

BACHELOR

Numerical methods for the shallow water equations

Burgerjon, Luc M.A.

Award date:
2021

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

**Department of Mathematics and
Computer Science**
*Centre for Analysis, Scientific computing
and Applications*
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
www.tue.nl

Date
July 19, 2021

**Numerical Methods for
the Shallow Water Equations**
Bachelor Final Project Applied Mathematics

Author
L.M.A. Burgerjon (1360744)
l.m.a.burgerjon@student.tue.nl

Supervisor
Prof.dr.ir. B. Koren
b.koren@tue.nl

Abstract

This report studies numerical methods to solve the shallow water equations, by deriving and comparing various numerical methods using Matlab. We mainly consider finite volume methods. For state interpolation methods, we consider first and second-order upwind and limited second-order methods, with the Minmod, Superbee and Koren limiters. For flux evaluations, in the one-dimensional case we consider flux vector splitting, flux difference splitting and HLL-type solvers. Moreover, for the 2D case we consider flux vector splitting and flux difference splitting. Time-integration is performed using a RK3b scheme. We test the methods on three test cases: a smooth initial hill on a flat bottom, a dam-break on a flat bottom and a supercritical flow over a bump.

Table of contents

Numerical Methods for
the Shallow Water Equations

1	Introduction	1
2	The shallow water equations	2
2.1	Derivation of the two-dimensional shallow water equations	2
2.2	One-dimensional form	5
3	Numerical methods in the 1D case	6
3.1	Finite difference methods	6
3.2	Finite volume methods	8
3.2.1	Discontinuities	8
3.2.2	Derivation of the 1D finite volume method	8
3.2.3	State interpolation	9
3.2.3.1	First-order upwind	9
3.2.3.2	Second-order upwind	9
3.2.3.3	Second-order limited scheme	9
3.2.3.4	Limiters	10
3.2.4	Flux evaluation	11
3.2.4.1	Flux vector splitting	11
3.2.4.2	Flux difference splitting	12
3.2.4.3	HLL-type solvers	14
3.2.5	Source term discretization	17
3.2.6	Time integration	17
4	Numerical methods in the 2D case	18
4.1	Derivation of the 2D finite volume method	18
4.2	State interpolation	19
4.2.1	First-order upwind	20
4.2.2	Second-order upwind	20
4.2.3	Second-order limited scheme	20
4.3	Flux evaluation	21
4.3.1	Flux vector splitting	21
4.3.2	Flux difference splitting	23
4.3.3	Source term discretization	26
4.3.4	Time integration	26

Table of contents

Numerical Methods for
the Shallow Water Equations

5 Numerical results	27
5.1 Test case 1: A water hill on a flat bottom	27
5.2 Test case 2: The dam-break problem	35
5.3 Test case 3: Flow over a bump on the bottom	43
6 Conclusions	46
7 Further research	47
References	48
A Numerical implementation of limiters	49
B Matlab code	50
B.1 Numerical implementation of state interpolation schemes	50
B.2 Test case 1: State interpolation methods	51
B.3 Test case 1: Flux evaluation methods	59
B.4 Test case 2: State interpolation methods	69
B.5 Test case 2: Flux evaluation methods	75
B.6 Test case 3: Flux vector splitting	83
B.7 Test case 3: Trapezoidal rule	90
B.8 Test case 3: Midpoint rule	97
B.9 Test case 3: Roe's method	104

1 Introduction

A wide range of physical phenomena is governed by the so-called *shallow water equations*. These hyperbolic partial differential equations describe the flow in air or in water. Examples include the water flow in open channels, rivers, tsunamis and flood modelling. Because of this, the equations are of high interest to a variety of engineers. However, these equations are not easily solved and a great deal of research has gone into developing methods for solving them [17,20,21]. A variety of numerical methods has been developed to numerically simulate the shallow water equations, some of which we consider in this report. We derive and discuss finite difference methods, but mainly finite volume methods to numerically solve these equations. We also compare them in order to determine which of them works best.

In Chapter 2 we derive the shallow water equations and discuss which assumptions are made in order to obtain them. In Chapter 3 we mainly derive a variety of finite volume methods to numerically solve the shallow water equations in the 1D case. We extend some of these numerical finite volume methods to the 2D case in Chapter 4. Finally, in Chapter 5 we consider three distinct test cases to compare how well each of the finite volume methods performs.

2 The shallow water equations

Before studying numerical methods for solving the shallow water equations, we give a derivation of the equations in two-dimensional differential form. Furthermore, by disregarding one spatial coordinate, the one-dimensional shallow water equations are obtained as well.

2.1 Derivation of the two-dimensional shallow water equations

The shallow water equations can be given in both one-dimensional and two-dimensional form. In his book *Shock-Capturing Methods for Free-Surface Shallow Flows* [20], Toro derives the two-dimensional shallow water equations using the general conservation laws of mass and momentum of a compressible fluid. In his deduction he clearly illustrates which assumptions are made and what the consequences of these assumptions are, which is why we roughly follow the same steps as his derivation. We start our derivation with the conservation laws of mass and momentum, given by

$$\rho_t + \nabla \cdot (\rho \vec{V}) = 0, \quad (2.1a)$$

$$\frac{\partial}{\partial t} (\rho \vec{V}) + \nabla \cdot [\rho \vec{V} \otimes \vec{V} + p \mathbf{I} - \mathbf{\Pi}] = \rho \vec{g}, \quad (2.1b)$$

where the independent variables are t for time and x, y, z . Furthermore, $\rho = \rho(x, y, z, t)$ is the fluid density, $\vec{V} = (u, v, w)$ is the fluid velocity vector where u, v, w are the x, y, z components of the velocity respectively, $p = p(x, y, z, t)$ is the pressure, \otimes represents the outer product, \mathbf{I} is the identity matrix of size 3×3 , $\mathbf{\Pi}$ is the viscous stress tensor and \vec{g} is the body force vector. Throughout this report we denote $\rho_t := \frac{\partial \rho}{\partial t}$.

When viscous effects are neglected, and thus the stress tensor $\mathbf{\Pi}$ vanishes, equations (2.1a) and (2.1b) can be written as

$$\rho_t + u\rho_x + v\rho_y + w\rho_z + \rho(u_x + v_y + w_z) = 0, \quad (2.2a)$$

$$u_t + uu_x + vu_y + wu_z + \frac{1}{\rho}p_x = g_1, \quad (2.2b)$$

$$v_t + uv_x + vv_y + wv_z + \frac{1}{\rho}p_y = g_2, \quad (2.2c)$$

$$w_t + uw_x + vw_y + ww_z + \frac{1}{\rho}p_z = g_3. \quad (2.2d)$$

For the remainder of the report we take the z -coordinate as the vertical direction of the spatial coordinates, i.e., the z -coordinate is the direction of the free-surface elevation.

We assume that both the density of the fluid ρ and the body force vector is $\vec{g} = (0, 0, -g)$ are constant, where g is the acceleration due to gravity. For simplicity, we take $g \equiv 1$ throughout the report. Equations (2.2) simplify to

$$u_x + v_y + w_z = 0, \quad (2.3a)$$

$$u_t + uu_x + vu_y + wu_z = -\frac{1}{\rho}p_x, \quad (2.3b)$$

$$v_t + uv_x + vv_y + wv_z = -\frac{1}{\rho}p_y, \quad (2.3c)$$

$$w_t + uw_x + vw_y + ww_z = -\frac{1}{\rho}p_z - g. \quad (2.3d)$$

Before finishing the derivation of the shallow water equations, let us consider the boundary conditions at the top and bottom boundaries on a rectangular domain $[x_L, x_R] \times [y_L, y_R]$. We consider the bottom boundary given by

$$z = b(x, y), \quad (2.4)$$

and the top boundary, that hereinafter will be referred to as the *free surface*, specified by

$$z = s(x, y, t) \equiv b(x, y) + h(x, y, t), \quad (2.5)$$

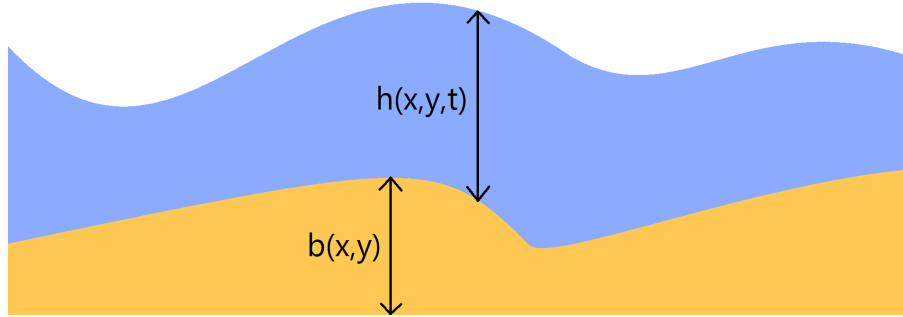


Figure 2.1: 1-dimensional sketch of the cross-section of the domain.

where $h = h(x, y, t)$ is the water depth. Figure 2.1 illustrates the situation for a cross-section of the domain. Assuming that a boundary is given by the surface

$$\psi(x, y, z, t) = 0, \quad (2.6)$$

for the bottom boundary as given by (2.4) we have

$$\psi(x, y, z, t) \equiv z - b(x, y) = 0, \quad (2.7)$$

and for the free surface as given by (2.5) we have

$$\psi(x, y, z, t) \equiv z - s(x, y, t) = 0. \quad (2.8)$$

We impose two boundary conditions on the free surface and one boundary condition on the bottom boundary. For both the free surface and the bottom boundary we impose the kinematic condition, which relates the motion of the local fluid velocity to both boundaries:

$$\frac{d}{dt}\psi(x, y, z, t) = \psi_t + u\psi_x + v\psi_y + w\psi_z = 0. \quad (2.9)$$

Additionally, we impose the dynamic condition for the free surface. This requires the stress to be continuous across the free surface which separates the water and the atmosphere and is given by

$$p(x, y, z, t)|_{z=s(x,y)} - p_{\text{atm}} = 0, \quad (2.10)$$

where p_{atm} is the atmospheric pressure, for convenience we set $p_{\text{atm}} \equiv 0$.

Now an important step follows. We assume that the vertical component of acceleration is negligible, thus

$$\frac{dw}{dt} = w_t + uw_x + vw_y + ww_z = 0. \quad (2.11)$$

By inserting equation (2.11) into equation (2.3d), equations (2.3) become

$$u_x + v_y + w_z = 0, \quad (2.12a)$$

$$u_t + uu_x + vv_y + ww_z = -\frac{1}{\rho}p_x, \quad (2.12b)$$

$$v_t + uv_x + vv_y + ww_z = -\frac{1}{\rho}p_y, \quad (2.12c)$$

$$p_z = -\rho g. \quad (2.12d)$$

Substituting the dynamic condition given by equation (2.10) into equation (2.12d) and taking into account that $p_{\text{atm}} \equiv 0$, we obtain

$$p = \rho g(s - z), \quad (2.13)$$

and hence

$$p_x = \rho g s_x, \quad (2.14a)$$

$$p_y = \rho g s_y. \quad (2.14b)$$

Note that both p_x and p_y are independent of z , and using equations (2.12b) and (2.12c) we thus know that the x and y components of the acceleration of water particles, denoted by $\frac{du}{dt}$ and $\frac{dv}{dt}$, are independent of z .

Assuming that the velocity components u and v are independent of z at a given time, say $t = 0$, we obtain that $u_z = v_z = 0$. Using this, equations (2.14) and dropping equation (2.12d), equations (2.12) become

$$u_x + v_y + w_z = 0, \quad (2.15a)$$

$$u_t + uu_x + vv_y = -gs_x, \quad (2.15b)$$

$$v_t + uv_x + vv_y = -gs_y. \quad (2.15c)$$

Now the final steps in the derivation follow. Integrating equation (2.15a) with respect to the vertical coordinate z between $z = b(x, y)$ and $z = s(x, y, t)$ gives

$$\int_{z=b}^{z=s} (u_x + v_y + w_z) dz = 0, \quad (2.16)$$

which using the Fundamental Theorem of Calculus can be written as

$$w|_{z=s} - w|_{z=b} + \int_{z=b}^{z=s} u_x dz + \int_{z=b}^{z=s} v_y dz = 0. \quad (2.17)$$

Applying Leibniz's Formula to the two integrals in equation (2.17) yields

$$\int_{z=b}^{z=s} u_x dz = \frac{\partial}{\partial x} \int_{z=b}^{z=s} u dz - u|_{z=s} s_x + u|_{z=b} b_x, \quad (2.18a)$$

$$\int_{z=b}^{z=s} v_y dz = \frac{\partial}{\partial y} \int_{z=b}^{z=s} v dz - v|_{z=s} s_y + v|_{z=b} b_y. \quad (2.18b)$$

Furthermore, applying the kinematic condition (2.9) to the bottom boundary and the free surface, given by (2.7) and (2.8) respectively, results in

$$w|_{z=s} = (s_t + us_x + vs_y)|_{z=s}, \quad (2.19a)$$

$$w|_{z=b} = (ub_x + vb_y)|_{z=b}. \quad (2.19b)$$

Substitution of equations (2.18)–(2.19) into equation (2.17) yields

$$s_t + \frac{\partial}{\partial x} \int_{z=b}^{z=s} u dz + \frac{\partial}{\partial y} \int_{z=b}^{z=s} v dz = 0. \quad (2.20)$$

Recall that $u_z = v_z = 0$, thus u and v are independent of z , and that $s = b + h$ and $b_t = 0$. Applying this to equation (2.20) yields

$$h_t + (hu)_x + (hv)_y = 0, \quad (2.21)$$

and hence, by using $s = b + h$, equations (2.15) can be written as

$$h_t + (hu)_x + (hv)_y = 0, \quad (2.22a)$$

$$u_t + uu_x + vv_y = -gb_x, \quad (2.22b)$$

$$v_t + uv_x + vv_y = -gb_y. \quad (2.22c)$$

However, equations (2.22b) and (2.22c) can also be expressed in conservation law form in accordance to (2.22a). Assuming that the water depth h is differentiable w.r.t. x and y , we obtain

$$h \frac{\partial h}{\partial x} = \frac{\partial}{\partial x} \left(\frac{1}{2} h^2 \right), \quad (2.23a)$$

$$h \frac{\partial h}{\partial y} = \frac{\partial}{\partial y} \left(\frac{1}{2} h^2 \right). \quad (2.23b)$$

Using (2.23a), adding equation (2.21) pre-multiplied by u , to equation (2.22b) pre-multiplied by h yields

$$(hu)_t + \left(hu^2 + \frac{1}{2}gh^2\right)_x + (huv)_y = -ghb_x. \quad (2.24)$$

Similarly, using (2.23b), adding equation (2.21) pre-multiplied by v , to equation (2.22c) pre-multiplied by h yields

$$(hv)_t + (huv)_x + \left(hv^2 + \frac{1}{2}gh^2\right)_y = -ghb_y. \quad (2.25)$$

Together, the three partial differential equations (2.21), (2.24) and (2.25) form the so-called 2-dimensional shallow water equations which are in differential conservation law form:

$$\vec{U}_t + \vec{F}(\vec{U})_x + \vec{G}(\vec{U})_y = \vec{S}(\vec{U}), \quad (2.26)$$

where

$$\vec{U} = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}, \quad \vec{F}(\vec{U}) = \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}, \quad \vec{G}(\vec{U}) = \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}, \quad \vec{S}(\vec{U}) = \begin{bmatrix} 0 \\ -ghb_x \\ -ghb_y \end{bmatrix}. \quad (2.27)$$

For computational reasons, in this report we use the equations above in the following equivalent form:

$$\vec{q} = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \end{bmatrix} = \begin{bmatrix} h \\ hu \\ hv \end{bmatrix}, \quad \vec{F}(\vec{q}) = \begin{bmatrix} q_2 \\ \frac{q_2^2}{q_1} + \frac{1}{2}gq_1^2 \\ \frac{q_2q_3}{q_1} \end{bmatrix}, \quad \vec{G}(\vec{q}) = \begin{bmatrix} q_3 \\ \frac{q_2q_3}{q_1} \\ \frac{q_3^2}{q_1} + \frac{1}{2}gq_1^2 \end{bmatrix}, \quad \vec{S}(\vec{q}) = \begin{bmatrix} 0 \\ -gq_1b_x \\ -gq_1b_y \end{bmatrix} \quad (2.28)$$

2.2 One-dimensional form

The shallow water equations can also be derived in one-dimensional form by not taking into account the y -direction. The one-dimensional shallow water equations are [20]

$$\vec{U}_t + \vec{F}(\vec{U})_x = \vec{S}(\vec{U}), \quad (2.29)$$

where

$$\vec{U} = \begin{bmatrix} h \\ hu \end{bmatrix}, \quad \vec{F}(\vec{U}) = \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \end{bmatrix}, \quad \vec{S}(\vec{U}) = \begin{bmatrix} 0 \\ -ghb_x \end{bmatrix}. \quad (2.30)$$

Similar to the two-dimensional case, we mostly use the equations above in the following form:

$$\vec{q} = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} h \\ hu \end{bmatrix}, \quad \vec{F}(\vec{q}) = \begin{bmatrix} q_2 \\ \frac{q_2^2}{q_1} + \frac{1}{2}gq_1^2 \end{bmatrix}, \quad \vec{S}(\vec{q}) = \begin{bmatrix} 0 \\ -gq_1b_x \end{bmatrix} \quad (2.31)$$

3 Numerical methods in the 1D case

In this chapter we discuss various numerical methods for simulating the 1-dimensional shallow water equations, which were derived in the previous chapter, and are given by (2.29) and (2.30). We discuss each method in detail, indicating what their (dis)advantages are. The main methods that are treated here are the finite difference method, which we only briefly discuss, and the finite volume method, which has a variety of implementations. To be more precise, for the finite volume method we first perform state interpolation and then evaluate the fluxes, for each of which various techniques are considered.

3.1 Finite difference methods

Finite difference methods may be the first option one thinks about when numerically approximating the 1D shallow equations, or any set of (partial) differential equations, which is why we discuss them here [22].

Let us consider a domain $[x_L, x_R] \times [0, T]$ in the $x - t$ plane. We divide the spatial domain of length $x_R - x_L$ into n equal parts, each with length Δx , where the nodes are given by $x_i = i\Delta x$ ($i = 0, \dots, n$). Figure 3.1 illustrates this subdivision of the spatial domain. Furthermore, the solution is approximated at times $t_j = j\Delta t$ ($j = 1, \dots, m$) using m time steps of length $\Delta t = \frac{T}{m}$. We denote the numerical approximation of $(h(x_i, t_j), u(x_i, t_j))$ by (h_i^j, u_i^j)

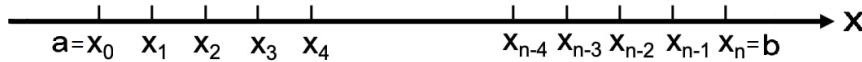


Figure 3.1: Sketch of the discretization of the spatial domain for the finite difference method.

We start by writing down the 1D shallow water equations slightly differently from (2.29) and (2.30):

$$\frac{\partial}{\partial t} h(x, t) + \frac{\partial}{\partial x} h(x, t)u(x, t) = 0, \tag{3.1a}$$

$$\frac{\partial}{\partial t} h(x, t)u(x, t) + \frac{\partial}{\partial x} \left[h(x, t)u^2(x, t) + \frac{1}{2}gh^2(x, t) \right] = 0. \tag{3.1b}$$

By applying the Chain Rule, equations (3.1) can be written as

$$h_t(x, t) + h(x, t)u_x(x, t) + u(x, t)h_x(x, t) = 0, \tag{3.2a}$$

$$h(x, t)u_t(x, t) + u(x, t)h_t(x, t) + 2h(x, t)u(x, t)u_x(x, t) + (u^2(x, t) + g \cdot h(x, t)) h_x(x, t) = 0, \tag{3.2b}$$

Subtracting equation (3.2a) pre-multiplied by $u(x, t)$ from equation (3.2b), equations (3.2) read

$$h_t(x, t) + h(x, t)u_x(x, t) + u(x, t)h_x(x, t) = 0, \tag{3.3a}$$

$$h(x, t)u_t(x, t) + h(x, t)u(x, t)u_x(x, t) + gh(x, t)h_x(x, t) = 0, \tag{3.3b}$$

and by dividing out $h(x, t)$ in equation (3.3b), the above becomes

$$h_t(x, t) + h(x, t)u_x(x, t) + u(x, t)h_x(x, t) = 0, \tag{3.4a}$$

$$u_t(x, t) + u(x, t)u_x(x, t) + gh_x(x, t) = 0, \tag{3.4b}$$

Because we are not focussed on achieving high-order accuracy, we simply apply forward differencing for time, i.e., we discretize h_t by

$$\frac{h_i^{j+1} - h_i^j}{\Delta t}, \tag{3.5}$$

and u_t by

$$\frac{u_i^{j+1} - u_i^j}{\Delta t}. \tag{3.6}$$

Moreover, we apply central differencing for space, i.e., we discretize h_x by

$$\frac{h_{i+1}^j - h_{i-1}^j}{2\Delta x}, \quad (3.7)$$

and u_x by

$$\frac{u_{i+1}^j - u_{i-1}^j}{2\Delta x}. \quad (3.8)$$

Using discretizations (3.5)–(3.8), our discretization scheme for equations (3.4) reads

$$\frac{h_i^{j+1} - h_i^j}{\Delta t} + h_i^j \frac{u_{i+1}^j - u_{i-1}^j}{2\Delta x} + u_i^j \frac{h_{i+1}^j - h_{i-1}^j}{2\Delta x} = 0, \quad (3.9a)$$

$$\frac{u_i^{j+1} - u_i^j}{\Delta t} + u_i^j \frac{u_{i+1}^j - u_{i-1}^j}{2\Delta x} + g \frac{h_{i+1}^j - h_{i-1}^j}{2\Delta x} = 0. \quad (3.9b)$$

For convenience, the numerical scheme (3.9) can also be written into explicit form, given by

$$h_i^{j+1} = h_i^j - \frac{\Delta t}{2\Delta x} \left[h_i^j (u_{i+1}^j - u_{i-1}^j) + u_i^j (h_{i+1}^j - h_{i-1}^j) \right], \quad (3.10a)$$

$$u_i^{j+1} = u_i^j - \frac{\Delta t}{2\Delta x} \left[u_i^j (u_{i+1}^j - u_{i-1}^j) + g (h_{i+1}^j - h_{i-1}^j) \right]. \quad (3.10b)$$

It can be shown using numerical stability analysis techniques from [22] that the scheme above is unconditionally unstable, rendering the scheme practically useless. Hence, a better alternative to the method above is needed which we discuss in the next section.

3.2 Finite volume methods

3.2.1 Discontinuities

An important property of nonlinear conservation laws, such as the shallow water equations, is that discontinuities can easily occur and even spontaneously arise from smooth initial data. Finite difference methods, in which derivatives are approximated by finite differences, break down near discontinuities in the solution where the differential equation does not hold [11]. For this reason we also consider finite volume methods, based on the integral form instead of the differential form. The integral formulation requires less smoothness of the solution, which allows discontinuous solutions [20,21].

3.2.2 Derivation of the 1D finite volume method

Let us consider the one-dimensional shallow water equations as given by (2.31) on a domain $[x_L, x_R] \times [0, T]$ in the $x - t$ plane. We start by subdividing the spatial domain of length $L := x_R - x_L$ into M equally sized finite volumes/cells, where $x \in I_i$ ($i = 1, \dots, M$) if and only if

$$x_{i-\frac{1}{2}} := (i-1)\Delta x \leq x \leq i\Delta x =: x_{i+\frac{1}{2}}, \tag{3.11}$$

and the size of each cell is

$$\Delta x := x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}} = \frac{x_R - x_L}{M}, \tag{3.12}$$

as can be seen in Figure 3.2. Consider the control volume $[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}]$, then by integrating (2.29) in space we obtain

$$\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \vec{q}_t(x, t) \, dx + \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \vec{F}(\vec{q}(x, t))_x \, dx = \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \vec{S}(\vec{q}(x, t)) \, dx. \tag{3.13}$$

Since the cell I_i remains unchanged in time the Leibniz integral rule can be applied to the first term above and the Fundamental Theorem of Calculus can be applied to the second term, which yields

$$\frac{d}{dt} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \vec{q}(x, t) \, dx + \vec{F}(\vec{q}(x_{i+\frac{1}{2}}, t)) - \vec{F}(\vec{q}(x_{i-\frac{1}{2}}, t)) = \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \vec{S}(\vec{q}(x, t)) \, dx. \tag{3.14}$$

Finally, the above can be rewritten into the finite volume equation

$$\frac{d\vec{q}_i}{dt} + \frac{1}{\Delta x} \left(\vec{F}_{i+\frac{1}{2}} - \vec{F}_{i-\frac{1}{2}} \right) = \vec{S}_i, \tag{3.15}$$

where

$$\vec{q}_i := \frac{1}{\Delta x} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \vec{q}(x, t) \, dx, \quad \vec{F}_{i+\frac{1}{2}} := \vec{F}(\vec{q}(x_{i+\frac{1}{2}}, t)), \quad \vec{S}_i := \frac{1}{\Delta x} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \vec{S}(\vec{q}(x, t)) \, dx. \tag{3.16}$$

Note that \vec{q}_i is the *cell average*. In our numerical method, we assume the numerical solution of \vec{q}_i to be constant in space in each cell I_i .

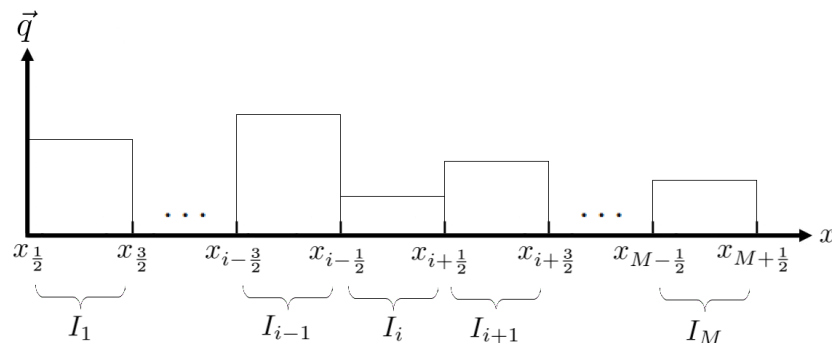


Figure 3.2: Sketch of the division of the spatial domain for the finite volume method.

3.2.3 State interpolation

In order to evaluate the fluxes $\vec{F}_{\frac{1}{2}}, \vec{F}_{\frac{3}{2}}, \dots, \vec{F}_{M+\frac{1}{2}}$ in the finite volume equation (3.15), one needs the numerical solutions of \vec{q} at $x_{\frac{1}{2}}, x_{\frac{3}{2}}, \dots, x_{M+\frac{1}{2}}$, which we denote by $\vec{q}_{\frac{1}{2}}, \vec{q}_{\frac{3}{2}}, \dots, \vec{q}_{M+\frac{1}{2}}$. However, since the numerical solutions of \vec{q}_i are constant in space in each cell I_i , each cell is very likely to have a different numerical solution than its neighbours. Hence, the numerical solution of \vec{q}_i is most often discontinuous at the boundaries $x_{i-\frac{1}{2}}$ and $x_{i+\frac{1}{2}}$ of each cell I_i and thus the values $\vec{q}_{i+\frac{1}{2}}$ and $\vec{q}_{i-\frac{1}{2}}$ are not straightforward. In order to solve this issue, a general approach is taken by setting

$$\vec{F}_{i+\frac{1}{2}} = \vec{\mathcal{F}}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r), \quad (3.17)$$

where $\vec{q}_{i+\frac{1}{2}}^l$ is the numerical solution to the left of $x_{i+\frac{1}{2}}$, $\vec{q}_{i+\frac{1}{2}}^r$ is the numerical solution to the right of $x_{i+\frac{1}{2}}$ and $\mathcal{F}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r)$ is the flux evaluation function which has yet to be chosen.

3.2.3.1 First-order upwind

The most straightforward option for $\vec{q}_{i+\frac{1}{2}}^l$ and $\vec{q}_{i+\frac{1}{2}}^r$ is setting

$$\vec{q}_{i+\frac{1}{2}}^l = \vec{q}_i, \quad (3.18a)$$

$$\vec{q}_{i+\frac{1}{2}}^r = \vec{q}_{i+1}, \quad (3.18b)$$

which is first-order accurate in space.

3.2.3.2 Second-order upwind

A more elaborate option, which was suggested by Van Leer [10], is taking

$$\vec{q}_{i+\frac{1}{2}}^l = \vec{q}_i + \frac{1+\kappa}{4}(\vec{q}_{i+1} - \vec{q}_i) + \frac{1-\kappa}{4}(\vec{q}_i - \vec{q}_{i-1}), \quad (3.19a)$$

$$\vec{q}_{i+\frac{1}{2}}^r = \vec{q}_{i+1} + \frac{1+\kappa}{4}(\vec{q}_i - \vec{q}_{i+1}) + \frac{1-\kappa}{4}(\vec{q}_{i+1} - \vec{q}_{i+2}), \quad (3.19b)$$

where κ is a parameter to be chosen in the range $[-1, 1]$. It can be shown that this scheme is second-order accurate in space for every value of κ . For $\kappa = -1$ this scheme reduces to a fully one-sided extrapolation and for $\kappa = 1$ this becomes a central interpolation, i.e., it takes the average of \vec{q}_i and \vec{q}_{i+1} . Furthermore, for $\kappa = \frac{1}{3}$ the scheme can even be third-order accurate in space [9].

