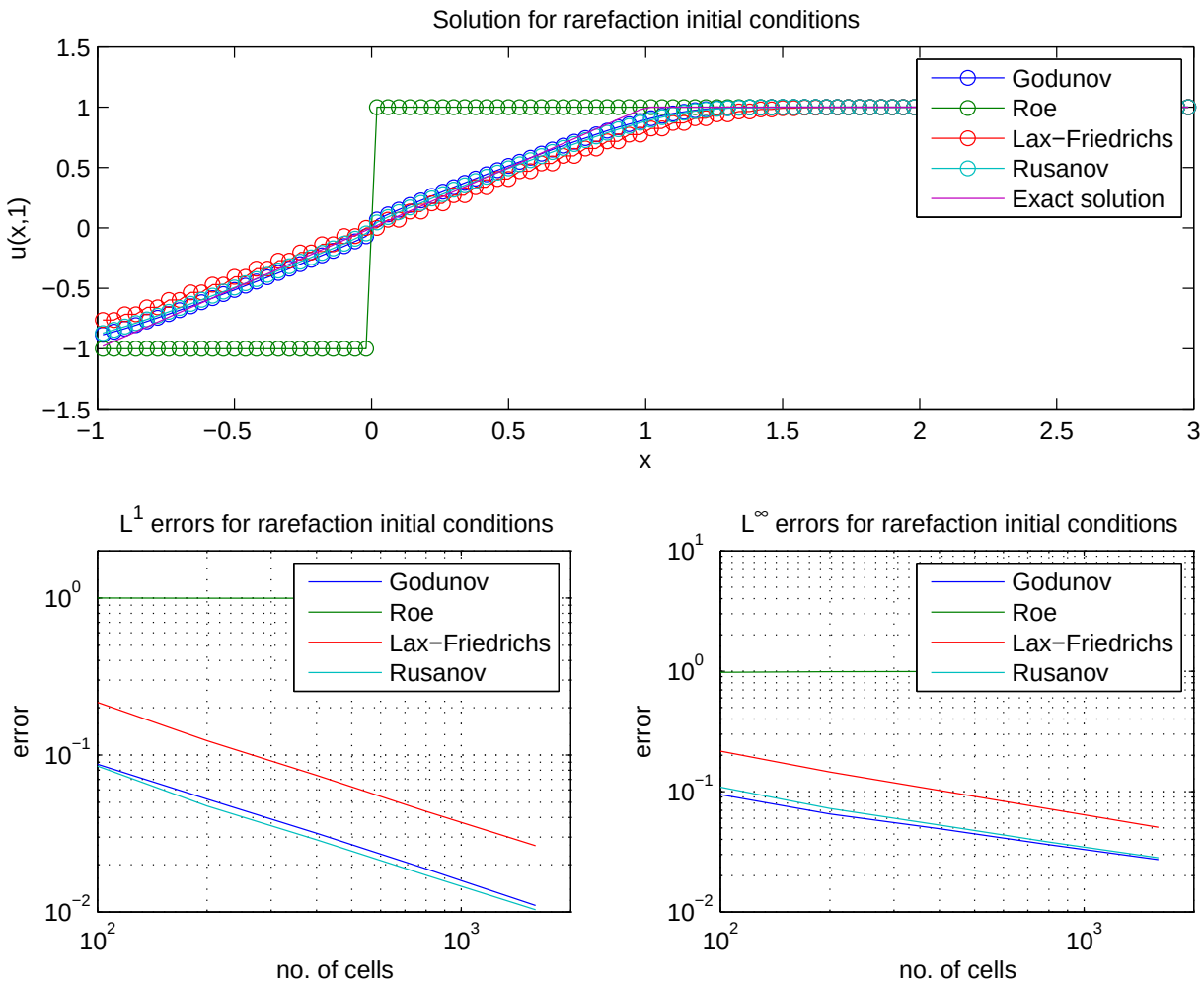

Figure 6.2: Exact solutions and FVM approximations for the rarefaction solution (6.1.3).

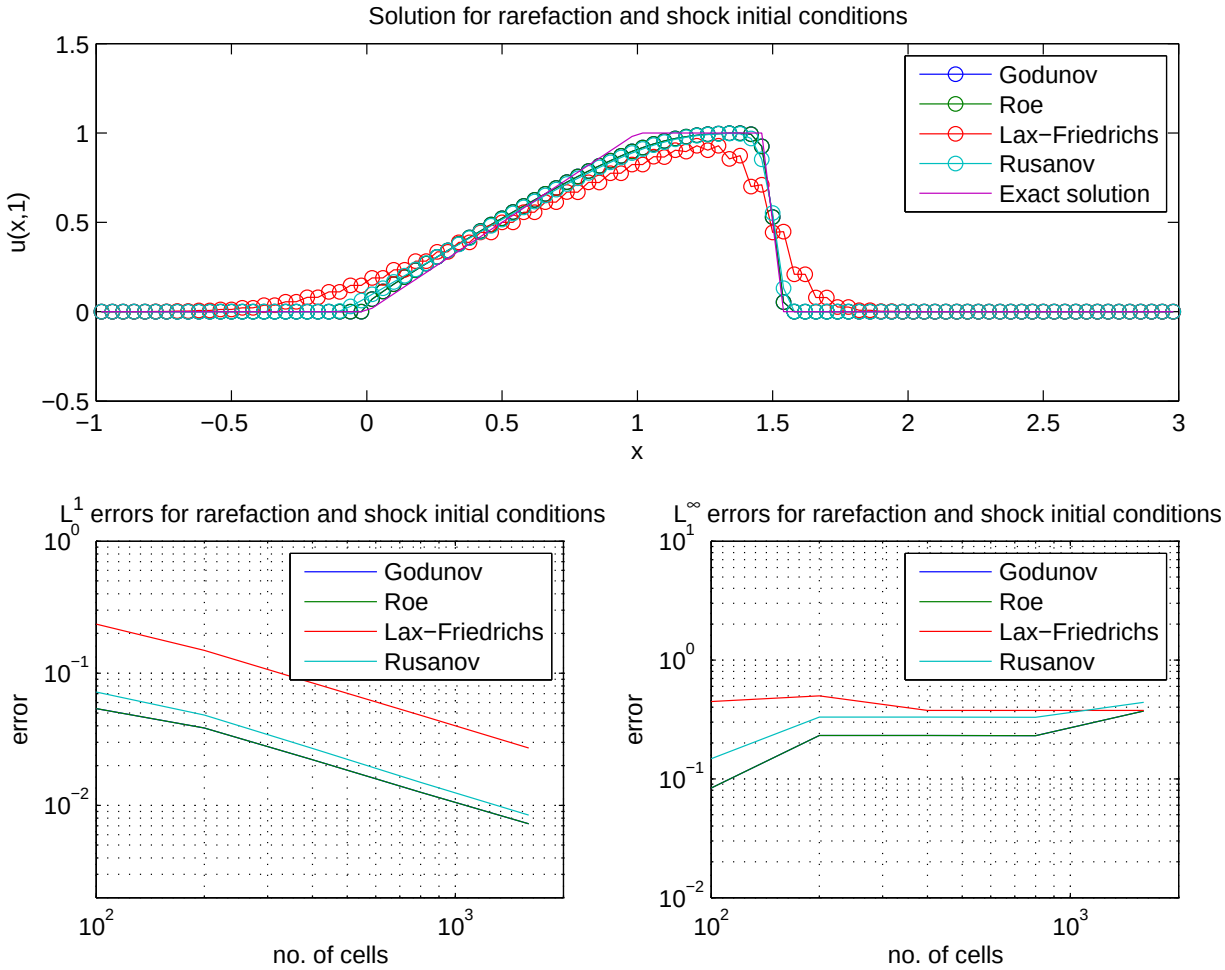

Figure 6.3: Exact solutions and FVM approximations for the rarefaction and shock solution (6.1.4).

Solution:

See the additional codes 6.15 - 6.18. For the error convergence plots, see 6.1 - 6.3.

Listing 6.15: Implementation for `geterrors`

```
function [err_L1, err_Li] = geterrors(u_e, f, f_prime,
   f_type, f_extrema, u0, l, r, T, N )

err_L1 = [];
err_Li = [];

for mesh_size=N
    [x, u, dx] = solve_all(f, f_prime, f_type, f_extrema, u0,
       l, r, T, mesh_size);
    u_avg = ones(size(u,1) , 1) * feval(getCellAverages(u_e,
       dx), x);
    err_L1 = [err_L1 get_L1(u, u_avg, dx)]; %#ok<AGROW>
    err_Li = [err_Li get_Li(u, u_avg, dx)]; %#ok<AGROW>
    display(mesh_size);
end

end
```

Listing 6.16: Implementation for `get_L1`

```
function [ err ] = get_L1( u, u_e, dx )

err = dx*sum(abs(u-u_e), 2);

end
```

Listing 6.17: Implementation for `get_Li`

```
function [ err ] = get_Li( u, u_e, ~ )

err = max(abs(u-u_e), [], 2);

end
```

Listing 6.18: Implementation for `getCellAverages`

```
function [ u_avg ] = getCellAverages( u, dx )

u_avg = @(x) avg_eval(x, u, dx);

end

function u_avg = avg_eval(x, u, dx)

```

```
u_avg = zeros(1,length(x));

for i=1:length(x)
    u_avg(i) = quad(u, x(i) - dx/2, x(i) + dx/2)/dx;
end

end
```

Problem 6.2 Shallow water equations in two dimensions

Consider the 2-dimensional shallow water equations

$$\begin{cases} h_t + (hu)_x + (hv)_y = 0, \\ (hu)_t + \left(hu^2 + \frac{1}{2}gh^2\right)_x + (huv)_y = -ghb_x, \\ (hv)_t + (huv)_x + \left(hv^2 + \frac{1}{2}gh^2\right)_y = -ghb_y. \end{cases} \tag{6.2.1}$$

Here, for a point $(x, y) \in \mathbb{R}^2$ and time instance t, variable $h(x, y, t)$ denotes the height of the fluid column above the bottom topography $b = b(x, y)$ over which the fluid flows and $(u, v) = (u(x, y, t), v(x, y, t))$ is the vertically averaged (or depth averaged) horizontal fluid velocity field. The constant g denotes the size of the negative vertical acceleration due to gravity and is set to $g = 9.812$ here. For now, we also consider flat bottom topography, i.e. we set $b \equiv \text{const} = 0$, and hence the right hand side of (6.2.1) becomes zero.