3.2.3.3 Second-order limited scheme

Koren [9] has shown that the previously mentioned schemes (3.18) and (4.9) do not satisfy the *Positive Coefficients Rule* [13] for the 1D advection equation, hence these schemes are not monotone which is associated with the unwanted phenomenon of spurious oscillations [21]. In an attempt to remedy this, the author suggests taking the second-order accurate scheme by Koren [9], which is given by

$$\vec{q}_{i+\frac{1}{2}}^l = \vec{q}_i + \frac{1}{2}\phi(r_{i+\frac{1}{2}}^l)(\vec{q}_i - \vec{q}_{i-1}), \quad (3.20a)$$

$$\vec{q}_{i+\frac{1}{2}}^r = \vec{q}_{i+1} + \frac{1}{2}\phi(r_{i+\frac{1}{2}}^r)(\vec{q}_{i+1} - \vec{q}_{i+2}), \quad (3.20b)$$

where

$$r_{i+\frac{1}{2}}^l = \frac{\vec{q}_{i+1} - \vec{q}_i}{\vec{q}_i - \vec{q}_{i-1}}, \quad r_{i+\frac{1}{2}}^r = \frac{\vec{q}_i - \vec{q}_{i+1}}{\vec{q}_{i+1} - \vec{q}_{i+2}}, \quad (3.21)$$

with the *limiter function* $\phi(r)$ yet to be chosen. Koren [9] has also shown that in some cases this scheme is third-order accurate in space. The numerical implementation of this scheme, in order to be able to work with division by zero cases, is explained in Appendix A.

3.2.3.4 Limiters

Many limiter functions $\phi(r)$ exist to choose from. Examples of these are the Superbee [14], Minmod [14] and Koren [9] limiters. As mentioned in Section 3.2.3.3, monotonicity is a desirable property of these limiters. This characteristic of limiters has been extensively researched and it has been shown that limiters, which have the *Totally Variation Diminishing* (TVD) property, preserve monotonicity [6]. Sweby [18] continued on this and determined a so-called *TVD Region*, between whose boundaries a limiter $\phi(r)$ has to be in order for it to be TVD. Furthermore, Sweby determined the *Second-Order TVD Region*, between which boundaries a limiter has to be in order for it to be both TVD, and thus monotone, and second-order accurate in space. The Second-Order TVD Region is drawn in Figure 3.3.

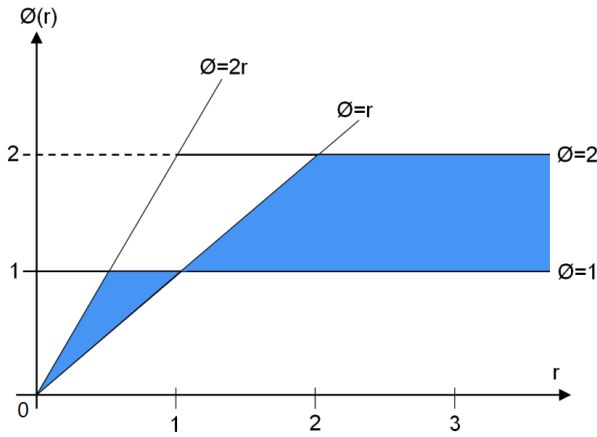


Figure 3.3: Sketch of the division of the spatial domain.

Examples of *Second-Order TVD Limiters* include the previously mentioned Superbee [14], Minmod [14] and Koren [9] limiters, which are given below.

$$\phi_{sb}(r) = \begin{cases} 0, & \text{if } r < 0 \\ \min(2r, 1), & \text{if } 0 \leq r < 1 \\ \min(2, r), & \text{if } r \geq 1. \end{cases} \quad (3.22)$$

$$\phi_{mm}(r) = \begin{cases} 0, & \text{if } r < 0 \\ \min(1, r), & \text{if } r \geq 0. \end{cases} \quad (3.23)$$

$$\phi_{kn}(r) = \begin{cases} 0, & \text{if } r < 0 \\ 2r, & \text{if } 0 \leq r < \frac{1}{4} \\ \min(\frac{1}{3} + \frac{2}{3}r, 2), & \text{if } r \geq \frac{1}{4}. \end{cases} \quad (3.24)$$

3.2.4 Flux evaluation

Once the state interpolation values have been calculated, the next step is to determine a flux evaluation function $\vec{F}_{i+\frac{1}{2}} = \vec{F}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r)$. Throughout the years many methods have been developed to do this, see [11, 14, 20, 21], and some of these methods will be worked out for the one-dimensional shallow water equations and compared to each other.

3.2.4.1 Flux vector splitting

Flux vector splitting schemes have a straightforward and efficient way of identifying the upwind directions [21], which is why it is an interesting flux evaluation scheme to discuss here. This method was developed by Steger and Warming [16] and is based on homogeneity property of the flux, that is

$$\mathbf{A}(\vec{q})\vec{q} = \vec{F}(\vec{q}), \quad (3.25)$$

where $\vec{F}(\vec{q})$ is as given by (2.31) and $\mathbf{A}(\vec{q})$ is the Jacobian matrix of $\vec{F}(\vec{q})$. Unfortunately, the shallow water equations do not satisfy this property. However, one can construct a matrix $\mathbf{A}^*(\vec{q})$ different from the Jacobian matrix $\mathbf{A}(\vec{q})$ which does satisfy (3.25), this is given by [1]

$$\mathbf{A}^*(\vec{q}) = \begin{pmatrix} 0 & 1 \\ -\frac{q_2^2}{q_1^2} + \frac{gq_1}{2} & \frac{2q_2}{q_1} \end{pmatrix}. \quad (3.26)$$

Now, using this matrix $\mathbf{A}^*(\vec{q})$, the first step is to write it as

$$\mathbf{A}^* = \mathbf{R}\mathbf{\Lambda}^*\mathbf{R}^{-1}, \quad (3.27)$$

where $\mathbf{\Lambda}^*$ is the diagonal matrix whose diagonal elements are the eigenvalues of \mathbf{A}^* , \mathbf{R} is the matrix whose columns are the eigenvectors of \mathbf{A}^* and finally \mathbf{R}^{-1} is the inverse of \mathbf{R} . One can verify that

$$\mathbf{\Lambda}^*(\vec{q}) = \begin{pmatrix} \lambda_1^* & 0 \\ 0 & \lambda_2^* \end{pmatrix}, \quad \mathbf{R}(\vec{q}) = \begin{pmatrix} 1 & 1 \\ \lambda_1^* & \lambda_2^* \end{pmatrix}, \quad \mathbf{R}(\vec{q})^{-1} = \frac{1}{\sqrt{2gq_1}} \begin{pmatrix} \lambda_2^* & -1 \\ -\lambda_1^* & 1 \end{pmatrix}, \quad (3.28)$$

where

$$\lambda_1^*(\vec{q}) = \frac{q_2}{q_1} - \sqrt{\frac{gq_1}{2}} = u - \sqrt{\frac{gh}{2}}, \quad \lambda_2^*(\vec{q}) = \frac{q_2}{q_1} + \sqrt{\frac{gq_1}{2}} = u + \sqrt{\frac{gh}{2}}. \quad (3.29)$$

It should be noted that the eigenvalues λ_1^* and λ_2^* are distinct from eigenvalues of the matrix \mathbf{A} , hence the wave speeds in simulations using this method could be incorrect. Furthermore, note that the eigenvalues λ_1^*, λ_2^* either have the same sign and all information goes from the left to the right or vice versa, or they have opposite sign and thus some information goes to the left and some information goes to the right. Flux vector splitting takes this into account by splitting the matrix \mathbf{A}^* in a positive and a negative part, given by

$$\mathbf{A}_+^* := \mathbf{R}\mathbf{\Lambda}_+^*\mathbf{R}^{*-1}, \quad \mathbf{A}_-^* := \mathbf{R}\mathbf{\Lambda}_-^*\mathbf{R}^{-1}, \quad (3.30)$$

where

$$\mathbf{\Lambda}_+^* := \begin{pmatrix} \max(\lambda_1^*, 0) & 0 \\ 0 & \max(\lambda_2^*, 0) \end{pmatrix}, \quad \mathbf{\Lambda}_-^* := \begin{pmatrix} \min(\lambda_1^*, 0) & 0 \\ 0 & \min(\lambda_2^*, 0) \end{pmatrix}. \quad (3.31)$$

One should note that it holds that

$$\mathbf{\Lambda}^* = \mathbf{\Lambda}_+^* + \mathbf{\Lambda}_-^*, \quad \mathbf{A}^* = \mathbf{A}_+^* + \mathbf{A}_-^*. \quad (3.32)$$

As a result, the inter cell flux is taken as

$$\vec{F}_{\text{FVS}}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r) := \mathbf{A}_+^*(\vec{q}_{i+\frac{1}{2}}^l)\vec{q}_{i+\frac{1}{2}}^l + \mathbf{A}_-^*(\vec{q}_{i+\frac{1}{2}}^r)\vec{q}_{i+\frac{1}{2}}^r. \quad (3.33)$$

3.2.4.2 Flux difference splitting

The use of the previously-discussed flux vector splitting method is unwanted in the case of the one-dimensional shallow water equations, since one is forced to take the matrix \mathbf{A}^* (with the wrong wave speeds) instead of the Jacobian \mathbf{A} . This is why flux difference splitting schemes will be discussed as an alternative.

First, one needs the Jacobian matrix $\mathbf{A}(\vec{q})$ of $\vec{F}(\vec{q})$, where $\vec{F}(\vec{q})$ is as given by (2.31), which is

$$\mathbf{A}(\vec{q}) = \begin{pmatrix} 0 & 1 \\ -\frac{q_2^2}{q_1^2} + gq_1 & \frac{2q_2}{q_1} \end{pmatrix}. \quad (3.34)$$

Similar to before, we decompose the matrix $\mathbf{A}(\vec{q})$ as

$$\mathbf{A} = \mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1}, \quad (3.35)$$

where $\mathbf{\Lambda}$ is the diagonal matrix whose diagonal elements are the eigenvalues of \mathbf{A} , \mathbf{S} is the matrix whose columns are the eigenvectors of \mathbf{A} and \mathbf{S}^{-1} is the inverse of \mathbf{S} . Furthermore, we have that

$$\mathbf{\Lambda}(\vec{q}) = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}, \quad \mathbf{S}(\vec{q}) = \begin{pmatrix} 1 & 1 \\ \lambda_1 & \lambda_2 \end{pmatrix}, \quad \mathbf{S}(\vec{q})^{-1} = \frac{1}{2\sqrt{gq_1}} \begin{pmatrix} \lambda_2 & -1 \\ -\lambda_1 & 1 \end{pmatrix}, \quad (3.36)$$

where

$$\lambda_1(\vec{q}) = \frac{q_2}{q_1} - \sqrt{gq_1}, \quad \lambda_2(\vec{q}) = \frac{q_2}{q_1} + \sqrt{gq_1}. \quad (3.37)$$

Similar to before, we now split the matrix \mathbf{A} in a positive and a negative part, given by

$$\mathbf{A}^+ := \mathbf{S}\mathbf{\Lambda}^+\mathbf{S}^{-1}, \quad \mathbf{A}^- := \mathbf{S}\mathbf{\Lambda}^-\mathbf{S}^{-1}, \quad (3.38)$$

where

$$\mathbf{\Lambda}^+ := \begin{pmatrix} \max(\lambda_1, 0) & 0 \\ 0 & \max(\lambda_2, 0) \end{pmatrix}, \quad \mathbf{\Lambda}^- := \begin{pmatrix} \min(\lambda_1, 0) & 0 \\ 0 & \min(\lambda_2, 0) \end{pmatrix}, \quad (3.39)$$

and again it holds that

$$\mathbf{\Lambda} = \mathbf{\Lambda}^+ + \mathbf{\Lambda}^-, \quad \mathbf{A} = \mathbf{A}^+ + \mathbf{A}^-. \quad (3.40)$$

Now the fundamental step follows, for which we will keep to Harten, Lax and Van Leer [7]. Suppose $\vec{q}_{i+\frac{1}{2}}^m$ is a state that does not differ much from states $\vec{q}_{i+\frac{1}{2}}^l$ and $\vec{q}_{i+\frac{1}{2}}^r$, then we require that

$$\begin{aligned} \vec{\mathcal{F}}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r) &= \vec{F}(\vec{q}_{i+\frac{1}{2}}^m) + \mathbf{A}^+(\vec{q}_{i+\frac{1}{2}}^m)(\vec{q}_{i+\frac{1}{2}}^l - \vec{q}_{i+\frac{1}{2}}^m) + \mathbf{A}^-(\vec{q}_{i+\frac{1}{2}}^m)(\vec{q}_{i+\frac{1}{2}}^r - \vec{q}_{i+\frac{1}{2}}^m) \\ &\quad + o(|\vec{q}_{i+\frac{1}{2}}^l - \vec{q}_{i+\frac{1}{2}}^m| + |\vec{q}_{i+\frac{1}{2}}^r - \vec{q}_{i+\frac{1}{2}}^m|). \end{aligned} \quad (3.41)$$

A natural choice here is to take

$$\vec{q}_{i+\frac{1}{2}}^m := \frac{\vec{q}_{i+\frac{1}{2}}^l + \vec{q}_{i+\frac{1}{2}}^r}{2}. \quad (3.42)$$

Moreover, note that

$$\vec{F}\left(\frac{\vec{q}_{i+\frac{1}{2}}^l + \vec{q}_{i+\frac{1}{2}}^r}{2}\right) = \frac{\vec{F}(\vec{q}_{i+\frac{1}{2}}^l) + \vec{F}(\vec{q}_{i+\frac{1}{2}}^r)}{2} + o(|\vec{q}_{i+\frac{1}{2}}^r - \vec{q}_{i+\frac{1}{2}}^l|). \quad (3.43)$$

Substituting (3.42) and (3.43) into (3.41) yields

$$\begin{aligned} \vec{\mathcal{F}}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r) &= \frac{1}{2} \left(\vec{F}(\vec{q}_{i+\frac{1}{2}}^l) + \vec{F}(\vec{q}_{i+\frac{1}{2}}^r) \right) - \left| \mathbf{A} \left(\frac{\vec{q}_{i+\frac{1}{2}}^l + \vec{q}_{i+\frac{1}{2}}^r}{2} \right) \right| \left(\vec{q}_{i+\frac{1}{2}}^r - \vec{q}_{i+\frac{1}{2}}^l \right) \\ &\quad + o(|\vec{q}_{i+\frac{1}{2}}^r - \vec{q}_{i+\frac{1}{2}}^l|), \end{aligned} \quad (3.44)$$

where the matrix $|\mathbf{A}|$ is given by

$$|\mathbf{A}| := \mathbf{A}^+ - \mathbf{A}^-. \quad (3.45)$$

Hence, we consider flux difference splitting schemes of the form

$$\vec{\mathcal{F}}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r) = \frac{1}{2} \left(\vec{F}(\vec{q}_{i+\frac{1}{2}}^l) + \vec{F}(\vec{q}_{i+\frac{1}{2}}^r) \right) - \frac{1}{2} \vec{d}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r), \quad (3.46)$$

where

$$\vec{d}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r) := \left| \mathbf{Q}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r) \right| (\vec{q}_{i+\frac{1}{2}}^r - \vec{q}_{i+\frac{1}{2}}^l), \quad (3.47)$$

and the consistency requirement

$$d(\vec{q}, \vec{q}) = 0 \quad (3.48)$$

is always satisfied. The reader should note that these schemes of this form are occasionally called Q-schemes in literature as well [1].

The midpoint rule

The simplest option for the matrix $\mathbf{Q}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r)$ in (3.47) is taking the arithmetic mean as given by (3.44), which is

$$\mathbf{Q}_{\text{MR}}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r) = \mathbf{A} \left(\frac{1}{2} (\vec{q}_{i+\frac{1}{2}}^l + \vec{q}_{i+\frac{1}{2}}^r) \right), \quad (3.49)$$

and hence the numerical flux is given by

$$\vec{\mathcal{F}}_{\text{MR}}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r) = \frac{1}{2} (\vec{F}(\vec{q}_{i+\frac{1}{2}}^l) + \vec{F}(\vec{q}_{i+\frac{1}{2}}^r)) - \frac{1}{2} \left| \mathbf{A} \left(\frac{1}{2} (\vec{q}_{i+\frac{1}{2}}^l + \vec{q}_{i+\frac{1}{2}}^r) \right) \right| (\vec{q}_{i+\frac{1}{2}}^r - \vec{q}_{i+\frac{1}{2}}^l). \quad (3.50)$$

The trapezoidal rule

Another straightforward option for the matrix $|\mathbf{Q}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r)|$ in (3.47) is taking

$$\left| \mathbf{Q}_{\text{TR}}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r) \right| = \frac{1}{2} (|\mathbf{A}(\vec{q}_{i+\frac{1}{2}}^l)| + |\mathbf{A}(\vec{q}_{i+\frac{1}{2}}^r)|), \quad (3.51)$$

and thus the numerical flux is given by

$$\vec{\mathcal{F}}_{\text{TR}}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r) = \frac{1}{2} (\vec{F}(\vec{q}_{i+\frac{1}{2}}^l) + \vec{F}(\vec{q}_{i+\frac{1}{2}}^r)) - \frac{1}{4} (|\mathbf{A}(\vec{q}_{i+\frac{1}{2}}^l)| + |\mathbf{A}(\vec{q}_{i+\frac{1}{2}}^r)|) (\vec{q}_{i+\frac{1}{2}}^r - \vec{q}_{i+\frac{1}{2}}^l). \quad (3.52)$$

Roe's method

Roe [15] suggested taking the matrix $\mathbf{Q}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r)$ such that

$$\vec{F}(\vec{q}_{i+\frac{1}{2}}^r) - \vec{F}(\vec{q}_{i+\frac{1}{2}}^l) = \mathbf{Q}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r) (\vec{q}_{i+\frac{1}{2}}^r - \vec{q}_{i+\frac{1}{2}}^l). \quad (3.53)$$

Here $\mathbf{Q}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r)$ is equal to the Jacobian matrix \mathbf{A} evaluated at some state $\vec{q}_{i+\frac{1}{2}}^*$ which is often called *Roe's average*. For the 1D shallow water equations Leveque [11] showed that by integrating the Jacobian matrix \mathbf{A} along a suitable path, it follows that the following choice satisfies condition (3.53):

$$\mathbf{Q}_{\text{Roe}}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r) = \mathbf{A}(\vec{q}_{i+\frac{1}{2}}^*) = \begin{pmatrix} 0 & 1 \\ -\tilde{u}_{i+\frac{1}{2}}^2 + \tilde{c}_{i+\frac{1}{2}}^2 & 2\tilde{u}_{i+\frac{1}{2}} \end{pmatrix}, \quad (3.54)$$

where

$$\vec{q}_{i+\frac{1}{2}}^l = \begin{pmatrix} h_{i+\frac{1}{2}}^l \\ h_{i+\frac{1}{2}}^l u_{i+\frac{1}{2}}^l \end{pmatrix}, \quad \vec{q}_{i+\frac{1}{2}}^r = \begin{pmatrix} h_{i+\frac{1}{2}}^r \\ h_{i+\frac{1}{2}}^r u_{i+\frac{1}{2}}^r \end{pmatrix}, \quad \vec{q}_{i+\frac{1}{2}}^* = \begin{pmatrix} \tilde{h}_{i+\frac{1}{2}} \\ \tilde{h}_{i+\frac{1}{2}} \tilde{u}_{i+\frac{1}{2}} \end{pmatrix} \quad (3.55)$$

and

$$\tilde{u}_{i+\frac{1}{2}} := \frac{\sqrt{h_{i+\frac{1}{2}}^l} u_{i+\frac{1}{2}}^l + \sqrt{h_{i+\frac{1}{2}}^r} u_{i+\frac{1}{2}}^r}{\sqrt{h_{i+\frac{1}{2}}^l} + \sqrt{h_{i+\frac{1}{2}}^r}}, \quad \tilde{h}_{i+\frac{1}{2}} := \frac{h_{i+\frac{1}{2}}^l + h_{i+\frac{1}{2}}^r}{2}, \quad \tilde{c}_{i+\frac{1}{2}} := \sqrt{g \tilde{h}_{i+\frac{1}{2}}}. \quad (3.56)$$

3.2.4.3 HLL-type solvers

The problem of evaluating a (system of) partial differential equation(s) at a single discontinuity, for example as in our case evaluating the flux $\vec{F}_{i+\frac{1}{2}} = \vec{F}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r)$ at each inter cell boundary $x_{i+\frac{1}{2}}$, is known as a *Riemann problem*. HLL-type solvers, named after its creators Harten, Lax and Van Leer [7], obtain an expression for the intercell flux directly, unlike the previously discussed schemes. For these solvers, the main difficulty is approximating the wave speeds at each cell boundary.

For the derivation of the HLL scheme we follow Toro [21]. An HLL-type solver aims to solve the *Riemann problem* at an intercell boundary, which is given by

$$\begin{aligned} \vec{q}_t(x, t) + \vec{F}(\vec{q}(x, t))_x &= \vec{0}, \\ \vec{q}(x, 0) &= \begin{cases} \vec{q}^l & \text{if } x \leq 0, \\ \vec{q}^r & \text{if } x > 0, \end{cases} \end{aligned} \quad (3.57)$$

where $\vec{q}(x, t)$ and $\vec{F}(\vec{q}(x, t))$ are as given by (2.31), i.e., we consider the one-dimensional shallow water equations but in its homogeneous form. Furthermore, \vec{q}^l and \vec{q}^r are the calculated left and right numerical values at some intercell boundary and time. We consider the control volume $[x^l, x^r] \times [0, T]$ in the $x - t$ plane, where x^l and x^r are such that

$$x^l \leq TS^l \leq 0, \quad x^r \geq TS^r \geq 0, \quad (3.58)$$

where S^l and S^r are the fastest velocities perturbing the initial states \vec{q}^l and \vec{q}^r respectively, i.e., x^l and x^r are such that $\vec{q}(x^l, T) = \vec{q}^l$ and $\vec{q}(x^r, T) = \vec{q}^r$. Then the integral form of (3.57) reads

$$\int_{x^l}^{x^r} \vec{q}(x, T) dx - \int_{x^l}^{x^r} \vec{q}(x, 0) dx + \int_0^T \vec{F}(\vec{q}(x^r, t)) dt - \int_0^T \vec{F}(\vec{q}(x^l, t)) dt = \vec{0}, \quad (3.59)$$

which can be rewritten to

$$\int_{x^l}^{x^r} \vec{q}(x, T) dx = x^r \vec{q}^r - x^l \vec{q}^l + T(\vec{F}(\vec{q}^r) - \vec{F}(\vec{q}^l)). \quad (3.60)$$

Note that it is also possible to split the integral on the left-hand side of (3.60) into three integrals, namely

$$\int_{x^l}^{x^r} \vec{q}(x, T) dx = \int_{x^l}^{TS^l} \vec{q}(x, T) dx + \int_{TS^l}^{TS^r} \vec{q}(x, T) dx + \int_{TS^r}^{x^r} \vec{q}(x, T) dx, \quad (3.61)$$

which can be evaluated as

$$\int_{x^l}^{x^r} \vec{q}(x, T) dx = \int_{TS^l}^{TS^r} \vec{q}(x, T) dx + (TS^l - x^l)\vec{q}^l + (x^r - TS^r)\vec{q}^r. \quad (3.62)$$

Finally, comparing (3.60) and (3.62) yields

$$\int_{TS^l}^{TS^r} \vec{q}(x, T) dx = T(S^r \vec{q}^r - S^l \vec{q}^l + \vec{F}(\vec{q}^r) - \vec{F}(\vec{q}^l)), \quad (3.63)$$

and thus the integral average of the exact solution of the *Riemann problem* as given by (3.57) reads

$$\vec{q}_{\text{HLL}} := \frac{1}{T(S^r - S^l)} \int_{TS^l}^{TS^r} \vec{q}(x, T) dx = \frac{S^r \vec{q}^r - S^l \vec{q}^l + \vec{F}(\vec{q}^r) - \vec{F}(\vec{q}^l)}{S^r - S^l}. \quad (3.64)$$

An illustration of the two waves of speeds S^l and S^r is given in Figure 3.4. We continue by considering the control volume $[0, x^r] \times [0, T]$ instead, then the integral form of (3.57) reads

$$\int_0^{x^r} \vec{q}(x, T) dx - \int_0^{x^r} \vec{q}(x, 0) dx + \int_0^T \vec{F}(\vec{q}(x^r, t)) dt - \int_0^T \vec{F}(\vec{q}(0, t)) dt = \vec{0}, \quad (3.65)$$

which can be rewritten to

$$\int_0^T \vec{F}(\vec{q}(0, t)) dt = \int_0^{TS^r} \vec{q}(x, T) dx + \int_{TS^r}^{x^r} \vec{q}(x, T) dx - \int_0^{x^r} \vec{q}(x, 0) dx + \int_0^T \vec{F}(\vec{q}(x^r, t)) dt, \quad (3.66)$$

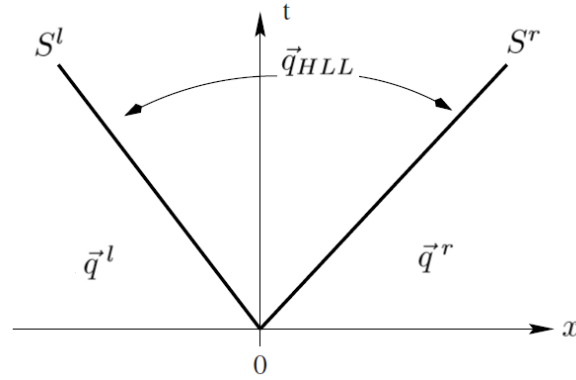


Figure 3.4: Sketch of the two waves of speeds S^l and S^r used in the HLL solver.

which evaluates to

$$\frac{1}{T} \int_0^T \vec{F}(\vec{q}(0, t)) dt = \frac{1}{T} \int_0^{TS^r} \vec{q}(x, T) dx - S^r \vec{q}^r + \vec{F}(\vec{q}^r). \quad (3.67)$$

Lastly, substituting \vec{q}_{HLL} in the integrand on the right-hand side of (3.67) yields

$$\vec{F}_{\text{HLL}} := \frac{1}{T} \int_0^T \vec{F}(\vec{q}(0, t)) dt = S^r (\vec{q}_{\text{HLL}} - \vec{q}^r) + \vec{F}(\vec{q}^r), \quad (3.68)$$

which using the expression for \vec{q}_{HLL} from (3.64) becomes

$$\vec{F}_{\text{HLL}} = \frac{S^r \vec{F}(\vec{q}^l) - S^l \vec{F}(\vec{q}^r) + S^l S^r (\vec{q}^r - \vec{q}^l)}{S^r - S^l} \quad (3.69)$$

and hence the numerical flux is given by

$$\vec{F}_{\text{HLL}}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r) = \begin{cases} \vec{F}(\vec{q}_{i+\frac{1}{2}}^l) & \text{if } 0 \leq S_{i+\frac{1}{2}}^l, \\ \frac{S_{i+\frac{1}{2}}^r \vec{F}(\vec{q}_{i+\frac{1}{2}}^l) - S_{i+\frac{1}{2}}^l \vec{F}(\vec{q}_{i+\frac{1}{2}}^r) + S_{i+\frac{1}{2}}^l S_{i+\frac{1}{2}}^r (\vec{q}_{i+\frac{1}{2}}^r - \vec{q}_{i+\frac{1}{2}}^l)}{S_{i+\frac{1}{2}}^r - S_{i+\frac{1}{2}}^l} & \text{if } S_{i+\frac{1}{2}}^l \leq 0 \leq S_{i+\frac{1}{2}}^r, \\ \vec{F}(\vec{q}_{i+\frac{1}{2}}^r) & \text{if } S_{i+\frac{1}{2}}^r \geq 0. \end{cases} \quad (3.70)$$

Thus, the only problem that remains is to choose suitable approximations for the wavespeeds $S_{i+\frac{1}{2}}^l$ and $S_{i+\frac{1}{2}}^r$.

The HLL solver

Toro [20] suggests from experience to take

$$S_{i+\frac{1}{2}}^l = u_{i+\frac{1}{2}}^l - p_{i+\frac{1}{2}}^l \sqrt{gh_{i+\frac{1}{2}}^l}, \quad S_{i+\frac{1}{2}}^r = u_{i+\frac{1}{2}}^r + p_{i+\frac{1}{2}}^r \sqrt{gh_{i+\frac{1}{2}}^r}, \quad (3.71)$$

where we write

$$\vec{q}_{i+\frac{1}{2}}^l = \begin{bmatrix} h_{i+\frac{1}{2}}^l \\ h_{i+\frac{1}{2}}^l u_{i+\frac{1}{2}}^l \end{bmatrix}, \quad \vec{q}_{i+\frac{1}{2}}^r = \begin{bmatrix} h_{i+\frac{1}{2}}^r \\ h_{i+\frac{1}{2}}^r u_{i+\frac{1}{2}}^r \end{bmatrix}, \quad (3.55)$$

and we define

$$p_{i+\frac{1}{2}}^K := \begin{cases} \frac{1}{h_{i+\frac{1}{2}}^K} \sqrt{\frac{1}{2} \hat{h}_{i+\frac{1}{2}} (\hat{h}_{i+\frac{1}{2}} + h_{i+\frac{1}{2}}^K)} & \text{if } \hat{h}_{i+\frac{1}{2}} > h_{i+\frac{1}{2}}^K, \\ 1 & \text{if } \hat{h}_{i+\frac{1}{2}} \leq h_{i+\frac{1}{2}}^K, \end{cases} \quad (3.72)$$

where

$$\hat{h}_{i+\frac{1}{2}} := \frac{1}{g} \left[\frac{1}{2} (\sqrt{gh_{i+\frac{1}{2}}^l} + \sqrt{gh_{i+\frac{1}{2}}^r}) + \frac{1}{4} (u_{i+\frac{1}{2}}^l - u_{i+\frac{1}{2}}^r) \right]^2. \quad (3.73)$$

The HLLC solver

Leveque [11] generalizesinfeldt's approach [4] for the wavespeed approximations, which was originally in the context of gas dynamics. This approach states that one should take

$$S_{i+\frac{1}{2}}^l = \min_{p=1,2}(\min(\lambda_i^p, \tilde{\lambda}_{i+\frac{1}{2}}^p)), \quad S_{i+\frac{1}{2}}^r = \max_{p=1,2}(\max(\lambda_{i+1}^p, \tilde{\lambda}_{i+\frac{1}{2}}^p)), \quad (3.74)$$

where λ_j^p is the p 'th eigenvalue of the Jacobian matrix $\mathbf{A}(\vec{q}_j)$ of $\vec{F}(\vec{q}_j)$ as given by (4.24) and $\tilde{\lambda}_{i+\frac{1}{2}}^p$ is the p 'th eigenvalue of the Roe matrix $\mathbf{Q}_{\text{Roe}}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r)$ as given by (3.54). Since the states \vec{q}_0 and \vec{q}_{M+1} are unknown, we suggest to take

$$S_{i+\frac{1}{2}}^l = \min_{p=1,2}(\min(\lambda_i^{l,p}, \tilde{\lambda}_{i+\frac{1}{2}}^p)), \quad S_{i+\frac{1}{2}}^r = \max_{p=1,2}(\max(\lambda_{i+1}^{r,p}, \tilde{\lambda}_{i+\frac{1}{2}}^p)), \quad (3.75)$$

where now alternatively $\lambda_j^{l,p}$ is the p 'th eigenvalue of the Jacobian matrix $\mathbf{A}(\vec{q}_{i+\frac{1}{2}}^l)$ of $\vec{F}(\vec{q}_{i+\frac{1}{2}}^l)$ and $\lambda_j^{r,p}$ is the p 'th eigenvalue of the Jacobian matrix $\mathbf{A}(\vec{q}_{i+\frac{1}{2}}^r)$ of $\vec{F}(\vec{q}_{i+\frac{1}{2}}^r)$.

3.2.5 Source term discretization

Having obtained methods of calculating $F_{i+\frac{1}{2}}$, we need not forget about the source term S_i . We propose to discretise the second term of S_i as follows:

$$S_{i,2} = \frac{1}{\Delta x} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} -gh(x,t)b_x(x) dx \approx -\frac{g(b_{i+1} - b_{i-1})}{2\Delta x} \left(h_{i-\frac{1}{2}}^r + h_{i+\frac{1}{2}}^l \right), \quad (3.76)$$

where $b_i := b(x_i)$. Note that the discretisation above is second order accurate in space, which is in accordance with the second-order limited scheme.

3.2.6 Time integration

Now that we have obtained expressions for the fluxes in the finite volume equation (3.15), we need to employ a time integration method to calculate the numerical values. For the time integration we employ the RK3b scheme [5] which has been shown to be monotone [8], this is given by

$$\vec{q}_i^{n+1} = \vec{q}_i^n + \frac{1}{6} \left(\vec{k}_1 + \vec{k}_2 + 4\vec{k}_3 \right), \quad (3.77)$$

where we define the following function:

$$\vec{K}(\vec{q}_i) = \Delta t \left(-\frac{1}{\Delta x} \left(\vec{F}_{i+\frac{1}{2}} - \vec{F}_{i-\frac{1}{2}} \right) + \vec{S}_i \right), \quad (3.78)$$

where we calculate

$$\vec{k}_1 = \vec{K}(\vec{q}_i), \quad (3.79a)$$

$$\vec{k}_2 = \vec{K}(\vec{q}_i + \vec{k}_1), \quad (3.79b)$$

$$\vec{k}_3 = \vec{K}(\vec{q}_i + \frac{1}{4}\vec{k}_1 + \frac{1}{4}\vec{k}_2). \quad (3.79c)$$

In order to ensure that the waves from two cell centres do not interact, we require that

$$\Delta t \max(|u \pm \sqrt{gh}|) < \frac{\Delta x}{2}, \quad (3.80)$$

and thus we require for the time step

$$\Delta t < c \cdot \frac{\Delta x}{2 \max_{i=0,\dots,M} (|u_{i\pm\frac{1}{2}}| + \sqrt{gh_{i+\frac{1}{2}}})}, \quad (3.81)$$

where $0 < c < 1$.

4 Numerical methods in the 2D case

4.1 Derivation of the 2D finite volume method

Let us consider the two-dimensional shallow water equations as given by (2.28). Similar to the one-dimensional case in Section 3.2.2, the finite volume equation can be derived in the two-dimensional case as well. Let us consider the space $[x_L, x_R] \times [y_L, y_R] \times [0, T]$ and subdivide the $x - y$ plane into equally sized finite volumes/cells, where each cell $I_{i,j}$ ($i = 1, \dots, M, j = 1, \dots, N$) is given by

$$I_{i,j} = [x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}] \times [y_{j-\frac{1}{2}}, y_{j+\frac{1}{2}}], \quad (4.1)$$

and the area of each cell is $\Delta x \cdot \Delta y$ where

$$\Delta x := x_{i+\frac{1}{2}} - x_{i-\frac{1}{2}} = \frac{x_R - x_L}{M}, \quad \Delta y := y_{j+\frac{1}{2}} - y_{j-\frac{1}{2}} = \frac{y_R - y_L}{N}. \quad (4.2)$$

Consider the control volume $[x_{i-\frac{1}{2}}, x_{i+\frac{1}{2}}] \times [y_{j-\frac{1}{2}}, y_{j+\frac{1}{2}}] \times [t_n, t_{n+1}]$, then by integrating (2.26) we obtain

$$\begin{aligned} & \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} \bar{q}_t(x, y, t) \, dx \, dy + \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} \vec{F}(\bar{q}(x, y, t))_x \, dx \, dy \\ & + \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} \vec{G}(\bar{q}(x, y, t))_y \, dx \, dy = \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} \vec{S}(\bar{q}(x, y, t)) \, dx \, dy. \end{aligned} \quad (4.3)$$

Since the cell $I_{i,j}$ remains unchanged in time the Leibniz Integration Rule can be applied to the first term above and the Fundamental Theorem of Calculus can be applied to both the second term and third term, which yields

$$\begin{aligned} & \frac{d}{dt} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} \bar{q}(x, y, t) \, dx \, dy + \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} \left(\vec{F}(\bar{q}(x_{i+\frac{1}{2}}, y, t)) - \vec{F}(\bar{q}(x_{i-\frac{1}{2}}, y, t)) \right) dy \\ & + \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \left(\vec{G}(\bar{q}(x, y_{j+\frac{1}{2}}, t)) - \vec{G}(\bar{q}(x, y_{j-\frac{1}{2}}, t)) \right) dx = \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} \vec{S}(\bar{q}(x, y, t)) \, dx \, dy. \end{aligned} \quad (4.4)$$

The above can be rewritten into the finite volume equation

$$\frac{d\bar{q}_{i,j}}{dt} + \frac{1}{\Delta x} \left(\vec{F}_{i+\frac{1}{2},j} - \vec{F}_{i-\frac{1}{2},j} \right) + \frac{1}{\Delta y} \left(\vec{G}_{i,j+\frac{1}{2}} - \vec{G}_{i,j-\frac{1}{2}} \right) = \vec{S}_{i,j}, \quad (4.5)$$

where we define the following averages:

$$\begin{aligned} \bar{q}_{i,j} &:= \frac{1}{\Delta x \Delta y} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} \bar{q}(x, y, t) \, dx \, dy, & \vec{F}_{i+\frac{1}{2},j} &:= \frac{1}{\Delta y} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} \vec{F}(\bar{q}(x_{i+\frac{1}{2}}, y, t)) \, dy, \\ \vec{G}_{i,j+\frac{1}{2}} &:= \frac{1}{\Delta x} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \vec{G}(\bar{q}(x, y_{j+\frac{1}{2}}, t)) \, dx, & \vec{S}_{i,j} &:= \frac{1}{\Delta x \Delta y} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} \vec{S}(\bar{q}(x, y, t)) \, dx \, dy. \end{aligned} \quad (4.6)$$

In this case, we have that $\bar{q}_{i,j}$ is the *cell average*. In our numerical method, we assume the numerical solution $\bar{q}_{i,j}$ to be constant in space in each cell $I_{i,j}$.

4.2 State interpolation

Similar to the one-dimensional case, the numerical scheme to simulate the two-dimensional equations will use the finite volume equation given by (4.5) to calculate the numerical approximation $\vec{q}_{i,j}$ in each cell $I_{i,j}$. In addition to evaluating the fluxes $\vec{F}_{\frac{1}{2},j}, \vec{F}_{\frac{3}{2},j}, \dots, \vec{F}_{M+\frac{1}{2},j}$ ($j = 1, 2, \dots, N$) in the finite volume equation (4.5), we also need to evaluate the fluxes $\vec{G}_{i,\frac{1}{2}}, \vec{G}_{i,\frac{3}{2}}, \dots, \vec{G}_{i,N+\frac{1}{2}}$ ($i = 1, 2, \dots, M$).

In order to evaluate these fluxes, one needs the numerical solutions of \vec{q} at both $(x_{\frac{1}{2}}, y_j), (x_{\frac{3}{2}}, y_j), \dots, (x_{M+\frac{1}{2}}, y_j)$ ($j = 1, 2, \dots, N$), which we denote by $\vec{q}_{\frac{1}{2},j}, \vec{q}_{\frac{3}{2},j}, \dots, \vec{q}_{M+\frac{1}{2},j}$, and $(x_i, y_{\frac{1}{2}}), (x_i, y_{\frac{3}{2}}), \dots, (x_i, y_{M+\frac{1}{2}})$ ($i = 1, 2, \dots, M$), which we denote by $\vec{q}_{i,\frac{1}{2}}, \vec{q}_{i,\frac{3}{2}}, \dots, \vec{q}_{i,M+\frac{1}{2}}$. The scenario from the one-dimensional case carries over, each cell is very likely to have a different numerical solution than its neighbours. Hence, the numerical solution $\vec{q}_{i,j}$ is most often discontinuous at its four boundaries and the values $\vec{q}_{i-\frac{1}{2},j}, \vec{q}_{i+\frac{1}{2},j}, \vec{q}_{i,j-\frac{1}{2}}, \vec{q}_{i,j+\frac{1}{2}}$ are not straightforward. In order to solve this issue we generalize our previous approach from the one-dimensional case by setting

$$\vec{F}_{i+\frac{1}{2},j} = \vec{F}(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r), \quad \vec{G}_{i,j+\frac{1}{2}} = \vec{G}(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r) \quad (4.7)$$

where $\vec{q}_{i+\frac{1}{2},j}^l$ is the numerical solution to the left of the right boundary of cell $I_{i,j}$ and $\vec{q}_{i+\frac{1}{2},j}^r$ is the numerical solution to the right of the right boundary of cell $I_{i,j}$. Figure 4.1 illustrates the precise lay-out of each cell $I_{i,j}$. Note that $\vec{F}(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r)$ and $\vec{G}(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r)$ are the flux evaluation functions which are yet to be chosen.

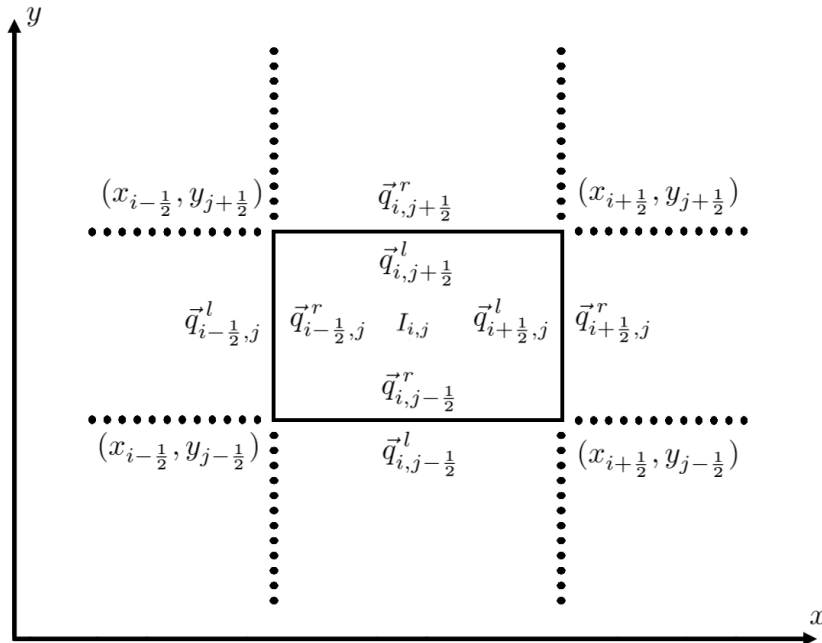


Figure 4.1: Top-down view of a two-dimensional cell.

4.2.1 First-order upwind

The first order upwind method from Section 3.2.3.1 can be generalized for the two-dimensional case to

$$\vec{q}_{i+\frac{1}{2},j}^l = \vec{q}_{i,j}, \quad (4.8a)$$

$$\vec{q}_{i+\frac{1}{2},j}^r = \vec{q}_{i+1,j}, \quad (4.8b)$$

$$\vec{q}_{i,j+\frac{1}{2}}^l = \vec{q}_{i,j}, \quad (4.8c)$$

$$\vec{q}_{i,j+\frac{1}{2}}^r = \vec{q}_{i,j+1}, \quad (4.8d)$$

which again is first-order accurate in space.

4.2.2 Second-order upwind

The method by Van Leer [10] from Section 3.2.3.2 can be generalized to

$$\vec{q}_{i+\frac{1}{2},j}^l = \vec{q}_{i,j} + \frac{1+\kappa}{4}(\vec{q}_{i+1,j} - \vec{q}_{i,j}) + \frac{1-\kappa}{4}(\vec{q}_{i,j} - \vec{q}_{i-1,j}), \quad (4.9a)$$

$$\vec{q}_{i+\frac{1}{2},j}^r = \vec{q}_{i+1,j} + \frac{1+\kappa}{4}(\vec{q}_{i,j} - \vec{q}_{i+1,j}) + \frac{1-\kappa}{4}(\vec{q}_{i+1,j} - \vec{q}_{i+2,j}), \quad (4.9b)$$

$$\vec{q}_{i,j+\frac{1}{2}}^l = \vec{q}_{i,j} + \frac{1+\kappa}{4}(\vec{q}_{i,j+1} - \vec{q}_{i,j}) + \frac{1-\kappa}{4}(\vec{q}_{i,j} - \vec{q}_{i,j-1}), \quad (4.9c)$$

$$\vec{q}_{i,j+\frac{1}{2}}^r = \vec{q}_{i,j+1} + \frac{1+\kappa}{4}(\vec{q}_{i,j} - \vec{q}_{i,j+1}) + \frac{1-\kappa}{4}(\vec{q}_{i,j+1} - \vec{q}_{i,j+2}), \quad (4.9d)$$

where κ is a parameter to be chosen in the range $[-1, 1]$.

4.2.3 Second-order limited scheme

The second-order limited scheme from Section 3.2.3.3 is in the two-dimensional case given by

$$\vec{q}_{i+\frac{1}{2},j}^l = \vec{q}_{i,j} + \frac{1}{2}\phi(r_{i+\frac{1}{2},j}^l)(\vec{q}_{i,j} - \vec{q}_{i-1,j}), \quad (4.10a)$$

$$\vec{q}_{i+\frac{1}{2},j}^r = \vec{q}_{i+1,j} + \frac{1}{2}\phi(r_{i+\frac{1}{2},j}^r)(\vec{q}_{i+1,j} - \vec{q}_{i+2,j}), \quad (4.10b)$$

$$\vec{q}_{i,j+\frac{1}{2}}^l = \vec{q}_{i,j} + \frac{1}{2}\phi(r_{i,j+\frac{1}{2}}^l)(\vec{q}_{i,j} - \vec{q}_{i,j-1}), \quad (4.10c)$$

$$\vec{q}_{i,j+\frac{1}{2}}^r = \vec{q}_{i,j+1} + \frac{1}{2}\phi(r_{i,j+\frac{1}{2}}^r)(\vec{q}_{i,j+1} - \vec{q}_{i,j+2}), \quad (4.10d)$$

where

$$r_{i+\frac{1}{2},j}^l = \frac{\vec{q}_{i+1,j} - \vec{q}_{i,j}}{\vec{q}_{i,j} - \vec{q}_{i-1,j}}, \quad r_{i+\frac{1}{2},j}^r = \frac{\vec{q}_{i,j} - \vec{q}_{i+1,j}}{\vec{q}_{i+1,j} - \vec{q}_{i+2,j}}, \quad r_{i,j+\frac{1}{2}}^l = \frac{\vec{q}_{i,j+1} - \vec{q}_{i,j}}{\vec{q}_{i,j} - \vec{q}_{i,j-1}}, \quad r_{i,j+\frac{1}{2}}^r = \frac{\vec{q}_{i,j} - \vec{q}_{i,j+1}}{\vec{q}_{i,j+1} - \vec{q}_{i,j+2}}, \quad (4.11)$$

with the *limiter function* $\phi(r)$ yet to be chosen, various limiters are given in Section 3.2.3.4. The numerical implementation of this scheme, in order to be able to work with division by zero cases, is explained in Appendix A.

4.3 Flux evaluation

Similar to the one-dimensional case, the next step is to determine the flux evaluation functions $\vec{F}(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r)$ and $\vec{G}(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r)$. We generalize some of the methods from the one-dimensional case and discuss new ones as well.

4.3.1 Flux vector splitting

The flux vector splitting scheme for the one-dimensional case in Section 3.2.4.1 can be extended to the two-dimensional case using the same ideas. In this case, one can determine if the homogeneity property is satisfied for both fluxes, these are

$$\mathbf{A}(\vec{q})\vec{q} = \vec{F}(\vec{q}), \quad \mathbf{B}(\vec{q})\vec{q} = \vec{G}(\vec{q}), \quad (4.12)$$

where $\vec{F}(\vec{q})$ and $\vec{G}(\vec{q})$ are as given by (2.28), $\mathbf{A}(\vec{q})$ is the Jacobian matrix of $\vec{F}(\vec{q})$ and $\mathbf{B}(\vec{q})$ is the Jacobian matrix of $\vec{G}(\vec{q})$. Unfortunately, the two-dimensional shallow water equations do not satisfy the properties given in (4.12). However, using the techniques from the one-dimensional case, it is possible to construct matrices $\mathbf{A}^*(\vec{q})$ different from the Jacobian matrix $\mathbf{A}(\vec{q})$ and $\mathbf{B}^*(\vec{q})$ different from the Jacobian matrix $\mathbf{B}(\vec{q})$ which do satisfy (4.12), these are given by

$$\mathbf{A}^*(\vec{q}) = \begin{bmatrix} 0 & 1 & 0 \\ -\frac{q_2^2}{q_1^2} + \frac{gq_1}{2} & \frac{2q_2}{q_1} & 0 \\ -\frac{q_2q_3}{q_1^2} & \frac{q_3}{q_1} & \frac{q_2}{q_1} \end{bmatrix}, \quad \mathbf{B}^*(\vec{q}) = \begin{bmatrix} 0 & 0 & 1 \\ -\frac{q_2q_3}{q_1^2} & \frac{q_3}{q_1} & \frac{q_2}{q_1} \\ -\frac{q_3^2}{q_1^2} + \frac{gq_1}{2} & 0 & \frac{2q_3}{q_1} \end{bmatrix}. \quad (4.13)$$

Now, using these matrices $\mathbf{A}^*(\vec{q})$ and $\mathbf{B}^*(\vec{q})$ decompose them as

$$\mathbf{A}^* = \mathbf{R}\mathbf{\Lambda}^*\mathbf{R}^{-1}, \quad \mathbf{B}^* = \mathbf{S}\mathbf{\Sigma}^*\mathbf{S}^{-1}, \quad (4.14)$$

where $\mathbf{\Lambda}^*$ is the diagonal matrix whose diagonal elements are the eigenvalues of \mathbf{A}^* , \mathbf{R} is the matrix whose columns are the eigenvectors of \mathbf{A}^* and \mathbf{R}^{-1} is the inverse of \mathbf{R} . Similarly, $\mathbf{\Sigma}^*$ is the diagonal matrix whose diagonal elements are the eigenvalues of \mathbf{B}^* , \mathbf{S} is the matrix whose columns are the eigenvectors of \mathbf{B}^* and \mathbf{S}^{-1} is the inverse of \mathbf{S} . It can be shown that

$$\mathbf{\Lambda}^*(\vec{q}) = \begin{pmatrix} \lambda_1^* & 0 & 0 \\ 0 & \lambda_2^* & 0 \\ 0 & 0 & \lambda_3^* \end{pmatrix}, \quad \mathbf{R}(\vec{q}) = \begin{pmatrix} 1 & 0 & 1 \\ \lambda_1^* & 0 & \lambda_3^* \\ \frac{q_3}{q_1} & 1 & \frac{q_3}{q_1} \end{pmatrix}, \quad (4.15)$$

where

$$\lambda_1^*(\vec{q}) = \frac{q_2}{q_1} - \sqrt{\frac{gq_1}{2}}, \quad \lambda_2^*(\vec{q}) = \frac{q_2}{q_1}, \quad \lambda_3^*(\vec{q}) = \frac{q_2}{q_1} + \sqrt{\frac{gq_1}{2}}, \quad (4.16)$$

and

$$\mathbf{\Sigma}^*(\vec{q}) = \begin{pmatrix} \sigma_1^* & 0 & 0 \\ 0 & \sigma_2^* & 0 \\ 0 & 0 & \sigma_3^* \end{pmatrix}, \quad \mathbf{S}(\vec{q}) = \begin{pmatrix} 1 & 0 & 1 \\ \frac{q_2}{q_1} & 1 & \frac{q_2}{q_1} \\ \sigma_1^* & 0 & \sigma_3^* \end{pmatrix}, \quad (4.17)$$

where

$$\sigma_1^*(\vec{q}) = \frac{q_3}{q_1} - \sqrt{\frac{gq_1}{2}}, \quad \sigma_2^*(\vec{q}) = \frac{q_3}{q_1}, \quad \sigma_3^*(\vec{q}) = \frac{q_3}{q_1} + \sqrt{\frac{gq_1}{2}}. \quad (4.18)$$

Similar to the 1D case, the reader should note that the wave speeds λ_1^* , λ_3^* , σ_1^* and σ_3^* are not physically correct and distinct to the eigenvalues of the matrices \mathbf{A} and \mathbf{B} . We split the matrices \mathbf{A}^* and \mathbf{B}^* in respective positive and negative parts, given by

$$\mathbf{A}_+^* := \mathbf{R}\mathbf{\Lambda}_+^*\mathbf{R}^{*-1}, \quad \mathbf{A}_-^* := \mathbf{R}\mathbf{\Lambda}_-^*\mathbf{R}^{-1}, \quad \mathbf{B}_+^* := \mathbf{S}\mathbf{\Sigma}_+^*\mathbf{S}^{*-1}, \quad \mathbf{B}_-^* := \mathbf{S}\mathbf{\Sigma}_-^*\mathbf{S}^{-1} \quad (4.19)$$

where

$$\mathbf{\Lambda}_+^* := \begin{pmatrix} \max(\lambda_1^*, 0) & 0 & 0 \\ 0 & \max(\lambda_2^*, 0) & 0 \\ 0 & 0 & \max(\lambda_3^*, 0) \end{pmatrix}, \quad \mathbf{\Lambda}_-^* := \begin{pmatrix} \min(\lambda_1^*, 0) & 0 & 0 \\ 0 & \min(\lambda_2^*, 0) & 0 \\ 0 & 0 & \min(\lambda_3^*, 0) \end{pmatrix}, \quad (4.20)$$

and

$$\Sigma_+^* := \begin{pmatrix} \max(\sigma_1^*, 0) & 0 & 0 \\ 0 & \max(\sigma_2^*, 0) & 0 \\ 0 & 0 & \max(\sigma_3^*, 0) \end{pmatrix}, \quad \Sigma_-^* := \begin{pmatrix} \min(\sigma_1^*, 0) & 0 & 0 \\ 0 & \min(\sigma_2^*, 0) & 0 \\ 0 & 0 & \min(\sigma_3^*, 0) \end{pmatrix}. \quad (4.21)$$

Here it holds that

$$\Lambda^* = \Lambda_+^* + \Lambda_-^*, \quad \mathbf{A}^* = \mathbf{A}_+^* + \mathbf{A}_-^*, \quad \Sigma^* = \Sigma_+^* + \Sigma_-^*, \quad \mathbf{B}^* = \mathbf{B}_+^* + \mathbf{B}_-^* \quad (4.22)$$

Finally, we take the flux evaluation functions as

$$\vec{\mathcal{F}}_{\text{FVS}}(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r) := \mathbf{A}_+^*(\vec{q}_{i+\frac{1}{2},j}^l) \vec{q}_{i+\frac{1}{2},j}^l + \mathbf{A}_-^*(\vec{q}_{i+\frac{1}{2},j}^r) \vec{q}_{i+\frac{1}{2},j}^r, \quad (4.23a)$$

$$\vec{\mathcal{G}}_{\text{FVS}}(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r) := \mathbf{B}_+^*(\vec{q}_{i,j+\frac{1}{2}}^l) \vec{q}_{i,j+\frac{1}{2}}^l + \mathbf{B}_-^*(\vec{q}_{i,j+\frac{1}{2}}^r) \vec{q}_{i,j+\frac{1}{2}}^r. \quad (4.23b)$$

4.3.2 Flux difference splitting

In this section we extend the flux difference scheme for the one-dimensional case in Section 3.2.4.2 to the two-dimensional case.