Denoting the vectors of conserved variables (h, hu and hv) as $\mathbf{U} = \mathbf{U}(\mathbf{x}, t) : \mathbb{R}^2 \times \mathbb{R}_+ \to \mathbb{R}^3$, and the directional (in x and y directions) fluxes as $\mathbf{F}, \mathbf{G} : \mathbb{R}^3 \to \mathbb{R}^3$, i.e.

$$\mathbf{U} = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}, \quad \mathbf{G} = \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}, \tag{6.2.2}$$

the system (6.2.1) with given initial data $\mathbf{U}_0$ is rewritten as a *system of conservation laws*,

$$\begin{cases} \mathbf{U}(\mathbf{x}, t)_t + \mathbf{F}(\mathbf{U})_x + \mathbf{G}(\mathbf{U})_y = 0, \\ \mathbf{U}(\mathbf{x}, 0) = \mathbf{U}_0(\mathbf{x}). \end{cases} \quad \mathbf{x} = (x, y) \in \mathbf{D}, \quad t > 0, \tag{6.2.3}$$

where we restrict the computational domain to some bounded Cartesian domain $\mathbf{D} \subset \mathbb{R}^2$.

The maximum directional wave speeds (in directions x and y), corresponding to the maximal eigenvalues of the matrices $\partial\mathbf{F}/\partial\mathbf{U}$ and $\partial\mathbf{G}/\partial\mathbf{U}$, are given by the

$$\lambda_x(\mathbf{U}) = |u| + \sqrt{gh}, \quad \lambda_y(\mathbf{U}) = |v| + \sqrt{gh}. \tag{6.2.4}$$

(6.2a) As a starting point, we first consider the one-dimensional version of the shallow water equations, obtained by setting $v \equiv 0$ in (6.2.1), removing flux $\mathbf{G}$, and denoting $\mathbf{U} = (h, hu)^\top$,

$$\begin{cases} h_t + (hu)_x = 0, \\ (hu)_t + \left(hu^2 + \frac{1}{2}gh^2\right)_x = 0. \end{cases} \tag{6.2.5}$$

For the finite domain $\mathbf{D} = I_1 = (0, 2)$, implement the Finite Volume solver using the Rusanov numerical flux and the Forward Euler time stepping, i.e.

$$\mathbf{U}_i^{n+1} := \mathbf{U}_i^n - \frac{\Delta t}{\Delta x}(\mathbf{F}_{i+\frac{1}{2}}^n - \mathbf{F}_{i-\frac{1}{2}}^n), \tag{6.2.6}$$

where $\mathbf{U}_i^n$ are the vectors $(h_i^n, (hu)_i^n)^\top$ denoting the cell averages of the solution $\mathbf{U} = (h, hu)^\top$, and $\mathbf{F}_{i+\frac{1}{2}}^n$ are approximated by the Rusanov flux function

$$\mathbf{F}_{i+\frac{1}{2}}^n(\mathbf{U}_i^n, \mathbf{U}_{i+1}^n) \approx \mathbf{F}^{\text{Rus}}(\mathbf{U}_L, \mathbf{U}_R) = \frac{(\mathbf{F}_L + \mathbf{F}_R)}{2} - \frac{\lambda_{\max}}{2}(\mathbf{U}_R - \mathbf{U}_L), \tag{6.2.7}$$

with *local* maximum wave speeds

$$\lambda_{\max} := \max(\lambda_x(\mathbf{U}_L), \lambda_x(\mathbf{U}_R)).$$

The mesh width is $\Delta x = |I_1|/N_x$, where N_x denotes the number of the mesh cells, and the time step size respects the CFL condition (with CFL number C_{CFL} set to 0.9)

$$\Delta t = C_{\text{CFL}} \frac{\Delta x}{\bar{\lambda}_x} \leq \frac{\Delta x}{\bar{\lambda}_x} = \frac{\Delta x}{\max_i \lambda_x(\mathbf{U}_i)}. \tag{6.2.8}$$

Use the "outflow" boundary conditions, which can be easily implemented by extrapolating the values of the variable to the ghost cell i.e. $\mathbf{U}_0^n = \mathbf{U}_1^n$ and $\mathbf{U}_{N+1}^n = \mathbf{U}_N^n$.

Run your code for the "dambreak" initial data given by

$$\mathbf{U}_0(x) = \Big(h_0(x), u_0(x)\Big)^\top = \begin{cases} (2 - b(x), 0)^\top, & \text{if} \quad x < 1, \\ (1.5 - b(x), 0)^\top, & \text{otherwise}, \end{cases} \tag{6.2.9}$$

for $N_x = 512$ mesh cells up to final time $T = 0.1$ and plot the results (both water column height h and velocity u).

HINT: Consider the evaluations of $\mathbf{U}$ at the cell mid-points x_i for the cell averages $\mathbf{U}_i$.

HINT: Implement the Rusanov flux (6.2.7) as a separate function; this way you will be able to quickly extend and adapt it for the 2-dimensional FVM solver in the next sub-problem.

HINT: The solution consists of the left-moving rarefaction wave and the right-moving shock wave, as depicted in Figure 6.4.

HINT: It is advisable to implement and use the conversion functions between the *observable* (also called *primitive*) variables (h, u) and the *conserved* variables (h, hu).

HINT: You might find the MATLAB function `diff` or the Python function `numpy.diff` useful.