First, one needs the Jacobian matrices $\mathbf{A}(\vec{q})$ of $\vec{F}(\vec{q})$ and $\mathbf{B}(\vec{q})$ of $\vec{G}(\vec{q})$, where $\vec{F}(\vec{q})$ and $\vec{G}(\vec{q})$ are as given by (2.28), these are

$$\mathbf{A}(\vec{q}) = \begin{pmatrix} 0 & 1 & 0 \\ -\frac{q_2^2}{q_1^2} + gq_1 & \frac{2q_2}{q_1} & 0 \\ -\frac{q_2q_3}{q_1^2} & \frac{q_3}{q_1} & \frac{q_2}{q_1} \end{pmatrix}, \quad \mathbf{B}(\vec{q}) = \begin{pmatrix} 0 & 0 & 1 \\ -\frac{q_2q_3}{q_1^2} & \frac{q_3}{q_1} & \frac{q_2}{q_1} \\ -\frac{q_3^2}{q_1^2} + gq_1 & 0 & \frac{2q_3}{q_1} \end{pmatrix}. \quad (4.24)$$

Similar to the flux vector splitting, we decompose the matrices $\mathbf{A}(\vec{q})$ and $\mathbf{B}(\vec{q})$ as

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}, \quad \mathbf{B} = \mathbf{W}\mathbf{\Sigma}\mathbf{W}^{-1}, \quad (4.25)$$

where $\mathbf{\Lambda}$ is the diagonal matrix whose diagonal elements are the eigenvalues of \mathbf{A} , \mathbf{V} is the matrix whose columns are the eigenvectors of \mathbf{A} and \mathbf{V}^{-1} is the inverse of \mathbf{S} . Furthermore, we have that

$$\mathbf{\Lambda}(\vec{q}) = \begin{pmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{pmatrix}, \quad \mathbf{V}(\vec{q}) = \begin{pmatrix} 1 & 0 & 1 \\ \lambda_1 & 0 & \lambda_3 \\ \frac{q_3}{q_1} & 1 & \frac{q_3}{q_1} \end{pmatrix}, \quad (4.26)$$

where

$$\lambda_1(\vec{q}) = \frac{q_2}{q_1} - \sqrt{gq_1}, \quad \lambda_2(\vec{q}) = \frac{q_2}{q_1}, \quad \lambda_3(\vec{q}) = \frac{q_2}{q_1} + \sqrt{gq_1}, \quad (4.27)$$

and

$$\mathbf{\Sigma}(\vec{q}) = \begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{pmatrix}, \quad \mathbf{S}(\vec{q}) = \begin{pmatrix} 1 & 0 & 1 \\ \frac{q_2}{q_1} & 1 & \frac{q_2}{q_1} \\ \sigma_1 & 0 & \sigma_3 \end{pmatrix}, \quad (4.28)$$

where

$$\sigma_1(\vec{q}) = \frac{q_3}{q_1} - \sqrt{gq_1}, \quad \sigma_2(\vec{q}) = \frac{q_3}{q_1}, \quad \sigma_3(\vec{q}) = \frac{q_3}{q_1} + \sqrt{gq_1}. \quad (4.29)$$

Similar to before, we now split the matrices \mathbf{A} and \mathbf{B} in positive and negative parts, given by

$$\mathbf{A}^+ := \mathbf{V}\mathbf{\Lambda}^+\mathbf{V}^{-1}, \quad \mathbf{A}^- := \mathbf{V}\mathbf{\Lambda}^-\mathbf{V}^{-1}, \quad \mathbf{B}^+ := \mathbf{W}\mathbf{\Sigma}^+\mathbf{W}^{-1}, \quad \mathbf{B}^- := \mathbf{W}\mathbf{\Sigma}^-\mathbf{W}^{-1}, \quad (4.30)$$

where

$$\mathbf{\Lambda}^+ := \begin{pmatrix} \max(\lambda_1, 0) & 0 & 0 \\ 0 & \max(\lambda_2, 0) & 0 \\ 0 & 0 & \max(\lambda_3, 0) \end{pmatrix}, \quad \mathbf{\Lambda}^- := \begin{pmatrix} \min(\lambda_1, 0) & 0 & 0 \\ 0 & \min(\lambda_2, 0) & 0 \\ 0 & 0 & \min(\lambda_3, 0) \end{pmatrix}, \quad (4.31)$$

and

$$\mathbf{\Sigma}^+ := \begin{pmatrix} \max(\sigma_1, 0) & 0 & 0 \\ 0 & \max(\sigma_2, 0) & 0 \\ 0 & 0 & \max(\sigma_3, 0) \end{pmatrix}, \quad \mathbf{\Sigma}^- := \begin{pmatrix} \min(\sigma_1, 0) & 0 & 0 \\ 0 & \min(\sigma_2, 0) & 0 \\ 0 & 0 & \min(\sigma_3, 0) \end{pmatrix}. \quad (4.32)$$

Again it holds that

$$\mathbf{\Lambda} = \mathbf{\Lambda}^+ + \mathbf{\Lambda}^-, \quad \mathbf{A} = \mathbf{A}^+ + \mathbf{A}^-, \quad \mathbf{\Sigma} = \mathbf{\Sigma}^+ + \mathbf{\Sigma}^-, \quad \mathbf{B} = \mathbf{B}^+ + \mathbf{B}^- \quad (4.33)$$

Similar to the one-dimensional case, the flux difference splitting schemes are of the form

$$\vec{F}(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r) = \frac{1}{2} \left(\vec{F}(\vec{q}_{i+\frac{1}{2},j}^l) + \vec{F}(\vec{q}_{i+\frac{1}{2},j}^r) \right) - \frac{1}{2} \vec{d}_F(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r), \quad (4.34a)$$

$$\vec{G}(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r) = \frac{1}{2} \left(\vec{G}(\vec{q}_{i,j+\frac{1}{2}}^l) + \vec{G}(\vec{q}_{i,j+\frac{1}{2}}^r) \right) - \frac{1}{2} \vec{d}_G(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r), \quad (4.34b)$$

where

$$\vec{d}_F(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r) := \left| \mathbf{Q}_F(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r) \right| (\vec{q}_{i+\frac{1}{2},j}^r - \vec{q}_{i+\frac{1}{2},j}^l), \quad (4.35a)$$

$$\vec{d}_G(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r) := \left| \mathbf{Q}_G(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r) \right| (\vec{q}_{i,j+\frac{1}{2}}^r - \vec{q}_{i,j+\frac{1}{2}}^l), \quad (4.35b)$$

and the consistency requirement

$$d_F(\vec{q}, \vec{q}) = d_G(\vec{q}, \vec{q}) = 0 \quad (4.36)$$

is always satisfied.

The midpoint rule

The simplest option for the matrices $\mathbf{Q}_F(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r)$ and $\mathbf{Q}_F(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r)$ in (4.35) are taking the arithmetic mean as given by (3.44), which is

$$\mathbf{Q}_{F,MR}(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r) = \mathbf{A} \left(\frac{1}{2} (\vec{q}_{i+\frac{1}{2},j}^l + \vec{q}_{i+\frac{1}{2},j}^r) \right), \quad (4.37a)$$

$$\mathbf{Q}_{G,MR}(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r) = \mathbf{B} \left(\frac{1}{2} (\vec{q}_{i,j+\frac{1}{2}}^l + \vec{q}_{i,j+\frac{1}{2}}^r) \right), \quad (4.37b)$$

and hence the flux evaluation function is given by

$$\vec{F}_{MR}(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r) = \frac{1}{2} (\vec{F}(\vec{q}_{i+\frac{1}{2},j}^l) + \vec{F}(\vec{q}_{i+\frac{1}{2},j}^r)) - \frac{1}{2} \left| \mathbf{A} \left(\frac{1}{2} (\vec{q}_{i+\frac{1}{2},j}^l + \vec{q}_{i+\frac{1}{2},j}^r) \right) \right| (\vec{q}_{i+\frac{1}{2},j}^r - \vec{q}_{i+\frac{1}{2},j}^l), \quad (4.38a)$$

$$\vec{G}_{MR}(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r) = \frac{1}{2} (\vec{F}(\vec{q}_{i,j+\frac{1}{2}}^l) + \vec{F}(\vec{q}_{i,j+\frac{1}{2}}^r)) - \frac{1}{2} \left| \mathbf{B} \left(\frac{1}{2} (\vec{q}_{i,j+\frac{1}{2}}^l + \vec{q}_{i,j+\frac{1}{2}}^r) \right) \right| (\vec{q}_{i,j+\frac{1}{2}}^r - \vec{q}_{i,j+\frac{1}{2}}^l). \quad (4.38b)$$

The trapezoidal rule

Another straightforward option for the matrix $|\mathbf{Q}(\vec{q}_{i+\frac{1}{2}}^l, \vec{q}_{i+\frac{1}{2}}^r)|$ in (4.35) is taking

$$\left| \mathbf{Q}_{F,TR}(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r) \right| = \frac{1}{2} (|\mathbf{A}(\vec{q}_{i+\frac{1}{2},j}^l)| + |\mathbf{A}(\vec{q}_{i+\frac{1}{2},j}^r)|), \quad (4.39a)$$

$$\left| \mathbf{Q}_{G,TR}(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r) \right| = \frac{1}{2} (|\mathbf{B}(\vec{q}_{i,j+\frac{1}{2}}^l)| + |\mathbf{B}(\vec{q}_{i,j+\frac{1}{2}}^r)|), \quad (4.39b)$$

and thus the numerical flux is given by

$$\vec{F}_{TR}(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r) = \frac{1}{2} (\vec{F}(\vec{q}_{i+\frac{1}{2},j}^l) + \vec{F}(\vec{q}_{i+\frac{1}{2},j}^r)) - \frac{1}{4} (|\mathbf{A}(\vec{q}_{i+\frac{1}{2},j}^l)| + |\mathbf{A}(\vec{q}_{i+\frac{1}{2},j}^r)|) (\vec{q}_{i+\frac{1}{2},j}^r - \vec{q}_{i+\frac{1}{2},j}^l), \quad (4.40a)$$

$$\vec{G}_{TR}(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r) = \frac{1}{2} (\vec{F}(\vec{q}_{i,j+\frac{1}{2}}^l) + \vec{F}(\vec{q}_{i,j+\frac{1}{2}}^r)) - \frac{1}{4} (|\mathbf{B}(\vec{q}_{i,j+\frac{1}{2}}^l)| + |\mathbf{B}(\vec{q}_{i,j+\frac{1}{2}}^r)|) (\vec{q}_{i,j+\frac{1}{2}}^r - \vec{q}_{i,j+\frac{1}{2}}^l). \quad (4.40b)$$

Roe's method

Roe's method for the one-dimensional shallow water equations, can be straightforward generalized to the two-dimensional case by taking

$$\mathbf{Q}_{F,Roe}(\vec{q}_{i+\frac{1}{2},j}^l, \vec{q}_{i+\frac{1}{2},j}^r) = \mathbf{A}(\vec{q}_{i+\frac{1}{2},j}^*) = \begin{pmatrix} 0 & 1 & 0 \\ -\tilde{u}_{i+\frac{1}{2},j}^2 + \tilde{c}_{i+\frac{1}{2},j}^2 & 2\tilde{u}_{i+\frac{1}{2},j} & 0 \\ -\tilde{u}_{i+\frac{1}{2},j}\tilde{v}_{i+\frac{1}{2},j} & \tilde{v}_{i+\frac{1}{2},j} & \tilde{u}_{i+\frac{1}{2},j} \end{pmatrix}, \quad (4.41)$$

where

$$\vec{q}_{i+\frac{1}{2},j}^l = \begin{pmatrix} h_{i+\frac{1}{2},j}^l \\ h_{i+\frac{1}{2},j}^l u_{i+\frac{1}{2},j}^l \\ h_{i+\frac{1}{2},j}^l v_{i+\frac{1}{2},j}^l \end{pmatrix}, \quad \vec{q}_{i+\frac{1}{2},j}^r = \begin{pmatrix} h_{i+\frac{1}{2},j}^r \\ h_{i+\frac{1}{2},j}^r u_{i+\frac{1}{2},j}^r \\ h_{i+\frac{1}{2},j}^r v_{i+\frac{1}{2},j}^r \end{pmatrix}, \quad \vec{q}_{i+\frac{1}{2},j}^* = \begin{pmatrix} \tilde{h}_{i+\frac{1}{2},j} \\ \tilde{h}_{i+\frac{1}{2},j} \tilde{u}_{i+\frac{1}{2},j} \\ \tilde{h}_{i+\frac{1}{2},j} \tilde{v}_{i+\frac{1}{2},j} \end{pmatrix}, \quad (4.42)$$

and

$$\tilde{u}_{i+\frac{1}{2},j} := \frac{\sqrt{h_{i+\frac{1}{2},j}^l} u_{i+\frac{1}{2},j}^l + \sqrt{h_{i+\frac{1}{2},j}^r} u_{i+\frac{1}{2},j}^r}{\sqrt{h_{i+\frac{1}{2},j}^l} + \sqrt{h_{i+\frac{1}{2},j}^r}}, \quad \tilde{v}_{i+\frac{1}{2},j} := \frac{\sqrt{h_{i+\frac{1}{2},j}^l} v_{i+\frac{1}{2},j}^l + \sqrt{h_{i+\frac{1}{2},j}^r} v_{i+\frac{1}{2},j}^r}{\sqrt{h_{i+\frac{1}{2},j}^l} + \sqrt{h_{i+\frac{1}{2},j}^r}}, \quad (4.43)$$

$$\tilde{h}_{i+\frac{1}{2},j} := \frac{h_{i+\frac{1}{2},j}^l + h_{i+\frac{1}{2},j}^r}{2}, \quad \tilde{c}_{i+\frac{1}{2},j} := \sqrt{g\tilde{h}_{i+\frac{1}{2},j}}.$$

Similarly, we choose

$$\mathbf{Q}_{G,\text{Roe}}(\vec{q}_{i,j+\frac{1}{2}}^l, \vec{q}_{i,j+\frac{1}{2}}^r) = \mathbf{B}(\vec{q}_{i,j+\frac{1}{2}}^*) = \begin{pmatrix} 0 & 0 & 1 \\ -\tilde{u}_{i,j+\frac{1}{2}} \tilde{v}_{i,j+\frac{1}{2}} & \tilde{v}_{i,j+\frac{1}{2}} & \tilde{u}_{i,j+\frac{1}{2}} \\ -\tilde{v}_{i,j+\frac{1}{2}}^2 + \tilde{c}_{i,j+\frac{1}{2}}^2 & 0 & 2\tilde{v}_{i,j+\frac{1}{2}} \end{pmatrix}, \quad (4.44)$$

where

$$\vec{q}_{i,j+\frac{1}{2}}^l = \begin{pmatrix} h_{i,j+\frac{1}{2}}^l \\ h_{i,j+\frac{1}{2}}^l u_{i,j+\frac{1}{2}}^l \\ h_{i,j+\frac{1}{2}}^l v_{i,j+\frac{1}{2}}^l \end{pmatrix}, \quad \vec{q}_{i,j+\frac{1}{2}}^r = \begin{pmatrix} h_{i,j+\frac{1}{2}}^r \\ h_{i,j+\frac{1}{2}}^r u_{i,j+\frac{1}{2}}^r \\ h_{i,j+\frac{1}{2}}^r v_{i,j+\frac{1}{2}}^r \end{pmatrix}, \quad \vec{q}_{i+\frac{1}{2},j}^* = \begin{pmatrix} \tilde{h}_{i,j+\frac{1}{2}} \\ \tilde{h}_{i,j+\frac{1}{2}} \tilde{u}_{i,j+\frac{1}{2}} \\ \tilde{h}_{i,j+\frac{1}{2}} \tilde{v}_{i,j+\frac{1}{2}} \end{pmatrix}, \quad (4.45)$$

and

$$\tilde{u}_{i,j+\frac{1}{2},j} := \frac{\sqrt{h_{i,j+\frac{1}{2}}^l} u_{i,j+\frac{1}{2}}^l + \sqrt{h_{i,j+\frac{1}{2}}^r} u_{i,j+\frac{1}{2}}^r}{\sqrt{h_{i,j+\frac{1}{2}}^l} + \sqrt{h_{i,j+\frac{1}{2}}^r}}, \quad \tilde{v}_{i,j+\frac{1}{2}} := \frac{\sqrt{h_{i,j+\frac{1}{2}}^l} v_{i,j+\frac{1}{2}}^l + \sqrt{h_{i,j+\frac{1}{2}}^r} v_{i,j+\frac{1}{2}}^r}{\sqrt{h_{i,j+\frac{1}{2}}^l} + \sqrt{h_{i,j+\frac{1}{2}}^r}}, \quad (4.46)$$

$$\tilde{h}_{i,j+\frac{1}{2}} := \frac{h_{i,j+\frac{1}{2}}^l + h_{i,j+\frac{1}{2}}^r}{2}, \quad \tilde{c}_{i,j+\frac{1}{2}} := \sqrt{g \tilde{h}_{i,j+\frac{1}{2}}}.$$

4.3.3 Source term discretization

The discretization method of the one-dimensional case can be generalised to the two-dimensional case as follows:

$$S_{i,j_2} := \frac{1}{\Delta x \Delta y} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} -gh(x, y, t) b_x(x, y) \, dx \, dy = -\frac{g(b_{i+1,j} - b_{i-1,j})}{2\Delta x} \left(h_{i-\frac{1}{2},j}^r + h_{i+\frac{1}{2},j}^l \right), \quad (4.47a)$$

$$S_{i,j_3} := \frac{1}{\Delta x \Delta y} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} -gh(x, y, t) b_y(x, y) \, dx \, dy = -\frac{g(b_{i,j+1} - b_{i,j-1})}{2\Delta y} \left(h_{i,j-\frac{1}{2}}^r + h_{i,j+\frac{1}{2}}^l \right), \quad (4.47b)$$

where $b_{i,j} := b(x_i, y_j)$. Note that the discretization above is second order accurate in space, which is in accordance with the second-order limited scheme.

4.3.4 Time integration

We can generalise the time integration for the one-dimensional case to the two-dimensional case by time stepping as follows:

$$\vec{q}_{i,j}^{n+1} = \vec{q}_{i,j}^n + \frac{1}{6} \left(\vec{k}_1 + \vec{k}_2 + 4\vec{k}_3 \right), \quad (4.48)$$

where we define the following function:

$$\vec{K}(\vec{q}_{i,j}) = \Delta t \left(-\frac{1}{\Delta x} \left(\vec{F}_{i+\frac{1}{2},j} - \vec{F}_{i-\frac{1}{2},j} \right) - \frac{1}{\Delta y} \left(\vec{G}_{i,j+\frac{1}{2}} - \vec{G}_{i,j-\frac{1}{2}} \right) + \vec{S}_{i,j} \right), \quad (4.49)$$

where we calculate

$$\vec{k}_1 = \vec{K}(\vec{q}_{i,j}), \quad (4.50a)$$

$$\vec{k}_2 = \vec{K}(\vec{q}_{i,j} + \vec{k}_1), \quad (4.50b)$$

$$\vec{k}_3 = \vec{K}(\vec{q}_{i,j} + \frac{1}{4}\vec{k}_1 + \frac{1}{4}\vec{k}_2). \quad (4.50c)$$

Similar to the 1D case, to ensure that waves do not intersect we require that

$$\Delta t \max_{i,j} (|u \pm \sqrt{gh}|, |v \pm \sqrt{gh}|) < \frac{\min(\Delta x, \Delta y)}{2}, \quad (4.51)$$

and thus we require for the time step

$$\Delta t < c \cdot \frac{\min(\Delta x, \Delta y)}{2 \max_{i,j} (|u \pm \sqrt{gh}|, |v \pm \sqrt{gh}|)}, \quad (4.52)$$

where $0 < c < 1$.

5 Numerical results

5.1 Test case 1: A water hill on a flat bottom

The first test case is one-dimensional. We consider a water hill on a flat bottom, i.e., for the bottom we take $b(x) \equiv 0$. This test problem is very similar to the first test case that Sweby [18] considered. We study this case because of its simplicity but most importantly to investigate how well each state interpolation scheme performs on a smooth initial solution.

We consider a computational domain $[x_L, x_R] = [-L, L]$ with time interval $[0, T]$ where we assume that at $t = T$ no waves are close to both of the non-periodic boundaries $x = -L$ and $x = L$. In our case we take $L = 8$ and $T = 3$. As initial conditions we set $u \equiv 0$ throughout the domain and for the free surface we take

$$h(x, 0) = b(x) + h(x, 0) = s(x, 0) = 1 + e^{-x^2}. \quad (5.1)$$

To solve the 1D shallow water equations, we employ each state interpolation method discussed in Section 3.2.3. To be more precise, we use the first-order upwind scheme, the second-order upwind scheme with $\kappa = \frac{1}{3}$ and the second-order limited scheme with the Superbee, Minmod and Koren limiters as given by (3.22)-(3.24). For the computations of each state interpolation method we use a mesh of 100 cells and employ the HLLE solver as flux evaluation method for no other reason than its computational speed. The methods are compared to a high resolution solution with 10000 cells which is computed using the Koren limiter and the HLLE solver. We use a c coefficient 0.9 for each computation.

Figures 5.1 to 5.6 show the computed free surface and velocity at $t = 1$, where the black solid line indicates the high resolution solution. Similarly, Figures 5.7 to 5.12 show the computed free surface and velocity at $t = 2$ and Figures 5.13 to 5.18 show the computed free surface and velocity at $t = 3$.

Figures 5.1 to 5.6 clearly show how the first-order upwind method under performs compared to the other used methods. This trend continues for $t = 2$ and $t = 3$ in Figures 5.7 to 5.18 and illustrates the usefulness of second-order accurate methods. When inspecting the $\kappa = \frac{1}{3}$ scheme, it seems to perform well at $t = 1$ as seen in Figures 5.1 to 5.6. But when the leftward propagating wave becomes steeper at $t = 2$, and especially at $t = 3$, it overestimates the crest of the wave and underestimates the trough of the wave in a seemingly unstable manner as is best illustrated in both Figures 5.15 and 5.18. It is noteworthy that the three limited schemes seem more stable and do not overestimate the crest of the wave and underestimate the trough of the wave, which is why they are likely the best option to go for in this scenario.

Upon closer inspection of the three limited schemes, the Minmod limiter seems to perform the worst of these three. When inspecting for example the crest of the leftward propagating wave of the free surface at $t = 1$, $t = 2$ and $t = 3$ it consistently underestimates the free surface height more than the Superbee and Koren limiters as can be seen in for example Figures 5.3, 5.9 and 5.15. Furthermore, when the wave becomes steeper at $t = 2$ and $t = 3$, the Mindmod limiters overestimates the trough of the leftward propagating wave of the free surface and its computed solution lies further from the computed high resolution solution than both the Superbee and Koren limiters as can be seen in Figures 5.11 and 5.17. Finally, the Superbee limiter seems to perform equally well as the Koren limiter at free surface and velocity of the leftward propagating wave at $t = 1$, $t = 2$ and $t = 3$.

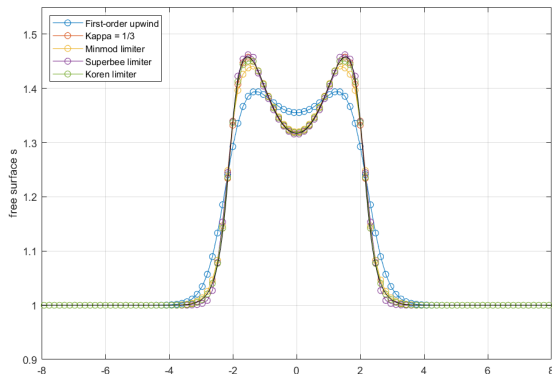


Figure 5.1: One-dimensional water hill on a flat bottom. View of the computed free surface at $t = 1$.

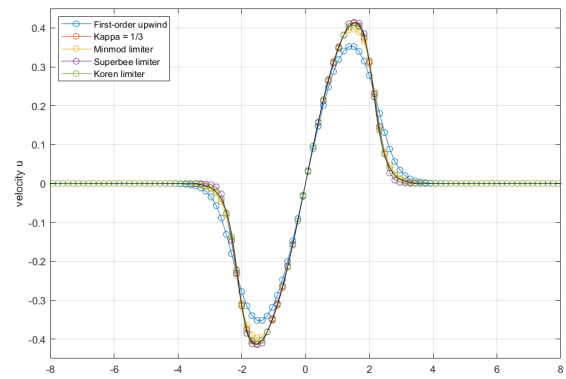


Figure 5.2: One-dimensional water hill on a flat bottom. View of the computed velocity at $t = 1$.

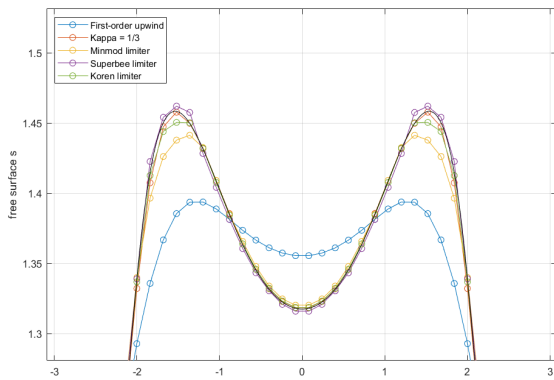


Figure 5.3: One-dimensional water hill on a flat bottom. View of the crest of both the leftward and rightward propagating waves of the computed free surface at $t = 1$.

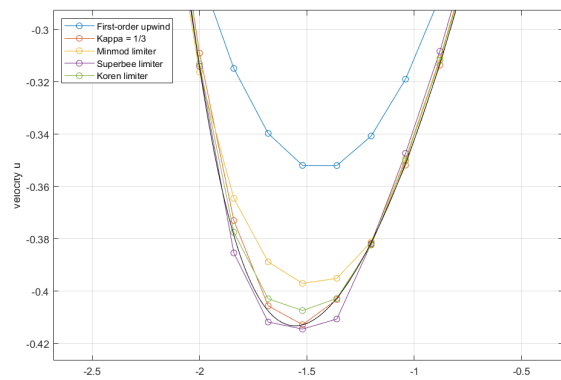


Figure 5.4: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 1$.

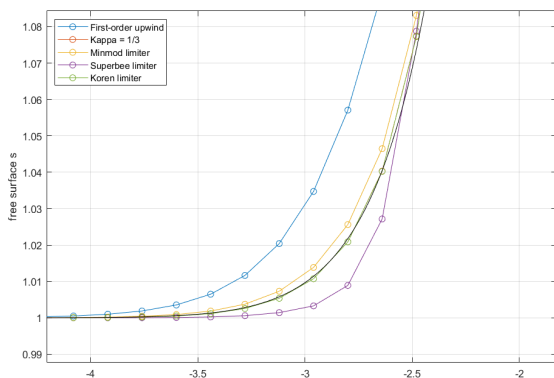


Figure 5.5: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 1$.

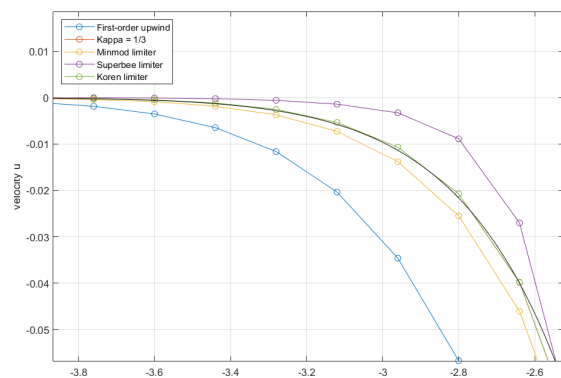


Figure 5.6: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 1$.

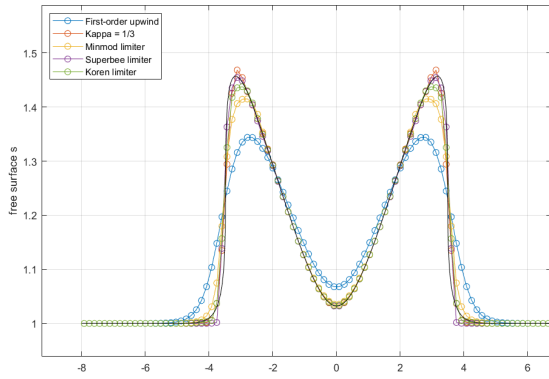


Figure 5.7: One-dimensional water hill on a flat bottom. View of the computed free surface at $t = 2$.

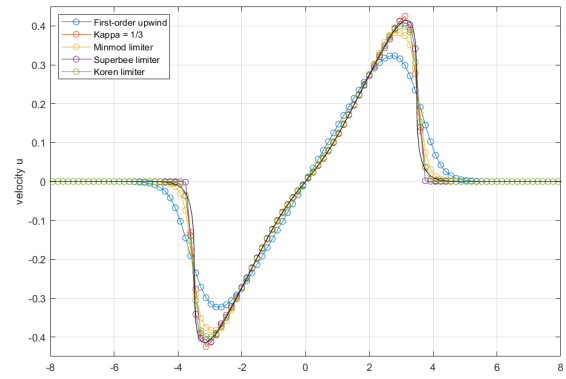


Figure 5.8: One-dimensional water hill on a flat bottom. View of the computed velocity at $t = 2$.

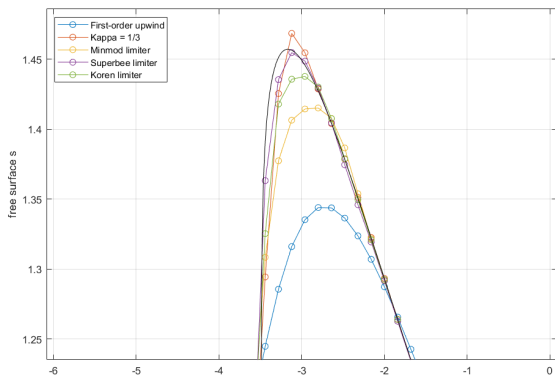


Figure 5.9: One-dimensional water hill on a flat bottom. View of the crest the leftward propagating wave of the computed free surface at $t = 2$.

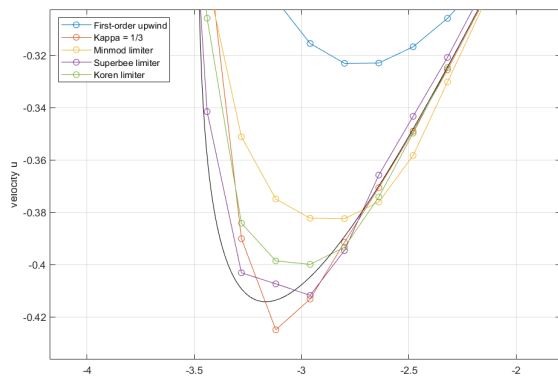


Figure 5.10: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 2$.

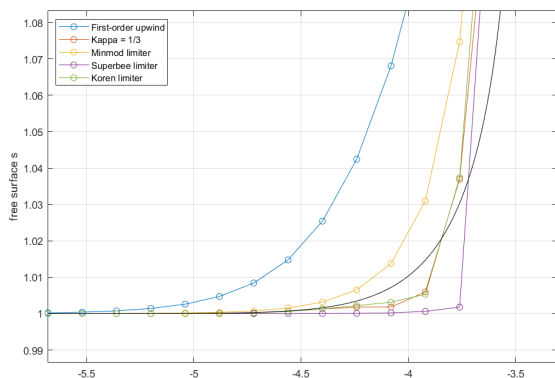


Figure 5.11: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 2$.