HINT: For debugging, it is always a good idea to plot the initial data and the solution after *one* time step, and verify that your numerical flux is correct and provides stable approximations.

Solution:

See codes 6.19 - 6.22, ignoring the code lines dealing with bottom topography, and Figure 6.4 for the approximate FVM solutions.

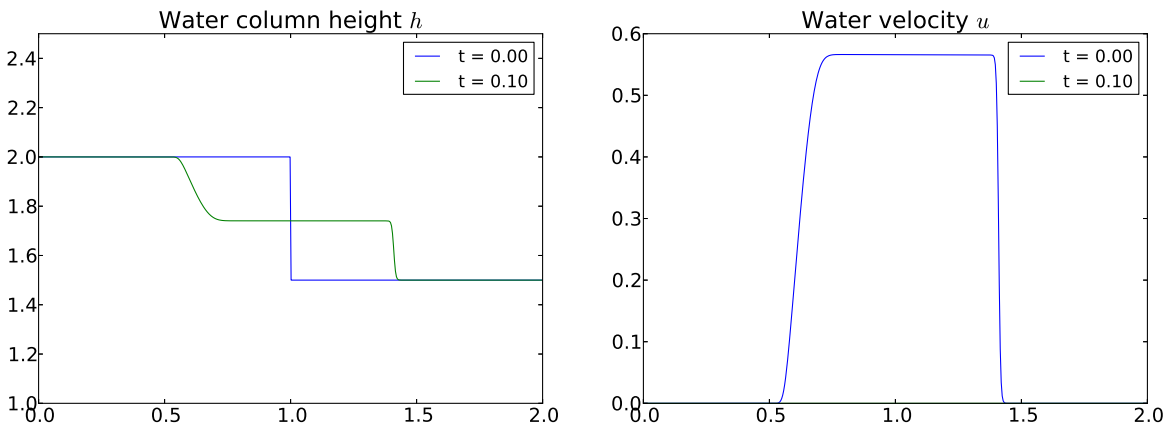

Figure 6.4: FVM approximations of the 1-D shallow water equations (6.2.5) and the dambreak problem (6.2.9) with flat bottom topography $b \equiv 0$.

Listing 6.19: Implementation for `shallow_water_1d`

```
# global imports
import numpy
import pylab

# local imports
from rusanov import rusanov_flux
from compute_flux_1d import compute_flux_1d
from variables import ConservedVars, PrimitiveVars

# plotting configuration
import matplotlib
matplotlib.rcParams['font.size'] = 18
matplotlib.rcParams['legend.fontsize'] = 16

def shallow_water_1d (D, g, T, h0, u0, F, DF, N, num_flux,
    CFL, b):

  # mesh width
  dx = numpy.fabs ( numpy.diff(D) ) / N

  # cell mid-points
  xr = numpy.linspace (D[0] + dx/2, D[1] - dx/2, N)

  # cell averages of the initial data
  h = h0 (xr)
  u = u0 (xr)

  # modify 'h', if bottom topography is present
  if b != None:
```

```
    h -= b(xr)

  # convert to conserved variables
  [h, hu] = ConservedVars (h, u)

  # cell averages of the bottom topography (if present)
  # and the 2nd order accurate approximation of derivative DB of B
  # where at the boundary (first and last cells), we set DB = 0
  if b != None:
    B = b(xr)
    DB = (B[2:] - B[:-2]) / (2 * dx)
    DB = numpy.insert(DB, 0, [0])
    DB = numpy.append(DB, [0])

  # Forward Euler time-stepping
  t = 0
  while t < T:

    # display current time 't'
    print t

    # compute the numerical flux
    [Fh, Fhu, lmax] = compute_flux_1d (h, hu, F, DF, g, num_flux)

    # compute the time step size
    dt = min (T - t, CFL * dx / lmax)

    # compute source terms, if bottom topography is present
    if b != None:
      S = - g * h * DB

    # perform time integration
    h  -= dt/dx * numpy.diff (Fh)
    hu -= dt/dx * numpy.diff (Fhu)

    # add source terms, if present
    if b != None:
      hu += dt * S

    # update the time step size
    t += dt;

  # return the solution
  return [xr] + PrimitiveVars (h, hu)
```

```python

if __name__ == '__main__':

  # problem data
  D = [0, 2]
  T = 0.1
  g = 8.912
  h0 = lambda x : numpy.where (x < 1, 2.0, 1.5)
  u0 = lambda x : numpy.zeros (x.shape)
  F  = lambda h, u, g : [h*u, h*u*u + 0.5*g*h*h]
  DF = lambda h, u, g : numpy.fabs(u) + numpy.sqrt(g*h)

  # discretization parameters
  N = 512
  CFL = 0.9

  # run the FVM scheme
  [xr, h, u] = shallow_water_1d (D, g, T, h0, u0, F, DF, N,
     rusanov_flux, CFL, None)

  # plot the results
  pylab.figure (figsize=(2*8,6))
  pylab.subplot (121)
  pylab.plot (xr, h0(xr), label='t = %.2f' % 0)
  pylab.plot (xr, h, label='t = %.2f' % T)
  pylab.legend (loc='upper right')
  pylab.ylim([1, 2.5])
  pylab.title (r'Water column height $h$')
  pylab.subplot (122)
  pylab.plot (xr, u, label='t = %.2f' % 0)
  pylab.plot (xr, u0(xr), label='t = %.2f' % T)
  pylab.legend (loc='upper right')
  pylab.title (r'Water velocity $u$')
  pylab.subplots_adjust(left=0.05,right=0.95)
  pylab.savefig ('../fig/shallow_water_1d.eps')
  pylab.show()

  # run the FVM scheme with bottom topography
  b  = lambda x : 0.1*numpy.sin(5*numpy.pi*x) - 0.2*x + 1.4
  [xr, h, u] = shallow_water_1d (D, g, T, h0, u0, F, DF, N,
     rusanov_flux, CFL, b)

  # plot the results
  pylab.figure (figsize=(2*8,6))
  pylab.subplot (121)
  pylab.plot (xr, h0(xr), label='t = %.2f' % 0)
  pylab.plot (xr, h + b(xr), label='t = %.2f' % T)
```

```
  pylab.plot (xr, b(xr), color='k', label='topography')
  pylab.fill_between(xr, b(xr), 0, color=(0.5, 0.5, 0.5))
  pylab.legend (loc='upper right')
  pylab.ylim([1, 2.5])
  pylab.title (r'Water column height $h$')
  pylab.subplot (122)
  pylab.plot (xr, u, label='t = %.2f' % 0)
  pylab.plot (xr, u0(xr), label='t = %.2f' % T)
  pylab.legend (loc='upper right')
  pylab.title (r'Water velocity $u$')
  pylab.subplots_adjust(left=0.05,right=0.95)
  pylab.savefig ('../fig/shallow_water_1d_topography.eps')
  pylab.show()
```