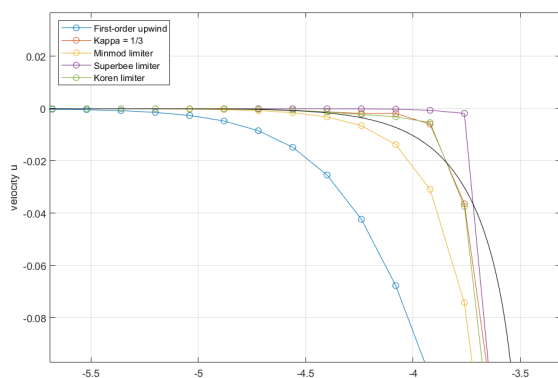


Figure 5.12: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 2$.

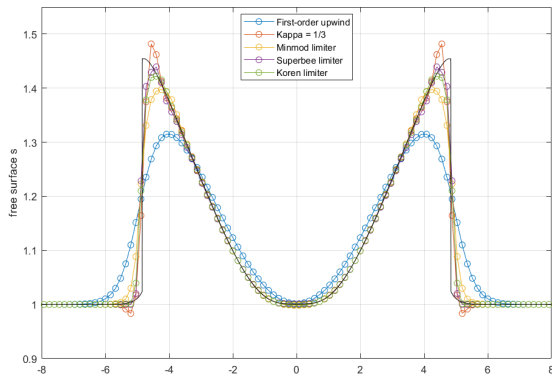


Figure 5.13: One-dimensional water hill on a flat bottom. View of the computed free surface at $t = 3$.

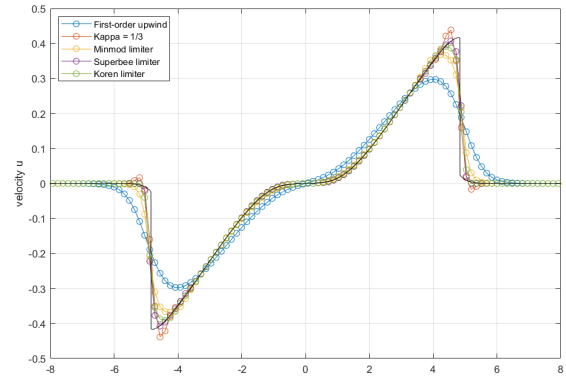


Figure 5.14: One-dimensional water hill on a flat bottom. View of the computed velocity at $t = 3$.

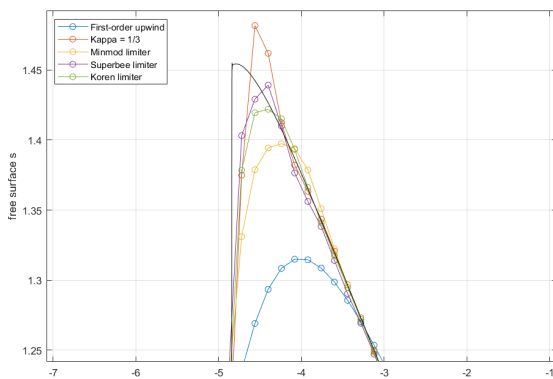


Figure 5.15: One-dimensional water hill on a flat bottom. View of the crest the leftward propagating wave of the computed free surface at $t = 3$.

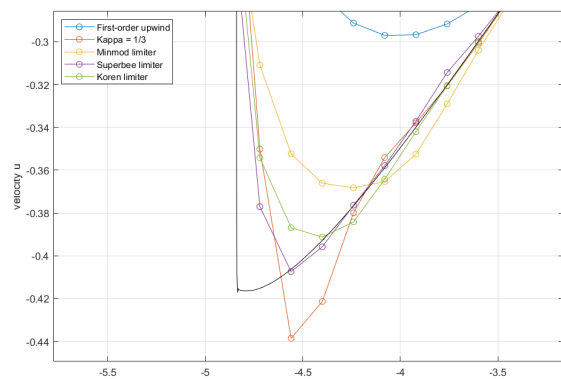


Figure 5.16: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 3$.

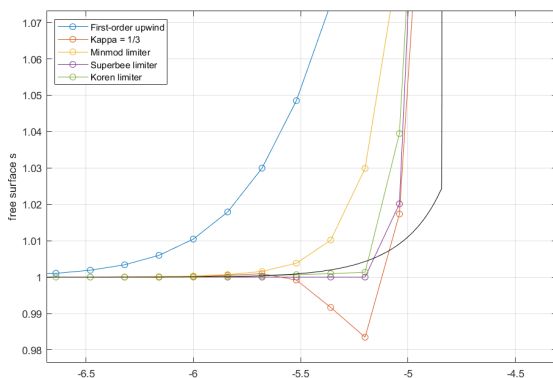


Figure 5.17: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 3$.

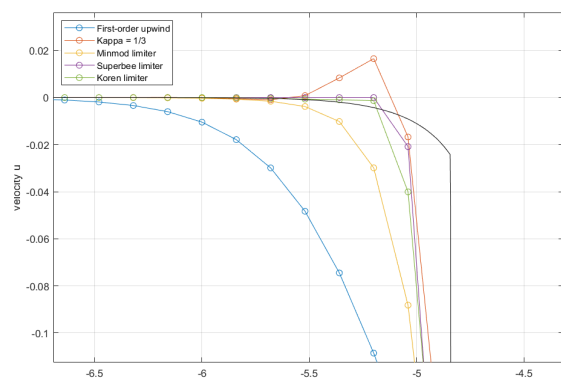


Figure 5.18: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 3$.

In addition to the previous experiment, we investigate each flux evaluation method discussed in Section 3.2.4. To be more precise, we use the Flux vector splitting scheme, the Flux difference splitting scheme, with the midpoint rule, the trapezoidal rule and Roe's method, the HLL solver and the HLLE solver. For the computations of each flux evaluation method we again use a mesh of 100 cells and employ the second-order limited scheme with the both the Superbee and Koren limiters since they performed best in our previous experiment. On this occasion, the methods are compared to a high resolution solution with 20000 cells which is also computed using either the Superbee or Koren limiter and the HLLE solver is used for its computational speed. Similar to before, we use a c coefficient 0.9 for each computation.

Figures 5.19 to 5.24 show the computed free surface and velocity using the Superbee limiter at $t = 3$, where the solid line indicates the high resolution solution calculated using the computationally fast HLLE solver. Similarly, Figures 5.25 to 5.29 show the computed free surface and velocity using the Koren limiter at $t = 3$, where the solid line indicates the high resolution solution. The reader should note that error of the high resolution solution at for example the tip of the crest in Figure 5.23 is inherent to the numerical methods used, for example halving the C constant does not prevent this from happening.

Upon close inspection of the leftward propagating wave in the case of the Koren limiter, as can be seen in Figures 5.27 to 5.30, the flux vector splitting and the trapezoidal rule methods perform worse at estimating the free surface and velocity than the other flux evaluation methods. The same occurs in the case of the Superbee limiter when estimating the free surface, as can be seen in Figures 5.21 and 5.23. When inspecting the computed velocity in the case of the Superbee limiter, the midpoint rule and Roe's method perform worse at estimating the velocity than the other flux evaluation methods, as can be see in Figures 5.22 and 5.24. The other remaining flux evaluation methods, the HLL solver and the HLLE solver, yield similar numerical solutions for both the Superbee and Koren limiters.

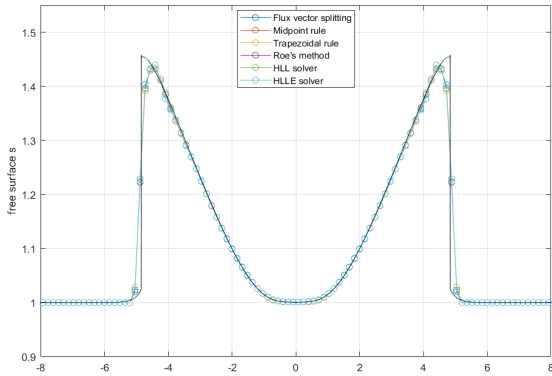


Figure 5.19: One-dimensional water hill on a flat bottom. View of the computed free surface at $t = 3$ computed using the Superbee limiter.

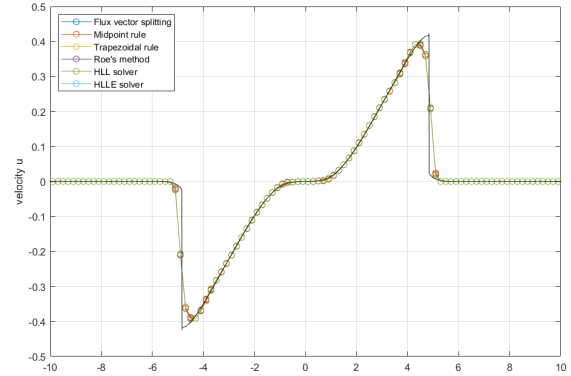


Figure 5.20: One-dimensional water hill on a flat bottom. View of the computed velocity at $t = 3$ computed using the Superbee limiter.

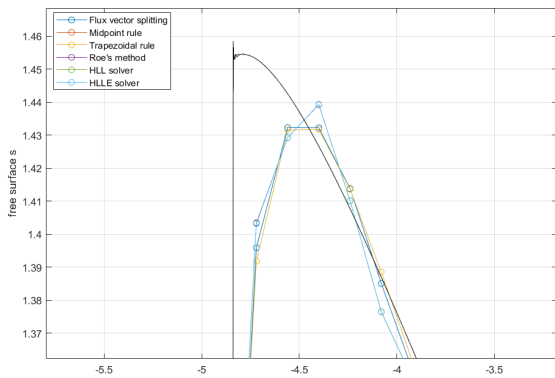


Figure 5.21: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Superbee limiter.

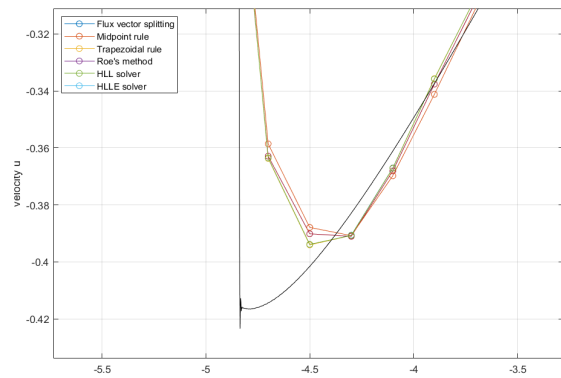


Figure 5.22: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Superbee limiter.

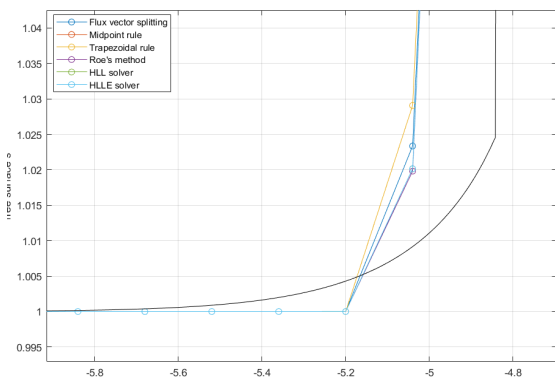


Figure 5.23: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Superbee limiter.

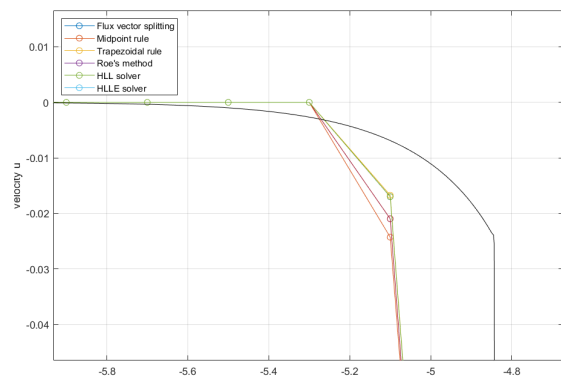


Figure 5.24: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Superbee limiter.

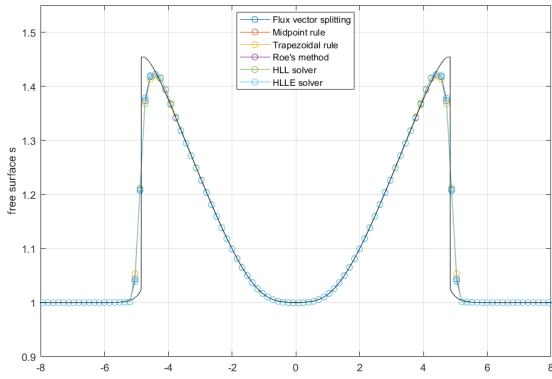


Figure 5.25: One-dimensional water hill on a flat bottom. View of the computed free surface at $t = 3$ computed using the Koren limiter.

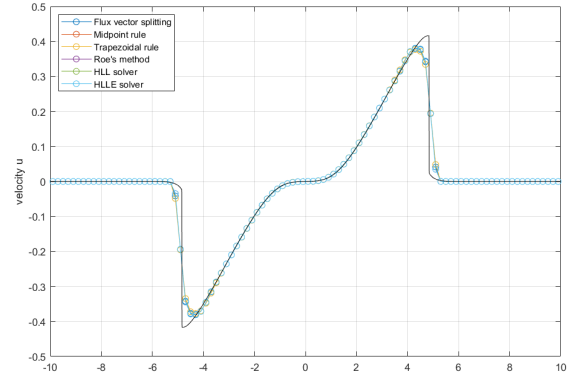


Figure 5.26: One-dimensional water hill on a flat bottom. View of the computed velocity at $t = 3$ computed using the Koren limiter.

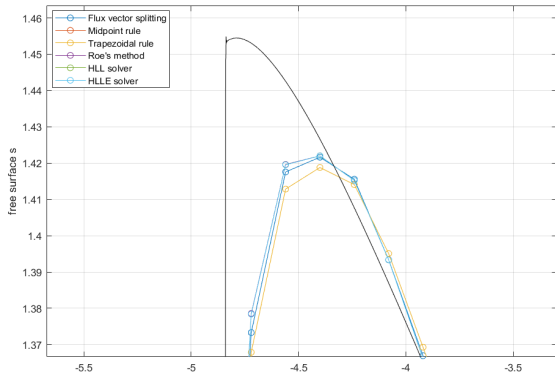


Figure 5.27: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Koren limiter.

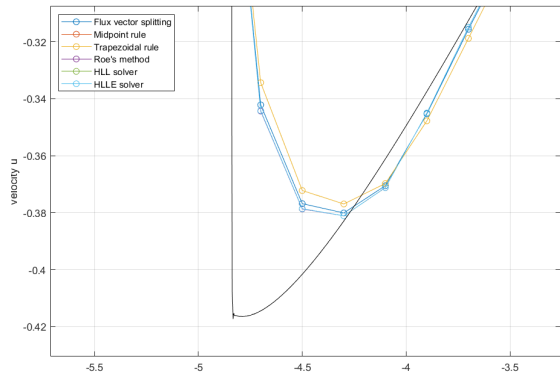


Figure 5.28: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Koren limiter.

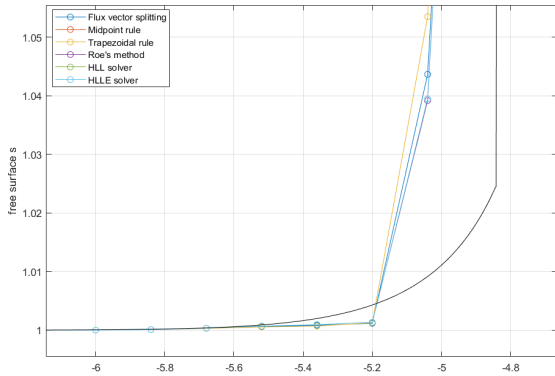


Figure 5.29: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Koren limiter.

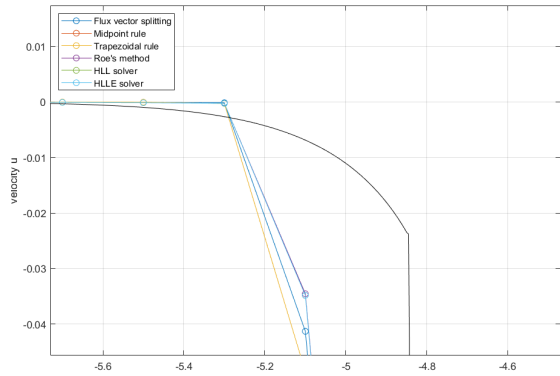


Figure 5.30: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Koren limiter.

We expect the numerical solution to become more accurate when the number of cells is increased, i.e., we expect our numerical methods to be consistent. Using this test case we can investigate this and even determine the order of convergence in space. For each flux evaluation method we determine the order of convergence numerically, with the second-order limited scheme as state interpolation method, by doubling the number of cells and halving the time step size.

To measure the order of convergence, we consider the total energy in the system which is the sum of the kinetic energy and the potential energy in the system. Suppose we have N cells, let $q_{1,i}$ denote the numerical solution of q_1 in cell i ($i = 1, \dots, N$) at some point in time and similarly let $q_{2,i}$ denote the numerical solution of q_2 in cell i . Then the total energy in the system can be calculated by integrating the formula $E = \frac{1}{2}u^2 + \frac{1}{2}gh$ over the domain. In our case we calculate this numerically using the following:

$$E_N = \frac{1}{N} \sum_{i=1}^N \frac{1}{2} \left(\frac{q_{2,i}}{q_{1,i}} \right)^2 + \frac{1}{2} g q_{1,i}. \tag{5.2}$$

Since we expect our numerical methods to be consistent, we expect this numerical energy at some point in time to converge for each flux evaluation method. Prior to running our tests, we determine the largest wave speed during the simulation from a high resolution solution computed using the Superbee limiter and HLLC solver. We then fix this wave speed, and thus the time step size, during the simulation instead of varying the time step size each time. As a result the time step size is halved when the number of cells is doubled. Using Richardson extrapolation [22], we estimate the spatial order of convergence as follows. Table 5.1 shows the results of the tests at $t = 1$, from which we can conclude that each flux evaluation method is at least second order convergent in space.

Flux evaluation	Cells N	Energy E_N (%)	Order of convergence
Flux vector splitting	100	0.556309	-
	200	0.556518	-
	400	0.556574	1.907
	800	0.556584	2.441
	1600	0.556587	1.922
Midpoint rule	100	0.556328	-
	200	0.556520	-
	400	0.556574	1.845
	800	0.556584	2.432
	1600	0.556587	1.914
Trapezoidal rule	100	0.556284	-
	200	0.556512	-
	400	0.556572	1.934
	800	0.556583	2.385
	1600	0.556586	1.914
Roe's method	100	0.556328	-
	200	0.556520	-
	400	0.556574	1.845
	800	0.556584	2.432
	1600	0.556587	1.914
HLL solver	100	0.556328	-
	200	0.556520	-
	400	0.556574	1.847
	800	0.556584	2.432
	1600	0.556587	1.914
HLLC solver	100	0.556328	-
	200	0.556520	-
	400	0.556574	1.847
	800	0.556584	2.432
	1600	0.556587	1.914

Table 5.1: One-dimensional water hill on a flat bottom. Comparison of the numerical energy of each flux evaluation method at $t = 1$ and the numerical order of convergence.

5.2 Test case 2: The dam-break problem

The dam-break problem is the most commonly investigated problem for the shallow water equations and often discussed in literature [11, 17, 20, 21]. Dam break simulation is important for societal reasons, because dam breaks can release an enormous amount of water in a short time which threatens both life and infrastructure. But dam-breaks have also become the standard test scenario for numerical methods for shallow water equations. Toro [20] has the dam-break problem in high regard, because its solution includes both continuous and discontinuous solutions at the same time. It is noteworthy that the dam-break problem does not satisfy the assumptions made in deriving the shallow water equations, hence it does not seem relevant to consider it here. However, several researches have shown that the shallow water equations are suitable for the representation of dam-break flows [2, 12] and thus we consider it here.

This test case is also one-dimensional. We consider the classical dam-break problem [17] on a flat bottom, i.e., for the bottom we take $b(x) \equiv 0$. As initial conditions we set $u \equiv 0$ throughout the domain and for the free surface we take

$$h(x, 0) = \begin{cases} h_L & \text{if } -L \leq x \leq 0, \\ h_R & \text{if } 0 < x \leq L. \end{cases} \quad (5.3)$$

We consider a computational domain $[x_L, x_R] = [-L, L]$ with time interval $[0, T]$ where again we assume that at $t = T$ no waves are close to any of the non-periodic boundaries $x = -L$ and $x = L$ and hence no discretization of the boundaries is required. We take $L = 8$ and $T = 3$. To solve the 1D shallow water equations, we again employ each state interpolation method discussed in Section 3.2.3 as in the previous test case. Similarly, for the computations of each state interpolation method we use a mesh of 100 cells and employ the HLLE solver as flux evaluation method for no other reason than its computational speed which we compare to the exact solution [3]. We use a c coefficient 0.9 for each computation.

Figures 5.31 to 5.36 show the computed free surface and velocity at $t = 1$, where the solid line indicates the exact solution. Similarly, Figures 5.37 to 5.42 show the computed free surface and velocity at $t = 2$ and Figures 5.43 to 5.48 show the computed free surface and velocity at $t = 3$. The non-physical extrema for the Superbee and Koren limiters in for example Figure 5.33 are inherent to these methods and can not be avoided by simply halving the c coefficient. However, this unwanted phenomenon disappears at $t = 2$ and $t = 3$.

Similar to the previous text case, Figures 5.31 to 5.36 clearly show how the first-order upwind method underperforms compared to the other used methods, which continues for $t = 2$ and $t = 3$ in Figures 5.37 to 5.48. When inspecting the $\kappa = \frac{1}{3}$ scheme, it does not seem stable as can be seen in Figures 5.34 and 5.40. Furthermore, it underestimates or overestimates the free surface height and velocity, which the other methods do not suffer from.

Upon closer inspection of the three limited schemes, the Minmod limiter seems to again perform the worst of these three. When inspecting for example the crest of the leftward propagating wave of the free surface at $t = 1$, $t = 2$ and $t = 3$ it consistently underestimates the free surface height more than the Superbee and Koren limiters as can be seen in for example Figures 5.35, 5.41 and 5.47. Furthermore, the Superbee limiter slightly outperforms the Koren limiter in both estimating the crest and trough of the wave at $t = 1$, $t = 2$ and $t = 3$ as can be seen in for example Figures 5.39 and 5.41.

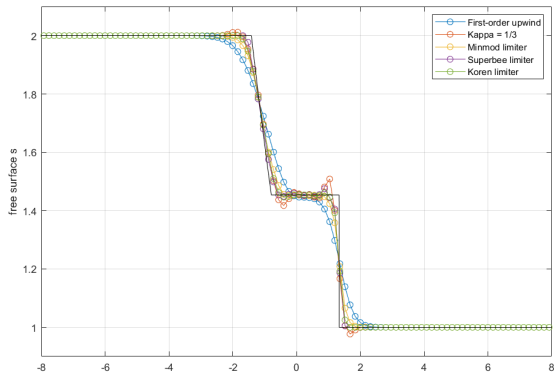


Figure 5.31: One-dimensional dam-break problem on a flat bottom. View of the computed free surface at $t = 1$.

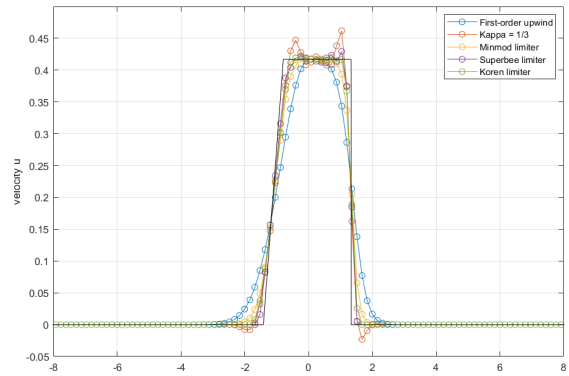


Figure 5.32: One-dimensional dam-break problem on a flat bottom. View of the computed velocity at $t = 1$.

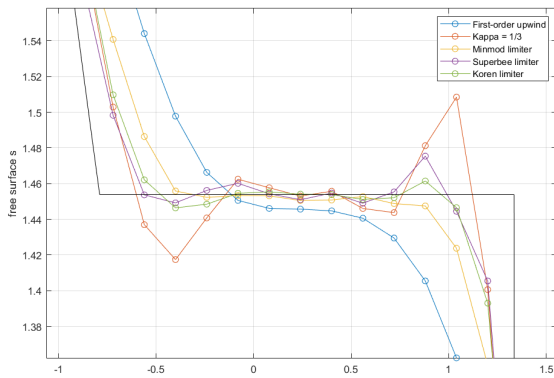


Figure 5.33: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave and the crest of the rightward propagating wave of the computed free surface at $t = 1$.

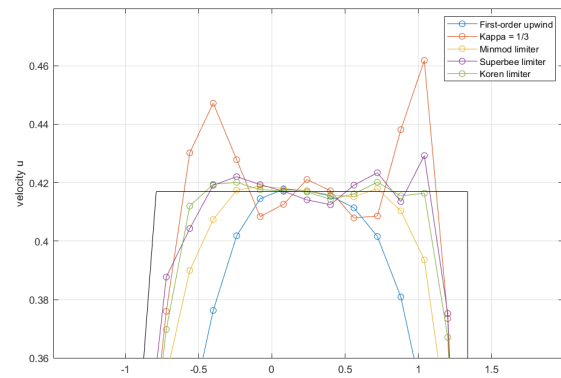


Figure 5.34: One-dimensional dam-break problem on a flat bottom. View of the crest of both the leftward and rightward propagating waves of the computed velocity at $t = 1$.

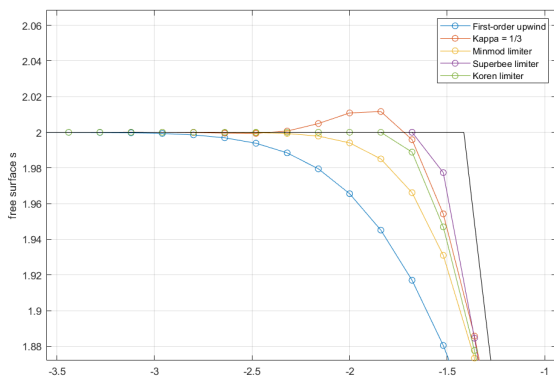


Figure 5.35: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 1$.

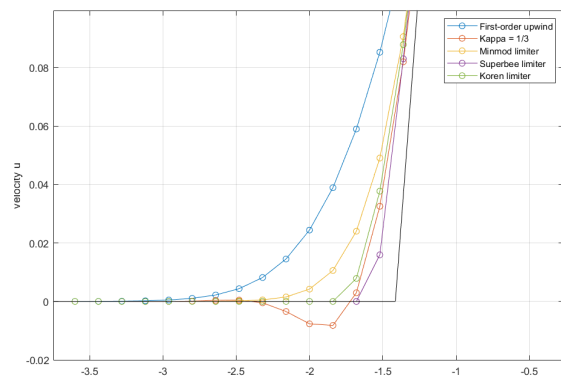


Figure 5.36: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 1$.

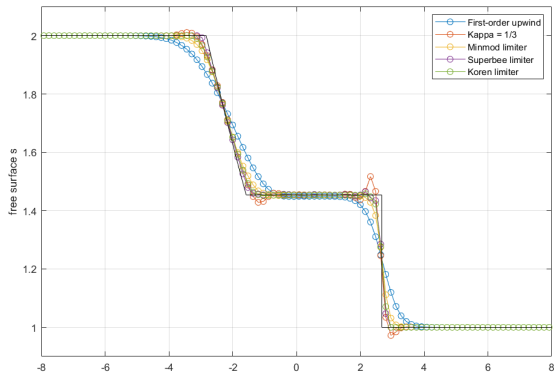


Figure 5.37: One-dimensional dam-break problem on a flat bottom. View of the computed free surface at $t = 2$.

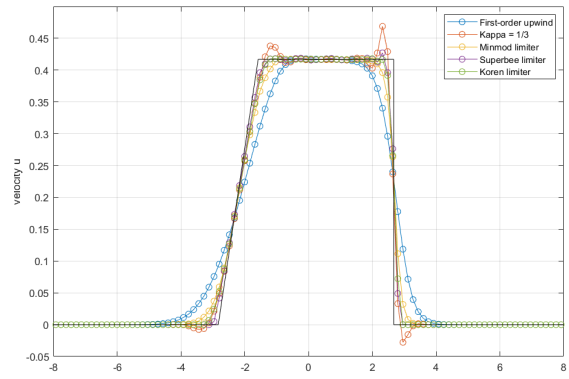


Figure 5.38: One-dimensional dam-break problem on a flat bottom. View of the computed velocity at $t = 2$.