Listing 6.20: Implementation for `variables`

```
# convert primitive variables (h, u, v) to conserved
   variables (h, hu, hv)
def ConservedVars (h, u, v=None):
  return [h, h*u, h*v] if v != None else [h, h*u]

# convert conserved variables (h, hu, hv) to primitive
   variables (h, u, v)
def PrimitiveVars (h, hu, hv=None):
  return [h, hu/h, hv/h] if hv != None else [h, hu/h]
```

Listing 6.21: Implementation for `rusanov`

```
# Rusanov flux
def rusanov_flux (FL, FR, lmax, UR, UL):
  return 0.5 * (FL  + FR)  - 0.5 * lmax * (UR  - UL)
```

Listing 6.22: Implementation for `compute_flux_1d`

```
import numpy
from variables import ConservedVars, PrimitiveVars

# compute all fluxes using the numerical flux 'num_flux'
def compute_flux_1d (h, hu, F, DF, g, num_flux):

  # get number of cells and increment by 1
  N = h.shape[0] + 1

  # allocate arrays for flux
  Fh  = numpy.zeros (N)
  Fhu = numpy.zeros (N)

  # allocate array for the wave speeds
  lmax = numpy.zeros (N)
```

```
  # compute primitive variables
  [h, u] = PrimitiveVars (h, hu)

  # compute fluxes at each cell interface
  for i in xrange(0, N):

    # set left and right indices
    # incorporating 'outflow' boundary conditions
    L = max (0, i - 1)
    R = min (i, N - 2)

    # compute the maximum wave speed
    lmax[i] = max ( DF (h[L], u[L], g), DF (h[R], u[R], g) )

    # compute the left and the right fluxes
    [FhL, FhuL] = F (h[L], u[L], g)
    [FhR, FhuR] = F (h[R], u[R], g)

    # compute the Rusanov flux
    Fh[i]  = num_flux (FhL,  FhR,  lmax[i], h [R], h [L])
    Fhu[i] = num_flux (FhuL, FhuR, lmax[i], hu[R], hu[L])

  # compute the global maximal wave speed
  lmax = numpy.max(lmax)

  # return the numerical flux and the global maximal wave
     speed
  return [Fh, Fhu, lmax]
```

(6.2b) Extend your code of subproblem (6.2a) to non-constant bottom topography, given by

$$b(x) = 0.1\sin(5\pi x) - 0.2x + 1.4. \tag{6.2.10}$$

The resulting FVM scheme then also needs to incorporate the source term $\mathbf{S}_i$,

$$\mathbf{U}_i^{n+1} := \mathbf{U}_i^n - \frac{\Delta t}{\Delta x}(\mathbf{F}_{i+\frac{1}{2}}^n - \mathbf{F}_{i-\frac{1}{2}}^n) + \Delta t \mathbf{S}_i, \tag{6.2.11}$$

which you can approximate by using second order accurate central differences,

$$\mathbf{S}_i = \begin{bmatrix} 0 \\ gh_i \frac{B_{i+1} - B_{i-1}}{2\Delta x} \end{bmatrix}. \tag{6.2.12}$$

I order to deal with the boundary conditions for b, simply set

$$\mathbf{S}_0 = (0,0)^\top, \quad \mathbf{S}_{N_x} = (0,0)^\top. \tag{6.2.13}$$

Run your code for the "dambreak" initial data (6.2.9) for $N_x = 512$ mesh cells up to final time $T = 0.1$ and plot the results (plot water surface $h + b$ and bottom topography b in the same plot).

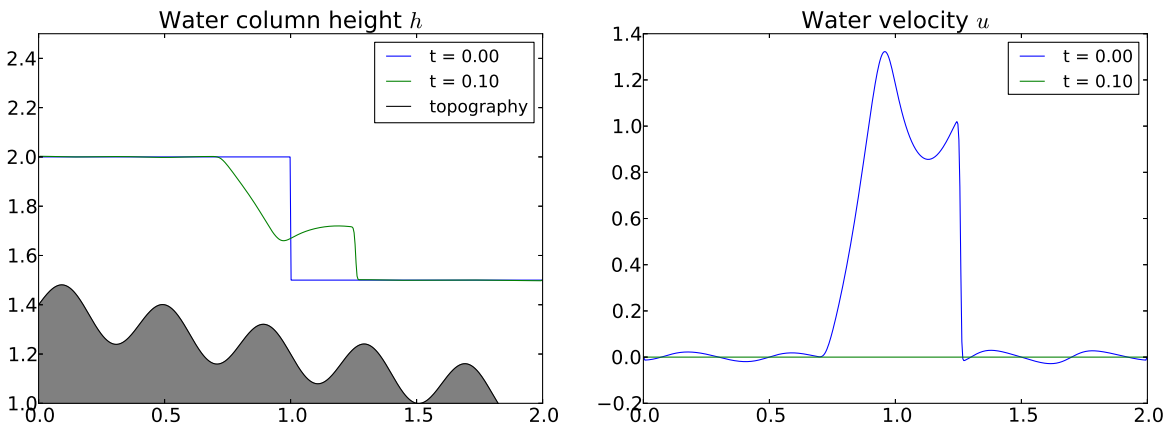

Figure 6.5: FVM approximations of the 1-D shallow water equations (6.2.5) and the dambreak problem (6.2.9) with varying bottom topography (6.2.10).

HINT: Do not forget to *subtract* bottom topography $b(x)$ from $h(x)$ in (6.2.9).

HINT: For plotting, do not forget to *add* bottom topography b to water column h to get the actual water surface.

HINT: The solution, analogously as in subproblem (6.2a), consists of the left-moving rarefaction wave and the right-moving shock wave, as depicted in Figure 6.5.

Solution:

See codes 6.19 - 6.22 and Figure 6.5 for the approximate FVM solutions.

(6.2c) In this sub-problem, we extend the 1-dimensional Finite Volume scheme from the subproblem (6.2a) with flat bottom topography $b \equiv 0$ to the 2-dimensional shallow water equations (6.2.3). Consider the finite domain $\mathbf{D} = I_1 \times I_2 = (0,2) \times (0,1)$ and the corresponding uniform axiparallel equidistant mesh with N_x cells in x direction and N_y cells in y direction. Implement the Finite Volume solver using the dimension splitting, Rusanov numerical flux and the Forward Euler time stepping, i.e.

$$\mathbf{U}_{i,j}^{n+1} := \mathbf{U}_{i,j}^{n} - \frac{\Delta t}{\Delta x}(\mathbf{F}_{i+\frac{1}{2},j}^{n} - \mathbf{F}_{i-\frac{1}{2},j}^{n}) - \frac{\Delta t}{\Delta y}(\mathbf{G}_{i,j+\frac{1}{2}}^{n} - \mathbf{G}_{i,j-\frac{1}{2}}^{n}), \tag{6.2.14}$$

where $\mathbf{U}_{i,j}^{n}$ are the vectors $(h_{i,j}^{n}, (hu)_{i,j}^{n}, (hv)_{i,j}^{n})^\top$ denoting the cell averages of the solution $\mathbf{U} = (h, hu, hv)^\top$, and the *directional* fluxes $\mathbf{F}_{i+\frac{1}{2},j}^{n}$ and $\mathbf{G}_{i,j+\frac{1}{2}}^{n}$ are approximated by the Rusanov flux function from (6.2.7) in the *corresponding direction*, i.e.

$$\mathbf{F}_{i+\frac{1}{2},j}^{n} \approx \mathbf{F}^{\text{Rus}}(\mathbf{F}_{i,j}^{n}, \mathbf{F}_{i+1,j}^{n}), \quad \mathbf{G}_{i,j+\frac{1}{2}}^{n} \approx \mathbf{F}^{\text{Rus}}(\mathbf{F}_{i,j}^{n}, \mathbf{F}_{i,j+1}^{n}). \tag{6.2.15}$$

The time step size respects the 2-dimensional version of the CFL condition (6.2.8)

$$\Delta t = C_{\text{CFL}}\left(\frac{\bar{\lambda}_x}{\Delta x} + \frac{\bar{\lambda}_y}{\Delta y}\right)^{-1}, \quad \bar{\lambda}_x = \max_{i,j} \lambda_x(\mathbf{U}_{i,j}), \quad \bar{\lambda}_y = \max_{i,j} \lambda_y(\mathbf{U}_{i,j}). \tag{6.2.16}$$