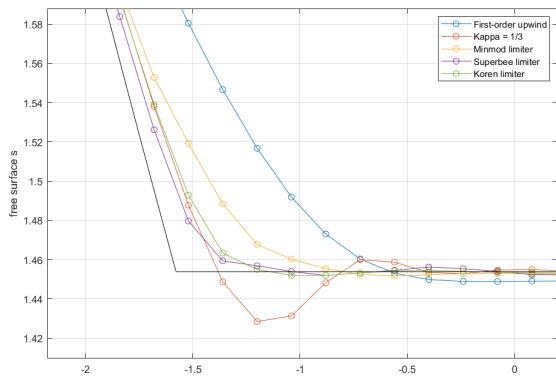


Figure 5.39: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 2$.

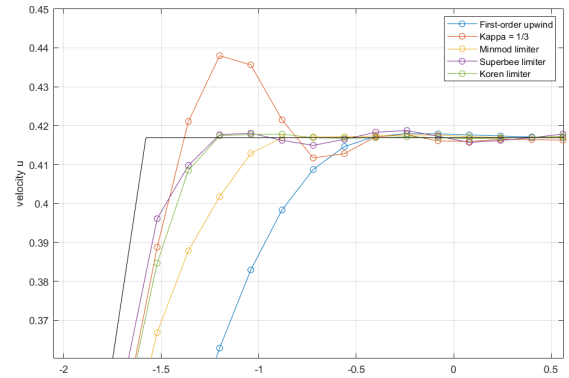


Figure 5.40: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 2$.

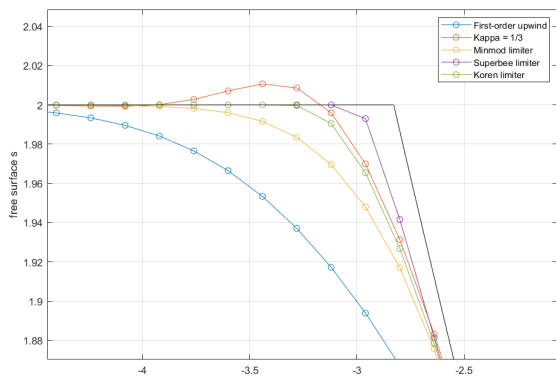


Figure 5.41: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 2$.

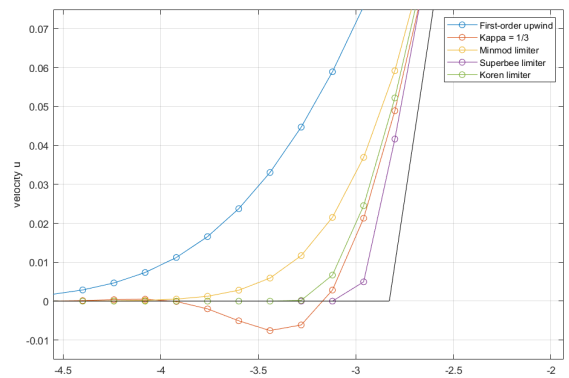


Figure 5.42: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 2$.

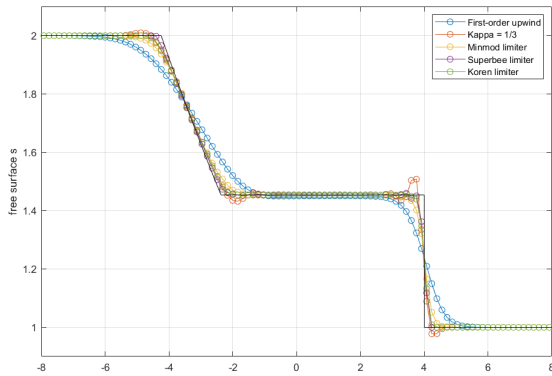


Figure 5.43: One-dimensional dam-break problem on a flat bottom. View of the computed free surface at $t = 3$.

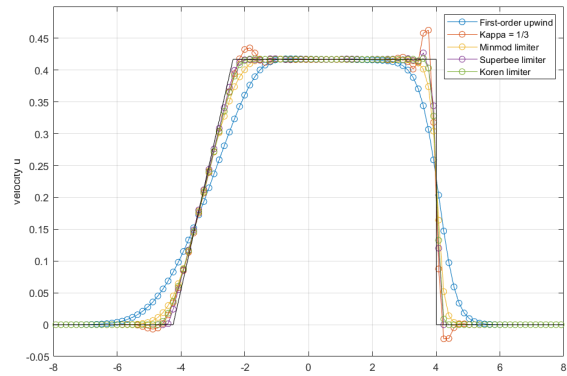


Figure 5.44: One-dimensional dam-break problem on a flat bottom. View of the computed velocity at $t = 3$.

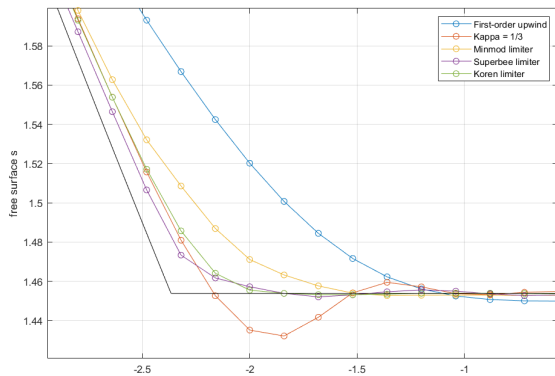


Figure 5.45: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 3$.

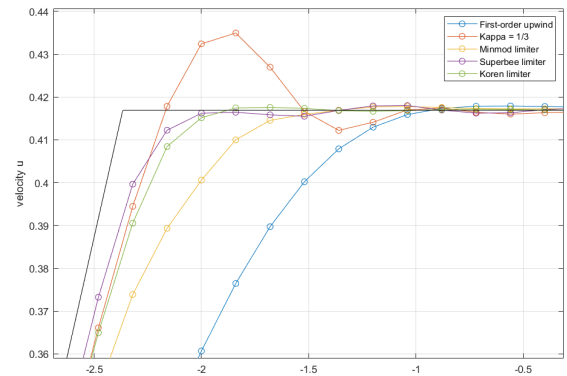


Figure 5.46: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 3$.

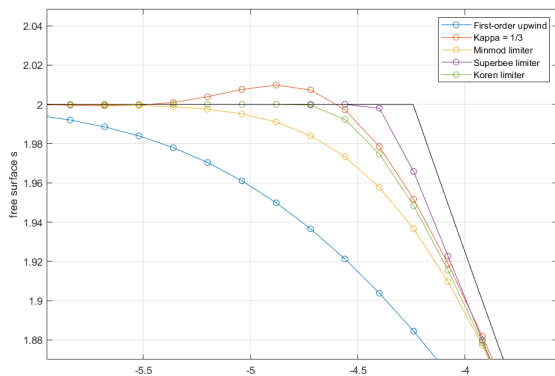


Figure 5.47: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 3$.

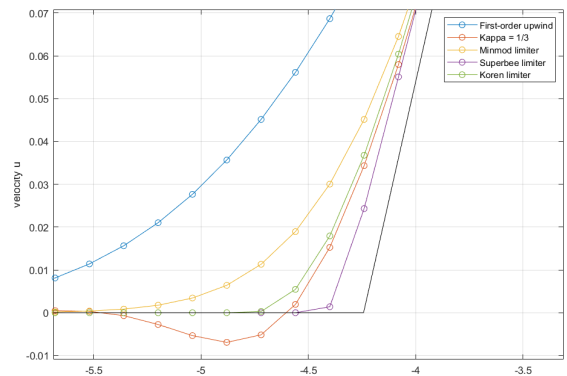


Figure 5.48: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 3$.

In addition to investigating the simulation visually, it makes sense to use the fact that an exact solution is available for this test case and compare numerical errors. Suppose we have N cells, let $q_{1,i}$ denote the numerical solution of q_1 in cell i ($i = 1, \dots, N$) at some point in time and similarly let $q_{2,i}$ denote the numerical solution of q_2 in cell i . Furthermore, let h_i and u_i denote the exact solutions at the cell centres x_i ($i = 1, \dots, N$). We consider the L_1 , L_2 and L_∞ errors of the computed free surface and velocity, given by

$$L_1(h) = \frac{1}{N} \sum_{i=1}^N |q_{1,i} - h_i| \quad L_1(u) = \frac{1}{N} \sum_{i=1}^N \left| \frac{q_{2,i}}{q_{1,i}} - u_i \right| \quad (5.4a)$$

$$L_2(h) = \frac{1}{N} \sqrt{\sum_{i=1}^N (q_{1,i} - h_i)^2} \quad L_2(u) = \frac{1}{N} \sqrt{\sum_{i=1}^N \left(\frac{q_{2,i}}{q_{1,i}} - u_i \right)^2} \quad (5.4b)$$

$$L_\infty(h) = \max_{i=1, \dots, N} |q_{1,i} - h_i| \quad L_\infty(u) = \max_{i=1, \dots, N} \left| \frac{q_{2,i}}{q_{1,i}} - u_i \right| \quad (5.4c)$$

Table 5.2 shows the L_1 , L_2 and L_∞ errors of the computed free surface and velocity at $t = 1$. Similarly, Table 5.3 shows the L_1 , L_2 and L_∞ errors of the computed free surface and velocity at $t = 2$ and Table 5.4 shows the L_1 , L_2 and L_∞ errors of the computed free surface and velocity at $t = 3$. As expected, the Superbee limiter has a smaller error in most cases than the other state interpolation methods.

	First-order upwind	Kappa = 1/3	Minmod limiter	Superbee limiter	Koren limiter
$L_1(h)$	0.0182	0.0074	0.0089	0.0051	0.0063
$L_2(h)$	0.0045	0.0023	0.0029	0.0022	0.0024
$L_\infty(h)$	0.2181	0.1668	0.2085	0.1864	0.1925
$L_1(u)$	0.0151	0.0063	0.0075	0.0045	0.0052
$L_2(u)$	0.0039	0.0020	0.0026	0.0020	0.0022
$L_\infty(u)$	0.2134	0.1623	0.2052	0.1848	0.1891

Table 5.2: One-dimensional dam-break problem on a flat bottom. Comparison of the L_1 , L_2 and L_∞ errors of the computed free surface and velocity of each state interpolation method at $t = 1$.

	First-order upwind	Kappa = 1/3	Minmod limiter	Superbee limiter	Koren limiter
$L_1(h)$	0.0237	0.0080	0.0095	0.0049	0.0061
$L_2(h)$	0.0049	0.0025	0.0027	0.0019	0.0022
$L_\infty(h)$	0.2053	0.2081	0.1795	0.1684	0.1772
$L_1(u)$	0.0193	0.0066	0.0078	0.0041	0.0050
$L_2(u)$	0.0042	0.0021	0.0023	0.0016	0.0019
$L_\infty(u)$	0.1777	0.1801	0.1533	0.1405	0.1511

Table 5.3: One-dimensional dam-break problem on a flat bottom. Comparison of the L_1 , L_2 and L_∞ errors of the computed free surface and velocity of each state interpolation method at $t = 2$.

	First-order upwind	Kappa = 1/3	Minmod limiter	Superbee limiter	Koren limiter
$L_1(h)$	0.0274	0.0078	0.0097	0.0046	0.0059
$L_2(h)$	0.0051	0.0019	0.0026	0.0016	0.0020
$L_\infty(h)$	0.2096	0.1186	0.1659	0.1180	0.1326
$L_1(u)$	0.0223	0.0065	0.0081	0.0040	0.0049
$L_2(u)$	0.0044	0.0017	0.0024	0.0015	0.0018
$L_\infty(u)$	0.2036	0.0991	0.1642	0.1201	0.1326

Table 5.4: One-dimensional dam-break problem on a flat bottom. Comparison of the L_1 , L_2 and L_∞ errors of the computed free surface and velocity of each state interpolation method at $t = 3$.

In addition to the previous experiment, we investigate each flux evaluation method similar to the previous test case. For the computations of each flux evaluation method we again use a mesh of 100 cells and employ the second-order limited scheme with both the Superbee and Koren limiters since they performed best in our previous experiment. We again compare them to the exact solution [3]. Finally, we use a c coefficient 0.9 for each computation.

Figures 5.49 to 5.54 show the computed free surface and velocity using the Superbee limiter at $t = 3$, where the solid line indicates the exact solution. Similarly, Figures 5.55 to 5.60 show the computed free surface and velocity using the Koren limiter at $t = 3$, where the solid line indicates the exact solution.

Upon close inspection of the leftward propagating wave in the case of the Superbee limiter, as can be seen in for example Figures 5.51 to 5.52, the flux vector splitting and the trapezoidal rule methods perform worse at estimating the free surface and velocity than the other flux evaluation methods. The same occurs in the case of the Koren limiter when estimating the free surface, as can be seen in for example Figures 5.59 and 5.60. The other remaining flux evaluation methods yield similar numerical solutions for both the Superbee and Koren limiters.

Table 5.5 shows the L_1 , L_2 and L_∞ errors of the computed free surface and velocity at $t = 3$ of each flux evaluation method using the Superbee limiter. As expected, the flux vector splitting and the trapezoidal rule methods perform worse at estimating the free surface and velocity than the other flux evaluation methods. The remaining four flux evaluation methods only differ very slightly in error.

	Flux vector splitting	Midpoint rule	Trapezoidal rule	Roe's method	HLL solver	HLLC solver
$L_1(h)$	0.0050438	0.0045832	0.0049796	0.0046019	0.0046256	0.0046197
$L_2(h)$	0.0017182	0.0016280	0.0018039	0.0016338	0.0016311	0.0016331
$L_\infty(h)$	0.1236692	0.1180718	0.1311247	0.1185618	0.1177128	0.1180421
$L_1(u)$	0.0041291	0.0039502	0.0043122	0.0039694	0.0039780	0.0039630
$L_2(u)$	0.0016078	0.0014959	0.0016507	0.0015016	0.0014965	0.0014992
$L_\infty(u)$	0.1312072	0.1201477	0.1326269	0.1206649	0.1197547	0.1201183

Table 5.5: One-dimensional dam-break problem on a flat bottom. Comparison of the L_1 , L_2 and L_∞ errors of the computed free surface and velocity using the Superbee limiter at $t = 3$.

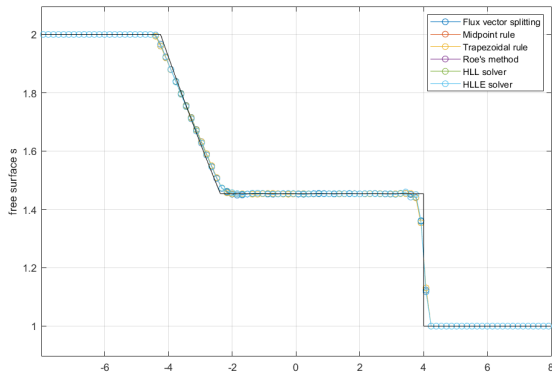


Figure 5.49: One-dimensional dam-break problem on a flat bottom. View of the computed free surface at $t = 3$ computed using the Superbee limiter.

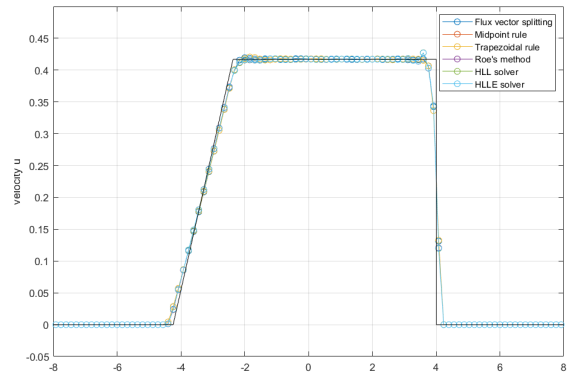


Figure 5.50: One-dimensional dam-break problem on a flat bottom. View of the computed velocity at $t = 3$ computed using the Superbee limiter.

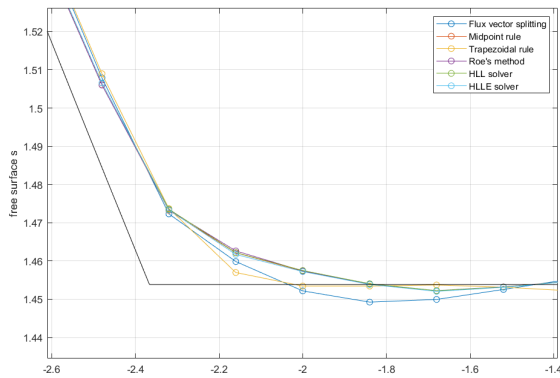


Figure 5.51: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Superbee limiter.

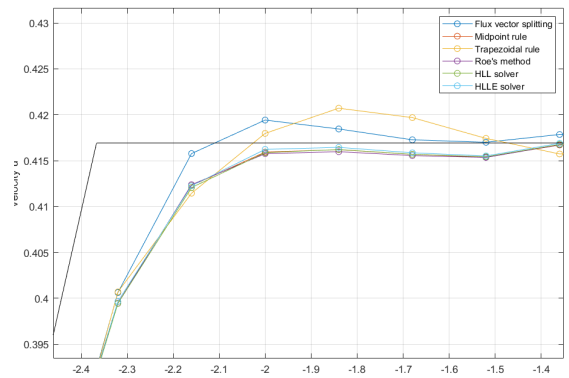


Figure 5.52: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Superbee limiter.

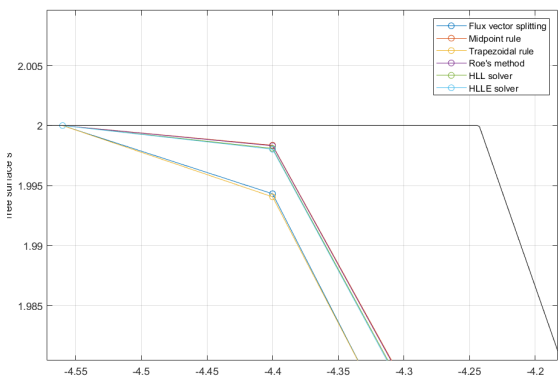


Figure 5.53: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Superbee limiter.

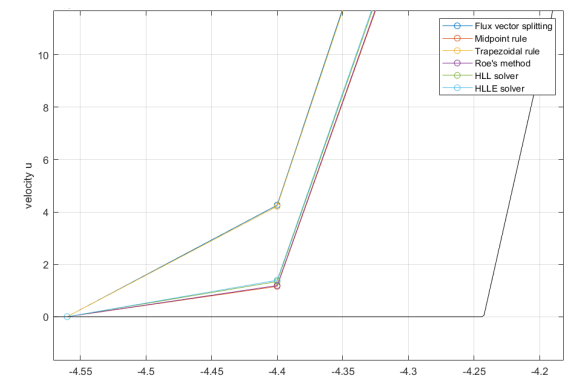


Figure 5.54: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Superbee limiter.

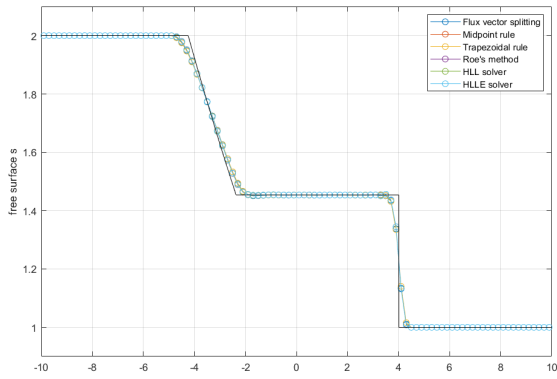


Figure 5.55: One-dimensional dam-break problem on a flat bottom. View of the computed free surface at $t = 3$ computed using the Koren limiter.

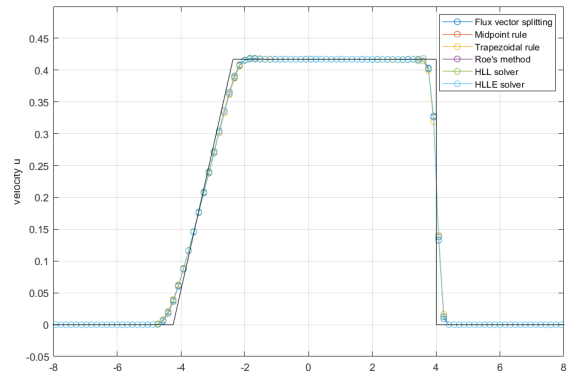


Figure 5.56: One-dimensional dam-break problem on a flat bottom. View of the computed velocity at $t = 3$ computed using the Koren limiter.

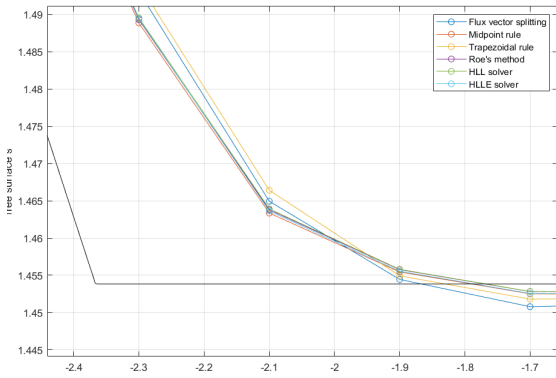


Figure 5.57: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Koren limiter.

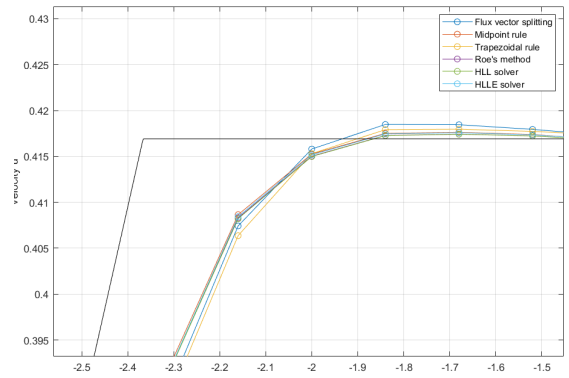


Figure 5.58: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Koren limiter.

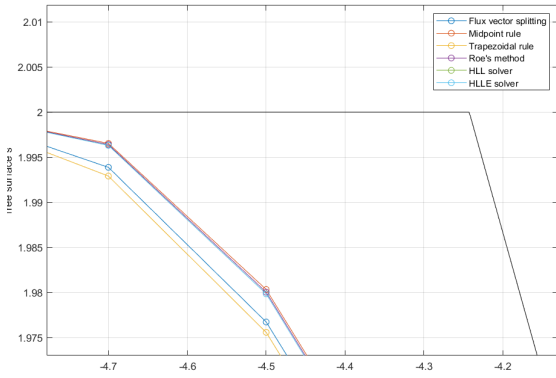


Figure 5.59: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Koren limiter.

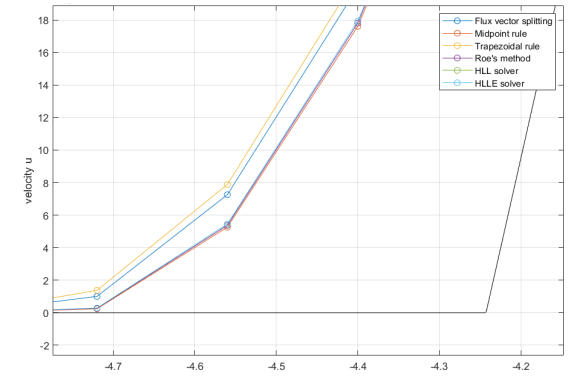


Figure 5.60: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Koren limiter.

5.3 Test case 3: Flow over a bump on the bottom

The third test case is two-dimensional. We consider water flowing over a bottom with a slight bump on it, which is given by $b(x, y) = \frac{1}{5}e^{-x^2-y^2}$. The physical phenomenon that should occur is known in literature as the Kelvin wedge [17] and it creates V-shaped waves. We study this case to investigate if the numerical methods are physically accurate as well, i.e., if the V-shaped waves are formed and if their angle is close to the expected angle.

We consider a computational domain $[-10, 10] \times [-10, 10]$ where the boundary at $x = x_L$ has inflow boundary conditions and the other three boundaries have outflow boundary conditions. As initial conditions we set $u \equiv 2$, i.e., there is supercritical flow in the x -direction, and $v \equiv 0$ throughout the domain and for the free surface we take

$$s(x, y, 0) = b(x, y) + h(x, y, 0) = 1. \tag{5.5}$$

Hence, this test case is flow over a bottom which is suddenly set in motion. To solve the 2D shallow water equations, we employ the Superbee and Koren limiters since they performed best in the 1D test cases. Furthermore, we employ each 2D flux evaluation method discussed in Section 4.3. We perform computations on a 100×100 grid and use a c coefficient 0.9 for each computation.

One can calculate the angle that the V-shaped waves make using the following formula:

$$\theta = \arcsin \frac{\sqrt{gh}}{u}, \tag{5.6}$$

which we can then use to determine if our numerical methods are physically accurate or not.

Figures 5.61 to 5.64 show the computed free surface at $t = 5$ from different angles. Similarly, Figures 5.65 to 5.68 show the computed free surface at $t = 10$ and Figures 5.69 to 5.72 show the computed free surface at $t = 15$. Table 5.6 shows the angles of the V-shaped waves of the free surface calculated using the Superbee limiter at $t = 15$. Table 5.7 shows the angles of the V-shaped waves of the free surface calculated using the Koren limiter at $t = 15$. These angles were calculated using the coordinates of the cells with the highest and lowest height values in one part of the V-shaped wave.

As expected, since the Froude-number $Fr = \frac{u}{\sqrt{gh}}$ is larger than 1 in this case, no waves propagate in the negative x -direction. Furthermore, with both limiters we are left with a stationary solution at $t = 15$, even though it went through an instationary phase prior to this.

The average measured height of the crest over all the simulations with both limiters, hence these are 8 in total, is 1.08. Furthermore, the average measured depth of the trough is 0.90 over all simulations. Using the formula given by (5.6) we can calculate that we expect the angle of the crest to be not far from 31.30 degrees and the angle of the trough to be around 28.38 degrees.

We can conclude from Tables 5.6 and 5.7 that each flux evaluation method with both limiters yields results not far from the previously calculated angles, thus our numerical methods seem well-suited for simulating this test case.

	Flux vector splitting	Midpoint rule	Trapezoidal rule	Roe's method
Crest angle	33.69	35.42	35.42	35.42
Trough angle	25.02	21.80	21.80	21.80

Table 5.6: Two-dimensional supercritical flow over a bump. Comparison of the angles of the crest and trough of the V-shaped waves of computed free surface using the Superbee limiter at $t = 15$.

	Flux vector splitting	Midpoint rule	Trapezoidal rule	Roe's method
Crest angle	31.89	34.56	34.56	34.56
Trough angle	23.96	21.80	22.89	22.89

Table 5.7: Two-dimensional supercritical flow over a bump. Comparison of the angles of the crest and trough of the V-shaped waves of computed free surface using the Koren limiter at $t = 15$.

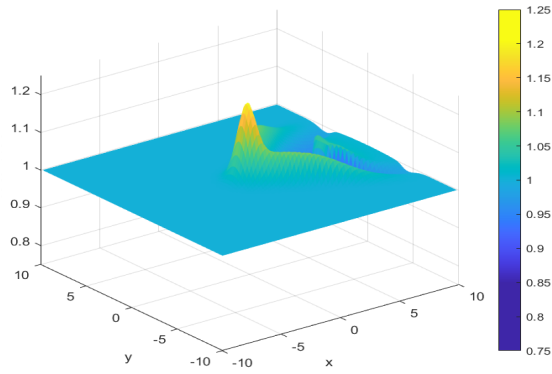


Figure 5.61: Two-dimensional supercritical flow over a bump. View of the computed free surface at $t = 5$ computed using the Koren limiter and Roe's method.

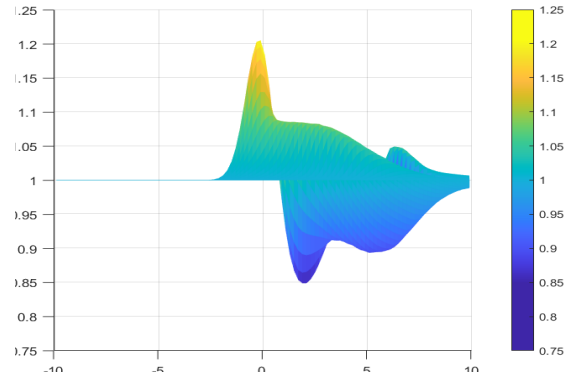


Figure 5.62: Two-dimensional supercritical flow over a bump. Side view of the computed free surface at $t = 5$ computed using the Koren limiter and Roe's method.

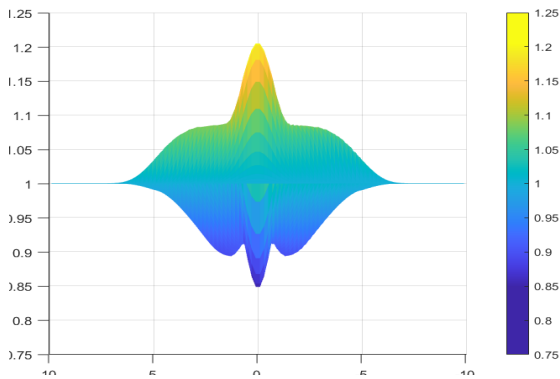


Figure 5.63: Two-dimensional supercritical flow over a bump. Front view of the computed free surface at $t = 5$ computed using the Koren limiter and Roe's method.