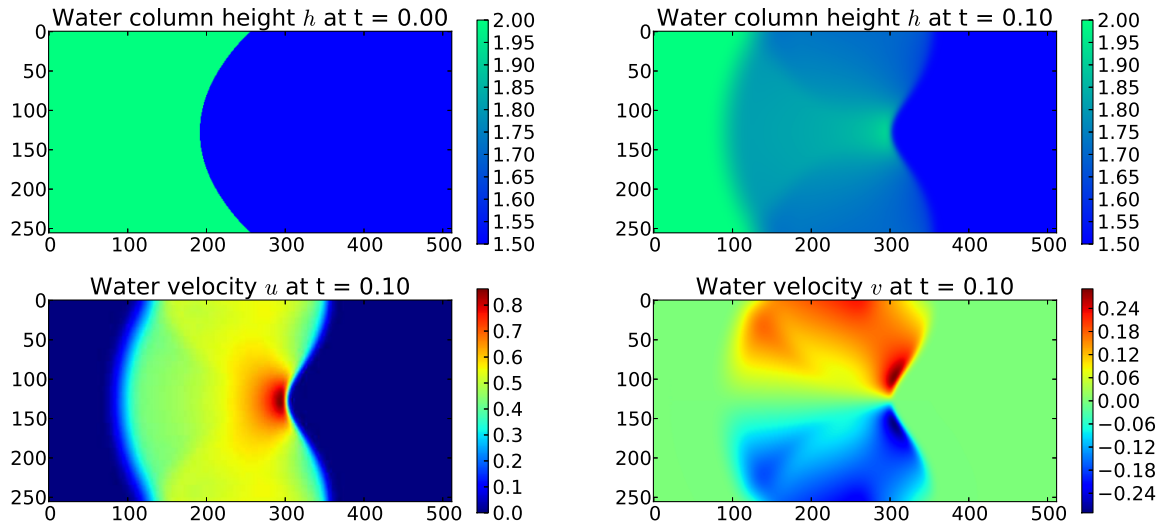

Figure 6.6: FVM approximations of the 2-D shallow water equations (6.2.1) and the dambreak problem (6.2.17).

Run your code for the 2-dimensional "dambreak" initial data given by

$$\mathbf{U}_0(x,y) = \Big(h_0(x,y), u_0(x,y), v_0(x,y)\Big)^\top = \begin{cases} (2-b(x,y),0,0)^\top, & \text{if } x < (y-0.5)^2 + 0.75, \\ (1.5-b(x,y),0,0)^\top, & \text{otherwise,} \end{cases} \tag{6.2.17}$$

for $N_x = 64$ and $N_y = 32$ mesh cells up to final time $T = 0.1$ and plot the results (both water column height h and velocities u and v). You might also want (this is optional) to generate a 3D plot with the water surface h being plotted on the z-axis. Use the "outflow" boundary conditions, which can be easily implemented by extrapolating the values of the variable to the ghost cell, i.e.

$$\mathbf{U}^n_{0,j} = \mathbf{U}^n_{1,j}, \quad \mathbf{U}^n_{N+1,j} = \mathbf{U}^n_{N,j} \tag{6.2.18}$$

in the x direction (i.e. for the evaluations of the flux $\mathbf{F}$), and

$$\mathbf{U}^n_{i,0} = \mathbf{U}^n_{i,1}, \quad \mathbf{U}^n_{i,N+1} = \mathbf{U}^n_{i,N}. \tag{6.2.19}$$

in the y direction (i.e. for the evaluations of the flux $\mathbf{G}$).

HINT: You might find the MATLAB function `meshgrid` or the Python function `meshgrid` (with option `indexing='ij'`) from `numpy` useful.

HINT: If you are eager to test the limits of your machine and prepared to wait longer (an hour or so), run your 2-D code with $N_x = 512$ and $N_y = 256$. For even larger resolutions (and hence smaller errors), parallel implementations are usually needed.

HINT: For code testing purposes, the reference plots (using $N_x = 512$ and $N_y = 256$) are given in Figure 6.6.

Solution:

See codes 6.23 - 6.24 and Figure 6.6 for the approximate FVM solutions.

Listing 6.23: Implementation for `shallow_water_2d`

```python
# global imports
import numpy
import pylab

# local imports
from rusanov import rusanov_flux
from compute_flux_2d import compute_flux_2d
from variables import ConservedVars, PrimitiveVars

# plotting configuration
import matplotlib
matplotlib.rcParams['font.size'] = 18
matplotlib.rcParams['legend.fontsize'] = 16

# constants for 'x' and 'y' directions
X = 0
Y = 1

def shallow_water_2d (D, g, T, h0, u0, v0, F, G, DF, DG, N,
    num_flux, CFL):

  # mesh widths
  dx = numpy.fabs ( numpy.diff(D[X]) ) / N[X]
  dy = numpy.fabs ( numpy.diff(D[Y]) ) / N[Y]

  # cell mid-points
  xr = numpy.linspace (D[X][0] + dx/2, D[X][1] - dx/2, N[X])
  yr = numpy.linspace (D[Y][0] + dy/2, D[Y][1] - dy/2, N[Y])

  # generate the mesh grid
  [xg, yg] = numpy.meshgrid(xr, yr, indexing='ij')

  # cell averages of the initial data
  h = h0 (xg, yg)
  u = u0 (xg, yg)
  v = v0 (xg, yg)
  [h, hu, hv] = ConservedVars (h, u, v)

  # Forward Euler time-stepping
  t = 0
  while t < T:

    # display current time 't'
    print t

    # compute the numerical fluxes
```

```
    [Fh, Fhu, Fhv, lmax_x] = compute_flux_2d (h, hu, hv, F, DF, g, num_flux, X)
    [Gh, Ghu, Ghv, lmax_y] = compute_flux_2d (h, hu, hv, G, DG, g, num_flux, Y)

    # compute the time step size
    dt = min ( T - t, CFL * (lmax_x / dx + lmax_y / dy)**(-1) )

    # perform time integration
    h  -= dt/dx * numpy.diff (Fh,  axis=X) + dt/dy * numpy.diff (Gh,  axis=Y)
    hu -= dt/dx * numpy.diff (Fhu, axis=X) + dt/dy * numpy.diff (Ghu, axis=Y)
    hv -= dt/dx * numpy.diff (Fhv, axis=X) + dt/dy * numpy.diff (Ghv, axis=Y)

    # update the time step size
    t += dt;

  # return the solution
  return [xg, yg] + PrimitiveVars (h, hu, hv)

if __name__ == '__main__':

  # problem data
  D = [[0, 2], [0, 1]]
  T = 0.1
  g = 8.912
  h0 = lambda x, y : numpy.where (x < (y - 0.5)**2 + 0.75, 2.0, 1.5)
  u0 = lambda x, y : numpy.zeros (x.shape)
  v0 = lambda x, y : numpy.zeros (x.shape)
  F  = lambda h, u, v, g : [h*u, h*u*u + 0.5*g*h*h, h*u*v]
  G  = lambda h, u, v, g : [h*v, h*u*v, h*v*v + 0.5*g*h*h]
  DF = lambda h, u, v, g : numpy.fabs(u) + numpy.sqrt(g*h)
  DG = lambda h, u, v, g : numpy.fabs(v) + numpy.sqrt(g*h)

  # discretization parameters
  #N = [512, 256]
  N = [64, 32]
  CFL = 0.9

  # run the FVM scheme
  [xg, yg, h, u, v] = shallow_water_2d (D, g, T, h0, u0, v0, F, G, DF, DG, N, rusanov_flux, CFL)