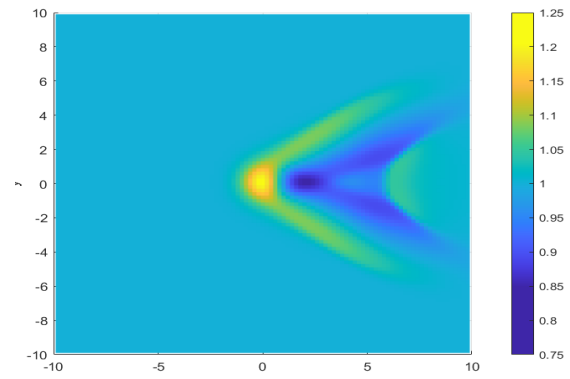


Figure 5.64: Two-dimensional supercritical flow over a bump. Top view of the computed free surface at $t = 5$ computed using the Koren limiter and Roe's method.

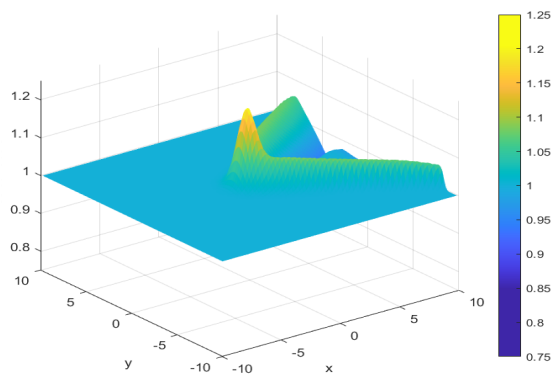


Figure 5.65: Two-dimensional supercritical flow over a bump. View of the computed free surface at $t = 10$ computed using the Koren limiter and Roe's method.

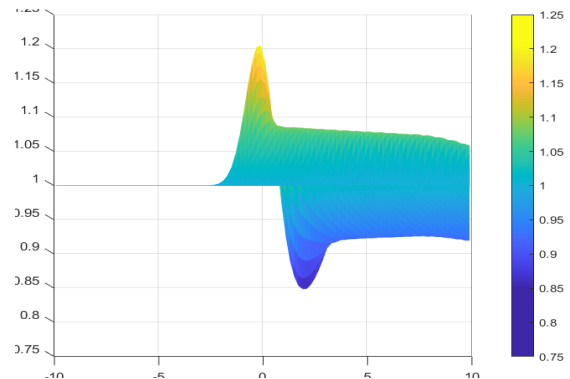


Figure 5.66: Two-dimensional supercritical flow over a bump. Side view of the computed free surface at $t = 10$ computed using the Koren limiter and Roe's method.

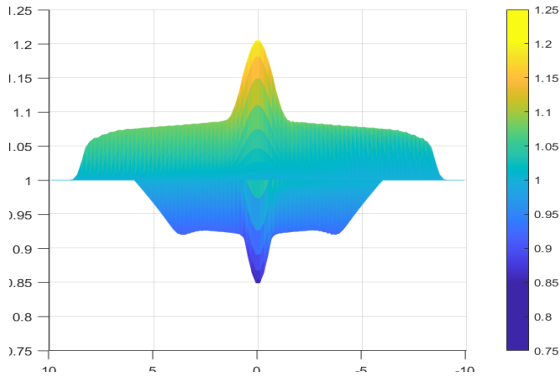


Figure 5.67: Two-dimensional supercritical flow over a bump. View of the computed free surface at $t = 10$ computed using the Koren limiter and Roe's method.

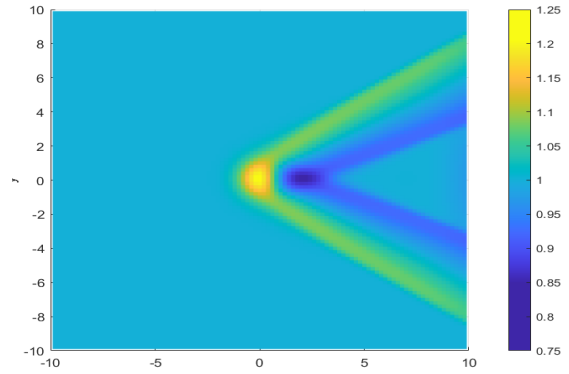


Figure 5.68: Two-dimensional supercritical flow over a bump. Side view of the computed free surface at $t = 10$ computed using the Koren limiter and Roe's method.

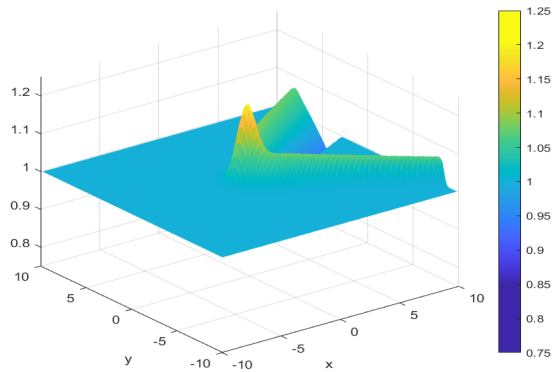


Figure 5.69: Two-dimensional supercritical flow over a bump. View of the computed free surface at $t = 15$ computed using the Koren limiter and Roe's method.

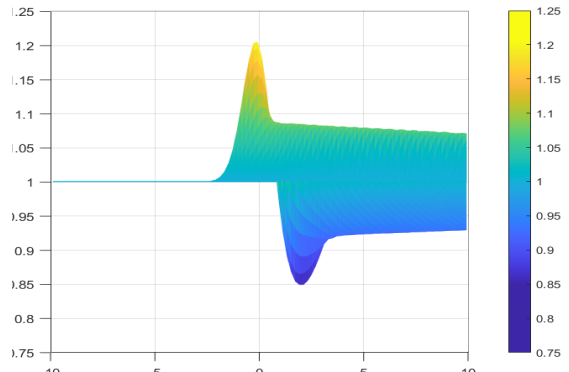


Figure 5.70: Two-dimensional supercritical flow over a bump. Side view of the computed free surface at $t = 15$ computed using the Koren limiter and Roe's method.

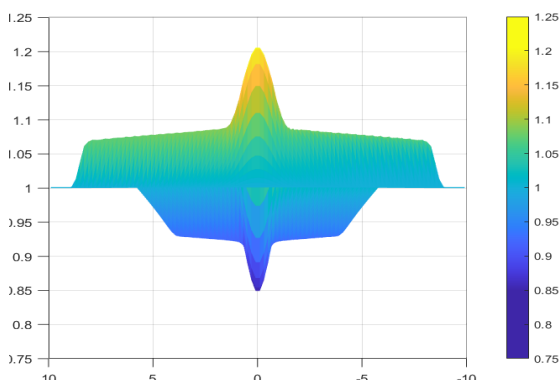


Figure 5.71: Two-dimensional supercritical flow over a bump. Front view of the computed free surface at $t = 15$ computed using the Koren limiter and Roe's method.

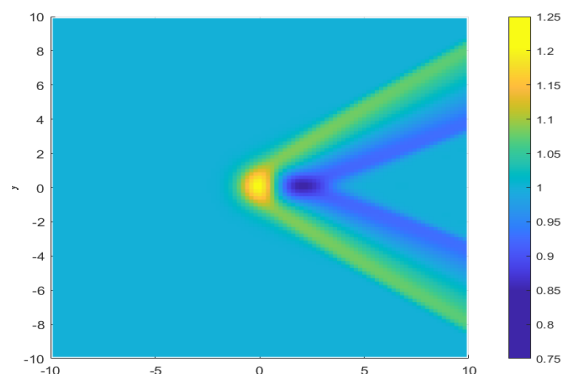


Figure 5.72: Two-dimensional supercritical flow over a bump. Top view of the computed free surface at $t = 15$ computed using the Koren limiter and Roe's method.

6 Conclusions

The main goal of this project was doing research on different numerical methods for the shallow water equations and comparing them by writing Matlab-code and running simulations.

In Chapter 3 we explored finite difference methods, but finite volume methods for the one-dimensional shallow water equations as well. We consider finite volume methods since finite difference methods are less suited when discontinuities occur or when the initial condition contains discontinuities. We considered multiple state interpolation methods, the most complex of which is the second-order method using limiter functions, and we derived a vast variety of flux evaluation methods based on different ideas. Furthermore, we chose a source term discretization and time integration. In Chapter 4 we extended these methods discussed in Chapter 3 for the two-dimensional shallow water case, but were unable to consider HLL-type solvers here as well.

In Chapter 5 we considered three different test cases to test our numerical methods and determine which one works best.

The first test case we considered was a one-dimensional water hill on a flat bottom, crucially the water hill considered was smooth at $t = 0$. This allowed us to compare how the different state interpolation methods and flux evaluation methods perform for smooth solution. What we learned from this is that the second-order limited state interpolation methods using either the Superbee or Koren limiter work best. Furthermore, the HLL-type solvers worked best in this case as well and have as advantage over the other flux evaluation methods that they require less computations and thus are faster.

The second test case we considered was a one-dimensional dam break on a flat bottom, crucially the dam break had a discontinuity at $x = 0$ at initial time $t = 0$. This allowed us to compare how the different state interpolation methods and flux evaluation methods perform for a discontinuous solution. What we learned from this is that again the second-order limited state interpolation methods using either the Superbee or Koren limiter work best. Moreover, anything but the flux vector splitting or trapezoidal rule method works well. The HLL-type solvers are still preferred in this case as they require strictly less computations.

The third and final test case we considered was a two-dimensional flow over a bump on the bottom, where the initial free surface height was the same over the whole domain. This allowed us to compare how the different state interpolation methods and flux evaluation methods perform in the 2D case. In this case we learned that the Koren limiter is better at representing reality than the Superbee limiter. Moreover, the flux vector splitting worked better than the other flux evaluation methods.

Finally, we succeeded in learning about and implementing various state interpolation and flux evaluation methods. The two 1D test cases yielded similar results and corroborated each other's results. The 2D case showed that even though the Superbee limiter seemed the best option for the 1D case, the Koren limiter is in the 2D test case we considered better at producing physically accurate results than the Superbee limiter.

7 Further research

We discuss the points at which the numerical integration methods and the simulations in this paper could be improved or what further research could focus on.

The source term discretization used for both the one-dimensional and two-dimensional shallow water equations is second-order accurate in space everywhere except at the boundaries. The source term discretization introduces small (barely noticeable) numerical waves at boundaries when the bottom is for example prescribed by a first-order polynomial. However, the numerical scheme is not exact in this case which is not wanted at all. Hence, one topic of interest is to discretize the source terms at the boundaries in such a way that the numerical scheme is exact in for example the case of a first-order polynomial.

The author was unable to generalize the HLL-methods, discussed in the one-dimensional case, to the 2D shallow water equations. Since these are by far the computationally fastest methods considered for the 1D case, they can likely be applied in a fast manner for the 2D case as well. Moreover, since computational speed is always of high importance when developing and testing numerical methods, this future research topic can be of great use to engineers and others who model shallow water situations.

Throughout the report we assumed that the bottom boundary is independent of the time t . The final test case we considered is about a bottom suddenly set into motion, but possibly more accurate results can be obtained by numerically solving the shallow water equations with a time-dependent bottom boundary which moves in a short amount of time but not instantaneous.

The sole purpose of this report was not to write the most efficient Matlab-code, but the focus lied on the mathematical part and reasoning behind the numerical methods and testing them. The code that was made and used to run the simulations is likely not very optimized and computationally efficient and consumer-grade hardware which was used to run the simulations is not the fastest way to get results, but even in the business case optimizing the Matlab-code can be hugely beneficial to companies, governments and other parties who run shallow water simulations.

References

- [1] BERMÚDEZ, A., VÁZQUEZ, M.E., Flux-vector and flux-difference splitting methods for the shallow water equations in a domain with variable depth, *Computer Modelling of Seas and Coastal Regions* (1992), pp. 255–267.
- [2] BRODTKORB, A.R., SÆTRA, M.L., ALTINAKAR, M., Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation, *Computers & Fluids*, **55** (2012), pp. 1–12.
- [3] DELESTRE, O., LUCAS, C., K SINANT, P., DARBOUX, F., LAGUERRE, C., VO, T.T.N, JAMES, F., CORDIER, S., SWASHES: a compilation of shallow water analytic solutions for hydraulic and environmental studies, *Journal of Scientific Computing*, **72** (2012), pp. 269–300.
- [4] EINFELDT, B., On Godunov-Type methods for gas dynamics, *SIAM Journal on Numerical Analysis*, **25** (1988), pp. 294–318.
- [5] FEHLBERG, E., Klassische Runge-Kutta-Formeln vierter und niedrigerer Ordnung mit schrittweisen-Kontrolle und ihre Anwendung auf Wärmeleitungsprobleme, *Computing*, **6** (1970), pp. 61–71.
- [6] HARTEN, A., High resolution schemes for hyperbolic conservation laws, *Journal of Computational Physics*, **49** (1983), pp. 357–393.
- [7] HARTEN, A., LAX, P.D., VAN LEER, B., On upstream differencing and Godunov-type schemes for hyperbolic conservation laws, *SIAM Review*, **25** (1983), pp. 35–61.
- [8] HUNSDORFER, W., KOREN, B, VAN LOON, M., VERWER, J.G., A positive finite-difference advection scheme, *Centrum voor Wiskunde en Informatica*, Report NM-R9309 (1993).
- [9] KOREN, B., A robust upwind discretization method for advection, diffusion and source terms, *Centrum voor Wiskunde en Informatica*, Report NM-R9308 (1993).
- [10] VAN LEER, B., Upwind-difference methods for aerodynamic problems governed by the Euler equations, *Lectures in Applied Mathematics, American Mathematical Society*, **22 - Part 2** (1985), pp. 327–336.
- [11] LEVEQUE, R.J., *Finite Volume Methods for Hyperbolic Problems*, Cambridge University Press (2002).
- [12] LIANG, Q., MARCHE, F., Numerical resolution of well-balanced shallow water equations with complex source terms, *Advances in Water Resources*, **32** (2009), pp. 873–884.
- [13] PATANKAR, S.V., *Numerical Heat Transfer and Fluid Flow*, McGraw-Hill Book Company (1980).
- [14] ROE, P.L., Characteristic-based schemes for the Euler equations, *Annual Review of Fluid Mechanics*, **18** (1986), pp. 337–365.
- [15] ROE, P.L., Approximate Riemann solvers, parameter vectors, and difference schemes, *Journal of Computational Physics*, **43** (1981), pp. 357–372.
- [16] STEGER, J.L., WARMING, R.F., Flux vector splitting of the inviscid gasdynamic equations with application to finite-difference methods, *Journal of Computational Physics*, **40** (1981), pp. 263–293.
- [17] STOKER, J.J., *Water Waves: The Mathematical Theory with Applications*, John Wiley & Sons, Inc (1992).
- [18] SWEBY, P.K., High resolution schemes using flux limiters for hyperbolic conservation laws, *SIAM Journal on Numerical Analysis*, **21** (1984), p. 995–1011.
- [19] TEUNISSEN, J.T., 3D Simulations and Analysis of Pulsed Discharges, PhD thesis, Eindhoven University of Technology, (2015).
- [20] TORO, E.F., *Shock-Capturing Methods for Free-Surface Shallow Flows*, John Wiley & Sons (2001).
- [21] TORO, E.F., *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*, Springer (2009).
- [22] VUIK, C., VERMOLEN, F.J., VAN GIJZEN, M.B., VUIK, M.J, *Numerical Methods for Ordinary Differential Equations*, Delft Academic Press (2015).

A Numerical implementation of limiters

In this appendix we discuss how to implement limiter functions and avoid division by zero cases, for which we continue on the work of Teunissen [19]. As an example, we work out this principle for the Superbee limiter, chosen for no particular reason at all.

Let us first recall the second-order limited scheme for the shallow water equations, which is given by

$$\bar{q}_{i+\frac{1}{2}}^l = \bar{q}_i + \frac{1}{2}\phi(r_{i+\frac{1}{2}}^l)(\bar{q}_i - \bar{q}_{i-1}), \quad (3.20a)$$

$$\bar{q}_{i+\frac{1}{2}}^r = \bar{q}_{i+1} + \frac{1}{2}\phi(r_{i+\frac{1}{2}}^r)(\bar{q}_{i+1} - \bar{q}_{i+2}), \quad (3.20b)$$

where

$$r_{i+\frac{1}{2}}^l = \frac{\bar{q}_{i+1} - \bar{q}_i}{\bar{q}_i - \bar{q}_{i-1}}, \quad r_{i+\frac{1}{2}}^r = \frac{\bar{q}_i - \bar{q}_{i+1}}{\bar{q}_{i+1} - \bar{q}_{i+2}}. \quad (3.21)$$

Furthermore, recall that the Superbee limiter is given by

$$\phi_{sb}(r) = \begin{cases} 0, & \text{if } r < 0 \\ \min(2r, 1), & \text{if } 0 \leq r < 1 \\ \min(2, r), & \text{if } r \geq 1. \end{cases} \quad (3.22)$$

Let us consider the left state interpolation given by (3.20a) using the Superbee limiter above. Now the fundamental step follows. Instead of using the function $\phi(r)$, the numerical implementation we used for the Superbee limiter is written in the following way:

$$\bar{q}_{i+\frac{1}{2}}^l = \bar{q}_i + \frac{1}{2}\phi_{t, sb}(\bar{q}_{i+1} - \bar{q}_i, \bar{q}_i - \bar{q}_{i-1}),$$

where $\phi_{t, sb}(a, b)$ is given by

$$\phi_{t, sb}(a, b) = \begin{cases} 0, & \text{if } ab \leq 0, \\ 2a, & \text{if } a^2 < \frac{1}{2}ab, \\ b, & \text{if } \frac{1}{2}ab \leq a^2 < ab, \\ a, & \text{if } ab \leq a^2 < 2ab, \\ 2b, & \text{if } a^2 \geq 2ab, \end{cases}$$

The interpretation above is based on the following logic: if $ab > 0$ then $a/b < x \iff a^2 < abx$. So, instead of looking at the fraction a/b , we look at a^2 and ab instead. It can be checked that for the cases that $ab > 0$ both schemes yield the exact same results.

Furthermore, we note that the interpretation that $\phi_{t, sb}(a, b) = 0$ if $ab \leq 0$ is a justified choice. Suppose that $a = 0$ and $b \neq 0$, then using (3.22) we have that $\phi_{sb} = 0$ and thus $\phi_s(a/b)b = 0$. Now, suppose that $a = b = 0$, then the states \bar{q}_{i-1} , \bar{q}_i and \bar{q}_{i+1} have the same value and we obviously want $\bar{q}_{i+\frac{1}{2}}^l$ to take this value as well, taking $\phi_{sb}(a/b)b = 0$ achieves this. Finally, suppose that $a \neq 0$ and $b = 0$, then either $\phi_{sb} = 0$ or $\phi_{sb} = 2$ depending on the sign of a , but since $b = 0$ we have that $\phi_{sb}(a/b)b = 0$. Hence, taking $\phi_{t, sb} = \phi_{sb}(a/b)b = 0$ if $ab \leq 0$ is a logical choice.

B Matlab code

B.1 Numerical implementation of state interpolation schemes

The Matlab code below shows the numerical implementation of the state interpolation methods, including limiters

```
1 function f = phi_t(number,a,b)
2
3     aa = a * a;
4     ab = a * b;
5
6     % first-order upwind
7     if number == 0
8         f = 0;
9
10    % kappa = 1/3
11    elseif number == 1
12        f = b * (1/3) + a * (2/3);
13
14    % minmod limiter
15    elseif number == 2
16        if ab <= 0
17            f = 0;
18        elseif aa < ab
19            f = a;
20        else
21            f = b;
22        end
23
24    % superbee limiter
25    elseif number == 3
26        if ab <= 0
27            f = 0;
28        elseif aa < (1/2) * ab
29            f = 2 * a;
30        elseif aa < ab
31            f = b;
32        elseif aa < 2 * ab
33            f = a;
34        else
35            f = 2 * b;
36        end
37
38    % Koren limiter
39    elseif number == 4
40        if ab <= 0
41            f = 0;
42        elseif aa < (1/4) * ab
43            f = 2 * a;
44        elseif aa < (5/2) * ab
45            f = (1/3) * b + (2/3) * a;
46        else
47            f = 2 * b;
48        end
49    end
50
51 end
```

B.2 Test case 1: State interpolation methods

The Matlab code below simulates multiple state interpolation methods in test case 1.

```

1 clear all
2 format long
3
4
5
6 %% Select state interpolation and flux evaluation methods
7
8 % state_inter      type
9 % -----
10 % 0                first-order upwind
11 % 1                kappa = 1/3
12 % 2                with minmod limiter
13 % 3                with superbee limiter
14 % 4                with Koren limiter
15
16
17
18 %% Define parameters
19
20 Nx = 100; % number of finite volumes (= cells)
21 CFL = 0.9; % Courant FriedrichsLewy condition
22 tmax = 3; % time at which the simulation ends
23 g = 1; % gravitational acceleration
24
25
26
27 %% Begin initial condition initialization
28
29 xmin = -8; % x-coordinate at the left boundary of the domain
30 xmax = 8; % x-coordinate at the right boundary of the domain
31 dx = ( xmax - xmin ) / Nx; % cell size
32
33 x = xmin:dx:xmax; % x-coordinates of the cell boundaries
34 xmid = (1/2)*( x(1:Nx) + x(2:Nx+1) ); % x-coordinates of the cell centres
35
36 % set the initial values for h, hu and b at the cell centres
37 b = zeros(Nx,1);
38 q = zeros(Nx,2);
39
40 for i=1:Nx
41     b(i) = 0; % b
42     if xmid(i) <= 0 % h
43         q(i,1) = 2 - b(i);
44     else
45         q(i,1) = 1 - b(i);
46     end
47     q(i,1) = 1 + exp(-xmid(i)^2); % h
48     q(i,2) = 0; % hu
49 end
50
51
52
53 %% Begin simulation
54
55 for k=1:5
56
57     state_inter = k-1;
58
59     b = zeros(Nx,1);
60     q = zeros(Nx,2);
61
62     for i=1:Nx
63         b(i) = 0; % b
64         if xmid(i) <= 0 % h
65             q(i,1) = 2 - b(i);
66         else
67             q(i,1) = 1 - b(i);
68         end
69         q(i,1) = 1 + exp(-xmid(i)^2); % h

```



```

70     q(i,2) = 0; % hu
71 end
72
73 % set the boundary values
74 q_left_boundary = q(1,:);
75 q_right_boundary = q(Nx,:);
76 q_left_boundary(2) = 0;
77 q_right_boundary(2) = 0;
78
79 ql = zeros(Nx+1,2);
80 qr = zeros(Nx+1,2);
81 F = zeros(Nx+1,2);
82
83 du1 = zeros(Nx,2);
84 du2 = zeros(Nx,2);
85 du3 = zeros(Nx,2);
86
87 t = 0;
88
89 while t < tmax
90
91     % calculate the timestep size
92     max_wave_speed = 0;
93     for i = 1:Nx
94         if abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1))) > max_wave_speed
95             max_wave_speed = abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1)));
96         end
97         if abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1))) > max_wave_speed
98             max_wave_speed = abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1)));
99         end
100     end
101     %if max_wave_speed > wave_speed
102     %    wave_speed = max_wave_speed;
103     %end
104     %max_wave_speed = 1.621310199408591;
105     dt = CFL * (dx / (2 * max_wave_speed) );
106
107     % if the simulation arrives at the final timestep
108     if t + dt - tmax > 0
109         dt = tmax - t;
110         break_this_loop = 1;
111     end
112
113     % adjust the time
114     t = t + dt;
115
116     %q_left_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
117     %q_right_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
118     %q_left_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
119     %q_right_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
120
121
122     qhelp = q; % used to calculate the values for h and hu in the intermediate RK3b steps
123
124     for stage = 1:3 % RK3b has three stages
125
126
127
128         %% Begin computation state interpolation
129
130         % note that ql(1,:) corresponds to q^{1/2} in the report and ql(Nx+1,:)
131         % corresponds to q^{1/2} in the report
132         for i = 1:Nx+1
133             if i==1
134                 ql(i,:) = q_left_boundary;
135             elseif i==2
136                 ql(i,:) = (1/2)*( qhelp(1,:) + qhelp(2,:) );
137             elseif i==Nx+1
138                 ql(i,:) = (3/2)*qhelp(Nx,:) - (1/2)*qhelp(Nx-1,:);
139             else
140                 for j=1:2

```

```

141         ql(i,j) = qhelp(i-1,j) + (1/2) * phi_t( state_inter , qhelp(i,j) -
142         qhelp(i-1,j) , qhelp(i-1,j) - qhelp(i-2,j) );
143         end
144     end
145
146     % note that qr(1,:) corresponds to q^r_{1/2} in the report and qr(Nx+1,:)
147     corresponds to q^r_{M+1/2} in the report
148     for i=1:Nx+1
149         if i==1
150             qr(i,:) = (3/2)*qhelp(1,:) - (1/2)*qhelp(2,:);
151         elseif i==Nx
152             qr(i,:) = (1/2)*( qhelp(Nx-1,:) + qhelp(Nx,:) );
153         elseif i==Nx+1
154             qr(i,:) = q_right_boundary;
155         else
156             for j=1:2
157                 qr(i,j) = qhelp(i,j) + (1/2) * phi_t( state_inter , qhelp(i-1,j) -
158                 qhelp(i,j) , qhelp(i,j) - qhelp(i+1,j) );
159             end
160         end
161     end
162
163     %% Flux method
164
165     % The HLL Solver
166     for i = 1:Nx+1
167         h_bar = (1/2)*(ql(i,1)+qr(i,1));
168         c_hat = sqrt(g*h_bar);
169         u_hat = ( (sqrt(ql(i,1))*(ql(i,2)/ql(i,1)))+(sqrt(qr(i,1))*(qr(i,2)/qr(i,1)))
170         )/(sqrt(ql(i,1))+sqrt(qr(i,1)));
171         lambda1 = u_hat - c_hat;
172         lambda2 = u_hat + c_hat;
173
174         sL = min(min(lambda1, lambda2), min((ql(i,2)/ql(i,1))+sqrt(g*ql(i,1)), (ql(i
175         ,2)/ql(i,1))-sqrt(g*ql(i,1))));
176         sR = max(max(lambda1, lambda2), max((qr(i,2)/qr(i,1))+sqrt(g*qr(i,1)), (qr(i
177         ,2)/qr(i,1))-sqrt(g*qr(i,1))));
178
179         if sL >= 0
180             F(i,1) = ql(i,2);
181             F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
182         elseif sR <= 0
183             F(i,1) = qr(i,2);
184             F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
185         else
186             F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR - sL
187             );
188             F(i,2) = ( sR*((ql(i,2)/ql(i,1))*ql(i,2) + (1/2)*g*(ql(i,1)^2)) - sL*((qr(
189             i,2)/qr(i,1))*qr(i,2) + (1/2)*g*(qr(i,1)^2)) + sR*sL*(qr(i,2)-ql(i,2)) ) / ( sR - sL );
190         end
191     end
192
193     %% Perform time-integration method
194
195     if stage == 1
196         du1(1,1)=- (dt/dx)*(F(2,1)-F(1,1));
197         du1(1,2)=- (dt/dx)*(F(2,2)-F(1,2))-dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
198         (2)-b(3))/(2*dx));
199         du1(Nx,1)=- (dt/dx)*(F(Nx+1,1)-F(Nx,1));
200         du1(Nx,2)=- (dt/dx)*(F(Nx+1,2)-F(Nx,2))-dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
201         b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
202         for i=2:Nx-1
203             du1(i,1)=- (dt/dx)*(F(i+1,1)-F(i,1));
204             du1(i,2)=- (dt/dx)*(F(i+1,2)-F(i,2))-dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
205             +1)-b(i-1))/(2*dx));
206         end
207         for i=3:Nx-2
208             for j=1:2