```

```
  # === plot the results

  # create figure of a desired size
  pylab.figure (figsize=(2*8,2*4))

  pylab.subplot (221)
  pylab.imshow (h0(xg,yg).T)
  pylab.colorbar()
  pylab.title (r'Water column height h at t = %.2f' % 0)

  # choose color scheme
  pylab.winter()

  pylab.subplot (222)
  pylab.imshow (h.T)
  pylab.colorbar()
  pylab.title (r'Water column height h at t = %.2f' % T)

  pylab.subplot (223)
  pylab.imshow (u.T)
  pylab.colorbar()
  pylab.title (r'Water velocity u at t = %.2f' % T)

  # choose color scheme
  pylab.jet()

  pylab.subplot (224)
  pylab.imshow (v.T)
  pylab.colorbar()
  pylab.title (r'Water velocity v at t = %.2f' % T)

  # adjust the margins, save figure and display in screen
  pylab.subplots_adjust(left=0.05,right=0.95)
  pylab.savefig ('../fig/shallow_water_2d.eps')
  pylab.show()
```

Listing 6.24: Implementation for `compute_flux_2d`

```
import numpy
from variables import ConservedVars, PrimitiveVars

# constants for 'x' and 'y' directions
X = 0
Y = 1

# compute all fluxes using the numerical flux 'num_flux' (in
   direction 'd')
def compute_flux_2d (h, hu, hv, F, DF, g, num_flux, d):
```

```
# get number of cells (in each direction)
N = h.shape

# for fluxes, increment the direction 'd' by 1
NF = list(N[:])
NF[d] += 1

# allocate arrays for fluxes
Fh  = numpy.zeros (NF)
Fhu = numpy.zeros (NF)
Fhv = numpy.zeros (NF)

# allocate array for the wave speeds
lmax = numpy.zeros (NF)

# compute primitive variables
[h, u, v] = PrimitiveVars (h, hu, hv)

# compute fluxes at each cell interface (in direction 'd')
for i in xrange(0, NF[X]):
  for j in xrange(0, NF[Y]):

    # set left and right index pairs
    # incorporating 'outflow' boundary conditions
    Li = max (0, i - (1 if d == X else 0) )
    Lj = max (0, j - (1 if d == Y else 0) )
    Ri = min (i, N[X] - 1)
    Rj = min (j, N[Y] - 1)

    # compute the maximum wave speed
    lmax[i][j] = max ( DF (h[Li][Lj], u[Li][Lj], v[Li][Lj], g), DF (h[Ri][Rj], u[Ri][Rj], v[Ri][Rj], g) )

    # compute the left and the right fluxes
    [FhL, FhuL, FhvL] = F (h[Li][Lj], u[Li][Lj], v[Li][Lj], g)
    [FhR, FhuR, FhvR] = F (h[Ri][Rj], u[Ri][Rj], v[Ri][Rj], g)

    # compute the Rusanov flux
    Fh  [i][j] = num_flux (FhL,  FhR,  lmax[i][j], h [Ri][Rj], h [Li][Lj])
    Fhu [i][j] = num_flux (FhuL, FhuR, lmax[i][j], hu[Ri][Rj], hu[Li][Lj])
    Fhv [i][j] = num_flux (FhvL, FhvR, lmax[i][j], hv[Ri][Rj], hv[Li][Lj])
```

```

    # compute the global maximal wave speed
    lmax = numpy.max(lmax)

    # return the numerical flux and the global maximal wave speed
    return [Fh, Fhu, Fhv, lmax]
```

Published on May 14th.

To be submitted on May 30th.

Last modified on June 12, 2014