```

```

203         qhelp(i,j)=q(i,j)+du1(i,j);
204     end
205     end
206     elseif stage == 2
207         du2(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
208         du2(1,2)=-(dt/dx)*(F(2,2)-F(1,2))-dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
(2)-b(3))/(2*dx));
209         du2(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
210         du2(Nx,2)=-(dt/dx)*(F(Nx+1,2)-F(Nx,2))-dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
211         for i=2:Nx-1
212             du2(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
213             du2(i,2)=-(dt/dx)*(F(i+1,2)-F(i,2))-dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
+1)-b(i-1))/(2*dx));
214         end
215         for i=3:Nx-2
216             for j=1:2
217                 qhelp(i,j)=q(i,j)+(1/4)*du1(i,j)+(1/4)*du2(i,j);
218             end
219         end
220     elseif stage == 3
221         du3(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
222         du3(1,2)=-(dt/dx)*(F(2,2)-F(1,2))-dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
(2)-b(3))/(2*dx));
223         du3(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
224         du3(Nx,2)=-(dt/dx)*(F(Nx+1,2)-F(Nx,2))-dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
225         for i=2:Nx-1
226             du3(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
227             du3(i,2)=-(dt/dx)*(F(i+1,2)-F(i,2))-dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
+1)-b(i-1))/(2*dx));
228         end
229         for i=3:Nx-2
230             for j=1:2
231                 q(i,j)=q(i,j)+(1/6)*du1(i,j)+(1/6)*du2(i,j)+(4/6)*du3(i,j);
232             end
233         end
234     end
235 end
236 end
237
238 if k == 2
239     hold on
240 end
241 uplot = q(:,2)./q(:,1);
242 plot(xmid,uplot,'-o');
243 xlabel('x');
244 ylabel('free surface');
245 axis([xmin,xmax,-0.5,0.5]);
246 %title(['state \ interpolation =',num2str(state_inter),', CFL=',num2str(CFL),', Nx=',
num2str(Nx),', Tmax=',num2str(tmax),', t=',num2str(round(t,4))]);
247 grid on
248
249 end
250
251 %% Define parameters
252
253 Nx = 10000; % number of finite volumes (= cells)
254 state_inter = 4;
255
256
257
258 %% Begin initial condition initialization
259 dx = (xmax - xmin) / Nx; % cell size
260
261 x = xmin:dx:xmax; % x-coordinates of the cell boundaries
262 xmid = (1/2)*(x(1:Nx) + x(2:Nx+1)); % x-coordinates of the cell centres
263
264 b = zeros(Nx,1);
265 q = zeros(Nx,2);
266
267 for i=1:Nx
268     b(i) = 0; % b

```

```

269     if xmid(i) <= 0                               % h
270         q(i,1) = 2 - b(i);
271     else
272         q(i,1) = 1 - b(i);
273     end
274     q(i,1) = 1 + exp(-xmid(i)^2); % h
275     q(i,2) = 0; % h
276 end
277
278 % set the boundary values
279 q_left_boundary = q(1,:);
280 q_right_boundary = q(Nx,:);
281 q_left_boundary(2) = 0;
282 q_right_boundary(2) = 0;
283
284 q1 = zeros(Nx+1,2);
285 qr = zeros(Nx+1,2);
286 F = zeros(Nx+1,2);
287
288 du1 = zeros(Nx,2);
289 du2 = zeros(Nx,2);
290 du3 = zeros(Nx,2);
291
292 t = 0;
293
294 while t < tmax
295
296     % calculate the timestep size
297     max_wave_speed = 0;
298     for i = 1:Nx
299         if abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1))) > max_wave_speed
300             max_wave_speed = abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1)));
301         end
302         if abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1))) > max_wave_speed
303             max_wave_speed = abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1)));
304         end
305     end
306     %if max_wave_speed > wave_speed
307     %     wave_speed = max_wave_speed;
308     %end
309     %max_wave_speed = 1.621310199408591;
310     dt = CFL * (dx / (2 * max_wave_speed) );
311
312     % if the simulation arrives at the final timestep
313     if t + dt - tmax > 0
314         dt = tmax - t;
315         break_this_loop = 1;
316     end
317
318     % adjust the time
319     t = t + dt;
320
321     %q_left_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
322     %q_right_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
323     %q_left_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
324     %q_right_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
325
326
327
328     qhelp = q; % used to calculate the values for h and hu in the intermediate RK3b steps
329
330     for stage = 1:3 % RK3b has three stages
331
332
333
334         %% Begin computation state interpolation
335
336         % note that q1(1,:) corresponds to q^1_{1/2} in the report and q1(Nx+1,:)
337         % corresponds to q^1_{M+1/2} in the report
338         for i = 1:Nx+1
339             if i==1
340                 q1(i,:) = q_left_boundary;
341             elseif i==2

```

```

341         ql(i,:) = (1/2)*( qhelp(1,:) + qhelp(2,:) );
342     elseif i==Nx+1
343         ql(i,:) = (3/2)*qhelp(Nx,:) - (1/2)*qhelp(Nx-1,:);
344     else
345         for j=1:2
346             ql(i,j) = qhelp(i-1,j) + (1/2) * phi_t( state_inter , qhelp(i,j) -
qhelp(i-1,j) , qhelp(i-1,j) - qhelp(i-2,j) );
347         end
348     end
349 end
350
351 % note that qr(1,:) corresponds to q^r_{1/2} in the report and qr(Nx+1,:)
corresponds to q^r_{M+1/2} in the report
352 for i=1:Nx+1
353     if i==1
354         qr(i,:) = (3/2)*qhelp(1,:) - (1/2)*qhelp(2,:);
355     elseif i==Nx
356         qr(i,:) = (1/2)*( qhelp(Nx-1,:) + qhelp(Nx,:) );
357     elseif i==Nx+1
358         qr(i,:) = q_right_boundary;
359     else
360         for j=1:2
361             qr(i,j) = qhelp(i,j) + (1/2) * phi_t( state_inter , qhelp(i-1,j) -
qhelp(i,j) , qhelp(i,j) - qhelp(i+1,j) );
362         end
363     end
364 end
365
366
367 %% Flux method
368
369 % The HLLC Solver
370 for i = 1:Nx+1
371     h_bar = (1/2)*(ql(i,1)+qr(i,1));
372     c_hat = sqrt(g*h_bar);
373     u_hat = ( (sqrt(ql(i,1))*(ql(i,2)/ql(i,1)))+(sqrt(qr(i,1))*(qr(i,2)/qr(i,1)))
)/(sqrt(ql(i,1))+sqrt(qr(i,1)));
374     lambda1 = u_hat - c_hat;
375     lambda2 = u_hat + c_hat;
376
377     sL = min(min(lambda1, lambda2), min((ql(i,2)/ql(i,1))+sqrt(g*ql(i,1)), (ql(i
,2)/ql(i,1))-sqrt(g*ql(i,1))));
378     sR = max(max(lambda1, lambda2), max((qr(i,2)/qr(i,1))+sqrt(g*qr(i,1)), (qr(i
,2)/qr(i,1))-sqrt(g*qr(i,1))));
379
380     if sL >= 0
381         F(i,1) = ql(i,2);
382         F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
383     elseif sR <= 0
384         F(i,1) = qr(i,2);
385         F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
386     else
387         F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR - sL
);
388         F(i,2) = ( sR*((ql(i,2)/ql(i,1))*ql(i,2) + (1/2)*g*(ql(i,1)^2)) - sL*((qr(
i,2)/qr(i,1))*qr(i,2) + (1/2)*g*(qr(i,1)^2)) + sR*sL*(qr(i,2)-ql(i,2)) ) / ( sR - sL );
389     end
390 end
391
392
393
394 %% Perform time-integration method
395
396 if stage == 1
397     du1(1,1)=- (dt/dx)*(F(2,1)-F(1,1));
398     du1(1,2)=- (dt/dx)*(F(2,2)-F(1,2))-dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
(2)-b(3))/(2*dx));
399     du1(Nx,1)=- (dt/dx)*(F(Nx+1,1)-F(Nx,1));
400     du1(Nx,2)=- (dt/dx)*(F(Nx+1,2)-F(Nx,2))-dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
401     for i=2:Nx-1
402         du1(i,1)=- (dt/dx)*(F(i+1,1)-F(i,1));

```

```

404         du1(i,2) = -(dt/dx)*(F(i+1,2)-F(i,2)) - dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
+1)-b(i-1))/(2*dx));
405     end
406     for i=3:Nx-2
407         for j=1:2
408             qhelp(i,j) = q(i,j) + du1(i,j);
409         end
410     end
411     elseif stage == 2
412         du2(1,1) = -(dt/dx)*(F(2,1)-F(1,1));
413         du2(1,2) = -(dt/dx)*(F(2,2)-F(1,2)) - dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
(2)-b(3))/(2*dx));
414         du2(Nx,1) = -(dt/dx)*(F(Nx+1,1)-F(Nx,1));
415         du2(Nx,2) = -(dt/dx)*(F(Nx+1,2)-F(Nx,2)) - dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
416         for i=2:Nx-1
417             du2(i,1) = -(dt/dx)*(F(i+1,1)-F(i,1));
418             du2(i,2) = -(dt/dx)*(F(i+1,2)-F(i,2)) - dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
+1)-b(i-1))/(2*dx));
419         end
420         for i=3:Nx-2
421             for j=1:2
422                 qhelp(i,j) = q(i,j) + (1/4)*du1(i,j) + (1/4)*du2(i,j);
423             end
424         end
425     elseif stage == 3
426         du3(1,1) = -(dt/dx)*(F(2,1)-F(1,1));
427         du3(1,2) = -(dt/dx)*(F(2,2)-F(1,2)) - dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
(2)-b(3))/(2*dx));
428         du3(Nx,1) = -(dt/dx)*(F(Nx+1,1)-F(Nx,1));
429         du3(Nx,2) = -(dt/dx)*(F(Nx+1,2)-F(Nx,2)) - dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
430         for i=2:Nx-1
431             du3(i,1) = -(dt/dx)*(F(i+1,1)-F(i,1));
432             du3(i,2) = -(dt/dx)*(F(i+1,2)-F(i,2)) - dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
+1)-b(i-1))/(2*dx));
433         end
434         for i=3:Nx-2
435             for j=1:2
436                 q(i,j) = q(i,j) + (1/6)*du1(i,j) + (1/6)*du2(i,j) + (4/6)*du3(i,j);
437             end
438         end
439     end
440 end
441 end
442
443
444
445 uplot = q(:,2)./q(:,1);
446 plot(xmid, uplot, '-k');
447 xlabel('x');
448 ylabel('velocity u');
449 axis([xmin, xmax, -0.5, 0.5]);
450 %title(['state \ interpolation =', num2str(state_inter), ', CFL =', num2str(CFL), ', Nx =', num2str(
Nx), ', Tmax =', num2str(tmax), ', t =', num2str(round(t,4))]);
451 grid on
452 hold off
453 legend({'First-order upwind', 'Kappa = 1/3', 'Minmod limiter', 'Superbee limiter', 'Koren limiter
', ''}, 'Location', 'northwest', 'Orientation', 'vertical')
454
455
456
457 %% Begin plotting
458 %if mod(count, round(Nx^(1/2))) == 0
459 %    for i = 1:Nx
460 %        freude(i) = abs(q(i,2)/q(i,1))/sqrt(g*q(i,1));
461 %    end
462 %    surface = q(:,1) + b;
463 %    for i = 1:Nx
464 %        stationary(i) = q_left_boundary(2)/q_left_boundary(1);
465 %    end
466
467 %    hold off

```

```

468 % plot(xmid, surface, '-o', xmid, freude, '-o', xmid, b, '-o', xmid, stationary, '-');
469 % xlabel('x');
470 % ylabel('free surface');
471 % axis([xmin, xmax, 0, 2]);
472 % title(['state \ interpolation =', num2str(state_inter), ', flux \ evaluation =', num2str(
flux_eval), ', CFL =', num2str(CFL), ', Nx =', num2str(Nx), ', Tmax =', num2str(tmax), ', t =',
num2str(round(t, 4))]);
473 % hold on
474 % grid on
475 % drawnow
476 %end
477
478
479
480 %% Calculate total energy in the system
481 %energy(count) = 0;
482 %for i = 1:Nx
483 % energy(count) = energy(count) + ((1/2)*q(i,1)*((q(i,2)/q(i,1))^2)) + (g*q(i,1)*(q(i,1)
+2*b(i))/2);
484 %end
485 %energy(count) = energy(count) / Nx;
486
487
488
489 %% Error calculation
490 %error(count) = 0;
491 %for i = 1:Nx
492 % error(count) = error(count) + abs(q(i,1) + b(i) - q_left_boundary(1));
493 %end
494 %error(count) = error(count)/Nx;
495
496
497
498
499
500 %tiledlayout(2,1)
501
502 %% Plot error calculation
503 %nexttile
504 %plot(time, error, '-');
505 %xlabel('time');
506 %ylabel('error vs stationary solution');
507
508
509
510 %% Plot energy calculation
511 %nexttile
512 %plot(time, energy, '-');
513 %xlabel('time');
514 %ylabel('energy');
515
516
517
518 %wave_speed
519 %energy = 0;
520 %for i = 1:Nx
521 % energy = energy + ((1/2)*((q(i,2)/q(i,1))^2)) + (g*(q(i,1)+2*b(i))/2);
522 %end
523 %energy = energy / Nx;
524 %energy

```

B.3 Test case 1: Flux evaluation methods

The Matlab code below simulates multiple flux evaluation methods in test case 1.

```

1 clear all
2 format long
3
4
5
6 %% Select state interpolation and flux evaluation methods
7
8 % state_inter      type
9 % -----
10 % 0               first-order upwind
11 % 1               kappa = 1/3
12 % 2               with minmod limiter
13 % 3               with superbee limiter
14 % 4               with Koren limiter
15
16 % flux_eval       type
17 % -----
18 % 0               flux vector splitting
19 % 1               flux difference splitting: the midpoint rule
20 % 2               flux difference splitting: the trapezoidal rule
21 % 3               flux difference splitting: Roe's method (LeVeque 15.3.3 in Finite Volume
22 % 4               HLL Solver (Toro 10.4.1 with (10.18) in E.F. Toro – Shock-capturing methods
23 % 5               HLLC Solver (LeVeque 15.3.7 in Finite Volume Methods for Hyperbolic Problems
24 %               )
25
26
27 %% Define parameters
28
29 Nx = 100; % number of finite volumes (= cells)
30 CFL = 0.9; % Courant-Friedrichs-Lewy condition
31 tmax = 3; % time at which the simulation ends
32 g = 1; % gravitational acceleration
33
34 state_inter = 4;
35
36
37
38 %% Begin initial condition initialization
39
40 xmin = -10; % x-coordinate at the left boundary of the domain
41 xmax = 10; % x-coordinate at the right boundary of the domain
42 dx = (xmax - xmin) / Nx; % cell size
43
44 x = xmin:dx:xmax; % x-coordinates of the cell boundaries
45 xmid = (1/2)*(x(1:Nx) + x(2:Nx+1)); % x-coordinates of the cell centres
46
47
48
49 %% Begin simulation
50
51 for k=1:6
52
53     b = zeros(Nx,1);
54     q = zeros(Nx,2);
55
56     for i=1:Nx
57         b(i) = 0; % b
58         q(i,1) = 1 + exp(-xmid(i)^2); % h
59         q(i,2) = 0; % hu
60     end
61
62     % set the boundary values
63     q_left_boundary = q(1,:);
64     q_right_boundary = q(Nx,:);
65     q_left_boundary(2) = 0;
66     q_right_boundary(2) = 0;

```



```

67
68 q1 = zeros(Nx+1,2);
69 qr = zeros(Nx+1,2);
70 F = zeros(Nx+1,2);
71
72 du1 = zeros(Nx,2);
73 du2 = zeros(Nx,2);
74 du3 = zeros(Nx,2);
75
76 t = 0;
77
78 energy = 0;
79 for i = 1:Nx
80     energy = energy + ((1/2)*((q(i,2)/q(i,1))^2)) + (g*(q(i,1)+2*b(i))/2);
81 end
82 energy = energy / Nx;
83 energy
84
85 while t < tmax
86
87     % calculate the timestep size
88     max_wave_speed = 0;
89     for i = 1:Nx
90         if abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1))) > max_wave_speed
91             max_wave_speed = abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1)));
92         end
93         if abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1))) > max_wave_speed
94             max_wave_speed = abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1)));
95         end
96     end
97     %if max_wave_speed > wave_speed
98     %     wave_speed = max_wave_speed;
99     %end
100     %max_wave_speed = 1.621310199408591;
101     dt = CFL * (dx / (2 * max_wave_speed) );
102
103     % if the simulation arrives at the final timestep
104     if t + dt - tmax > 0
105         dt = tmax - t;
106         break_this_loop = 1;
107     end
108
109     % adjust the time
110     t = t + dt;
111
112     %q_left_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
113     %q_right_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
114     %q_left_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
115     %q_right_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
116
117
118
119     qhelp = q; % used to calculate the values for h and hu in the intermediate RK3b steps
120
121     for stage = 1:3 % RK3b has three stages
122
123
124
125         %% Begin computation state interpolation
126
127         % note that q1(1,:) corresponds to q^1_{1/2} in the report and q1(Nx+1,:)
128         % corresponds to q^1_{M+1/2} in the report
129         for i = 1:Nx+1
130             if i==1
131                 q1(i,:) = q_left_boundary;
132             elseif i==2
133                 q1(i,:) = (1/2)*( qhelp(1,:) + qhelp(2,:) );
134             elseif i==Nx+1
135                 q1(i,:) = (3/2)*qhelp(Nx,:) - (1/2)*qhelp(Nx-1,:);
136             else
137                 for j=1:2
138                     q1(i,j) = qhelp(i-1,j) + (1/2) * phi_t( state_inter , qhelp(i,j) -
139 qhelp(i-1,j), qhelp(i-1,j) - qhelp(i-2,j) );

```

```

138         end
139     end
140 end
141
142 % note that qr(1,:) corresponds to q^r_{1/2} in the report and qr(Nx+1,:)
corresponds to q^r_{M+1/2} in the report
143 for i=1:Nx+1
144     if i==1
145         qr(i,:) = (3/2)*qhhelp(1,:) - (1/2)*qhhelp(2,:);
146     elseif i==Nx
147         qr(i,:) = (1/2)*( qhhelp(Nx-1,:) + qhhelp(Nx,:) );
148     elseif i==Nx+1
149         qr(i,:) = q_right_boundary;
150     else
151         for j=1:2
152             qr(i,j) = qhhelp(i,j) + (1/2) * phi_t( state_inter , qhhelp(i-1,j) -
qhhelp(i,j) , qhhelp(i,j) - qhhelp(i+1,j) );
153         end
154     end
155 end
156
157
158 %% Flux method
159
160 % Flux vector splitting
161 if k == 1
162     for i = 1:Nx+1
163         lambda1L = (q1(i,2)/q1(i,1)) - sqrt(g*q1(i,1)/2);
164         lambda2L = (q1(i,2)/q1(i,1)) + sqrt(g*q1(i,1)/2);
165         diagL = diag([max(lambda1L,0) ; max(lambda2L,0)]);
166         RL = [ 1, 1; lambda1L, lambda2L ];
167         R_invL = (1/sqrt(2*g*q1(i,1))) * [ lambda2L, -1; -lambda1L, 1];
168
169         lambda1R = (qr(i,2)/qr(i,1)) - sqrt(g*qr(i,1)/2);
170         lambda2R = (qr(i,2)/qr(i,1)) + sqrt(g*qr(i,1)/2);
171         diagR = diag([min(lambda1R,0) ; min(lambda2R,0)]);
172         RR = [ 1, 1; lambda1R, lambda2R ];
173         R_invR = (1/sqrt(2*g*qr(i,1))) * [ lambda2R, -1; -lambda1R, 1];
174
175         F(i,:) = RL*diagL*R_invL*q1(i,:).' + RR*diagR*R_invR*qr(i,:).';
176     end
177
178
179 % Flux difference splitting: The midpoint rule
180 elseif k == 2
181     for i = 1:Nx+1
182         qm = (1/2)*( q1(i,:) + qr(i,:) );
183         lambda1 = (qm(2)/qm(1)) - sqrt(g*qm(1));
184         lambda2 = (qm(2)/qm(1)) + sqrt(g*qm(1));
185         diag_matrix = diag([abs(lambda1) ; abs(lambda2)]);
186         R = [ 1, 1; lambda1, lambda2 ];
187         R_inv = (1/(2*sqrt(g*qm(1)))) * [ lambda2, -1; -lambda1, 1];
188
189         F(i,:) = ( (1/2)*([q1(i,2) ; (q1(i,2)/q1(i,1))*q1(i,2)+((g/2)*(q1(i,1)^2)
)] + [qr(i,2) ; (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2))] ) - (1/2)*R*diag_matrix*
R_inv*((qr(i,:)-q1(i,:)).') ).';
190     end
191
192 % Flux difference splitting: The trapezoidal rule
193 elseif k == 3
194     for i = 1:Nx+1
195         lambda1L = (q1(i,2)/q1(i,1)) - sqrt(g*q1(i,1));
196         lambda2L = (q1(i,2)/q1(i,1)) + sqrt(g*q1(i,1));
197         diagL = diag([abs(lambda1L) ; abs(lambda2L)]);
198         RL = [ 1, 1; lambda1L, lambda2L ];
199         R_invL = (1/sqrt(2*g*q1(i,1))) * [ lambda2L, -1; -lambda1L, 1];
200
201         lambda1R = (qr(i,2)/qr(i,1)) - sqrt(g*qr(i,1));
202         lambda2R = (qr(i,2)/qr(i,1)) + sqrt(g*qr(i,1));
203         diagR = diag([abs(lambda1R) ; abs(lambda2R)]);
204         RR = [ 1, 1; lambda1R, lambda2R ];
205         R_invR = (1/sqrt(2*g*qr(i,1))) * [ lambda2R, -1; -lambda1R, 1];
206

```

```

207         F(i,:) = ( (1/2)*([ql(i,2) ; (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2)
    ] + [qr(i,2) ; (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2))] ) - (1/4)*(RL*diagL*R_invL
    + RR*diagR*R_invR)*((qr(i,:)-ql(i,:)).') ).';
208         end
209
210         % Flux difference splitting: Roe's method
211         elseif k == 4
212             for i = 1:Nx+1
213                 h_bar = (1/2)*(ql(i,1)+qr(i,1));
214                 c_hat = sqrt(g*h_bar);
215                 u_hat = ( (sqrt(ql(i,1))*(ql(i,2)/ql(i,1)))+(sqrt(qr(i,1))*(qr(i,2)/qr(i
    ,1))) )/(sqrt(ql(i,1))+sqrt(qr(i,1)));
216                 lambda1 = u_hat - c_hat;
217                 lambda2 = u_hat + c_hat;
218
219                 diag_matrix = diag([abs(lambda1) ; abs(lambda2)]);
220                 R = [ 1, 1; lambda1, lambda2 ];
221                 R_inv = (1/(2*c_hat)) * [ lambda2, -1; -lambda1, 1];
222
223                 F(i,:) = ( (1/2)*([ql(i,2) ; (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2)
    ] + [qr(i,2) ; (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2))] ) - (1/2)*R*diag_matrix*
    R_inv*((qr(i,:)-ql(i,:)).') ).';
224             end
225
226             % The HLL Solver
227             elseif k == 5
228                 for i = 1:Nx+1
229                     h_hat = (1/g)*(( (1/2)*(sqrt(g*ql(i,1))+sqrt(g*qr(i,1))) + (1/4)*((ql(i
    ,2)/ql(i,1))-qr(i,2)/qr(i,1)) ) ^2);
230
231                     if h_hat > ql(i,1)
232                         pl = sqrt( (1/2)*((h_hat*(h_hat+ql(i,1)))/(ql(i,1)^2)) );
233                     else
234                         pl = 1;
235                     end
236
237                     if h_hat > qr(i,1)
238                         pr = sqrt( (1/2)*((h_hat*(h_hat+qr(i,1)))/(qr(i,1)^2)) );
239                     else
240                         pr = 1;
241                     end
242
243                     sL = (ql(i,2)/ql(i,1)) - (sqrt(g*ql(i,1))*pl);
244                     sR = (qr(i,2)/qr(i,1)) + (sqrt(g*qr(i,1))*pr);
245
246                     if sL >= 0
247                         F(i,1) = ql(i,2);
248                         F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
249                     elseif sR <= 0
250                         F(i,1) = qr(i,2);
251                         F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
252                     else
253                         F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR
    - sL );
254                         F(i,2) = ( sR*((ql(i,2)/ql(i,1))*ql(i,2) + (1/2)*g*(ql(i,1)^2)) - sL
    *((qr(i,2)/qr(i,1))*qr(i,2) + (1/2)*g*(qr(i,1)^2)) + sR*sL*(qr(i,2)-ql(i,2)) ) / ( sR - sL
    );
255                     end
256                 end
257
258             % The HLL Solver
259             elseif k == 6
260                 for i = 1:Nx+1
261                     h_bar = (1/2)*(ql(i,1)+qr(i,1));
262                     c_hat = sqrt(g*h_bar);
263                     u_hat = ( (sqrt(ql(i,1))*(ql(i,2)/ql(i,1)))+(sqrt(qr(i,1))*(qr(i,2)/qr(i
    ,1))) )/(sqrt(ql(i,1))+sqrt(qr(i,1)));
264                     lambda1 = u_hat - c_hat;
265                     lambda2 = u_hat + c_hat;
266
267                     sL = min(min(lambda1, lambda2), min((ql(i,2)/ql(i,1))+sqrt(g*ql(i,1)), (ql
    (i,2)/ql(i,1))-sqrt(g*ql(i,1))));

```

```
268         sR = max(max(lambda1, lambda2), max((qr(i,2)/qr(i,1))+sqrt(g*qr(i,1)), (qr
269         (i,2)/qr(i,1))-sqrt(g*qr(i,1))));
270
271         if sL >= 0
272             F(i,1) = ql(i,2);
273             F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
274         elseif sR <= 0
275             F(i,1) = qr(i,2);
276             F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
277         else
278             F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR
279             - sL );
280             F(i,2) = ( sR*((ql(i,2)/ql(i,1))*ql(i,2) + (1/2)*g*(ql(i,1)^2)) - sL
281             *((qr(i,2)/qr(i,1))*qr(i,2) + (1/2)*g*(qr(i,1)^2)) + sR*sL*(qr(i,2)-ql(i,2)) ) / ( sR - sL
282             );
283         end
284     end
285 end
286
287 %% Perform time-integration method
288
289 if stage == 1
290     du1(1,1)=- (dt/dx)*(F(2,1)-F(1,1));
291     du1(1,2)=- (dt/dx)*(F(2,2)-F(1,2))-dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
292     (2)-b(3))/(2*dx));
293     du1(Nx,1)=- (dt/dx)*(F(Nx+1,1)-F(Nx,1));
294     du1(Nx,2)=- (dt/dx)*(F(Nx+1,2)-F(Nx,2))-dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
295     b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
296     for i=2:Nx-1
297         du1(i,1)=- (dt/dx)*(F(i+1,1)-F(i,1));
298         du1(i,2)=- (dt/dx)*(F(i+1,2)-F(i,2))-dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
299         +1)-b(i-1))/(2*dx));
300     end
301     for i=3:Nx-2
302         for j=1:2
303             qhelp(i,j)=q(i,j)+du1(i,j);
304         end
305     end
306 elseif stage == 2
307     du2(1,1)=- (dt/dx)*(F(2,1)-F(1,1));
308     du2(1,2)=- (dt/dx)*(F(2,2)-F(1,2))-dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
309     (2)-b(3))/(2*dx));
310     du2(Nx,1)=- (dt/dx)*(F(Nx+1,1)-F(Nx,1));
311     du2(Nx,2)=- (dt/dx)*(F(Nx+1,2)-F(Nx,2))-dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
312     b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
313     for i=2:Nx-1
314         du2(i,1)=- (dt/dx)*(F(i+1,1)-F(i,1));
315         du2(i,2)=- (dt/dx)*(F(i+1,2)-F(i,2))-dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
316         +1)-b(i-1))/(2*dx));
317     end
318     for i=3:Nx-2
319         for j=1:2
320             qhelp(i,j)=q(i,j)+(1/4)*du1(i,j)+(1/4)*du2(i,j);
321         end
322     end
323 elseif stage == 3
324     du3(1,1)=- (dt/dx)*(F(2,1)-F(1,1));
325     du3(1,2)=- (dt/dx)*(F(2,2)-F(1,2))-dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
326     (2)-b(3))/(2*dx));
327     du3(Nx,1)=- (dt/dx)*(F(Nx+1,1)-F(Nx,1));
328     du3(Nx,2)=- (dt/dx)*(F(Nx+1,2)-F(Nx,2))-dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
329     b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
330     for i=2:Nx-1
331         du3(i,1)=- (dt/dx)*(F(i+1,1)-F(i,1));
332         du3(i,2)=- (dt/dx)*(F(i+1,2)-F(i,2))-dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
333         +1)-b(i-1))/(2*dx));
334     end
335     for i=3:Nx-2
336         for j=1:2
337             q(i,j)=q(i,j)+(1/6)*du1(i,j)+(1/6)*du2(i,j)+(4/6)*du3(i,j);
338         end
339     end
```

```

328         end
329     end
330 end
331 end
332
333     if k == 2
334         hold on
335     end
336     plot(xmid,q(:,2)./q(:,1),'-o');
337     xlabel('x');
338     ylabel('velocity u');
339     axis([xmin,xmax,-0.5,0.5]);
340     %title(['state \ interpolation =',num2str(state_inter),', CFL=',num2str(CFL),', Nx=',
341         num2str(Nx),', Tmax=',num2str(tmax),', t=',num2str(round(t,4))]);
342     grid on
343 end
344
345 %% Define parameters
346
347 Nx = 10000; % number of finite volumes (= cells)
348 wave_speed = 0;
349
350 %% Begin initial condition initialization
351 dx = (xmax - xmin) / Nx; % cell size
352
353 x = xmin:dx:xmax; % x-coordinates of the cell boundaries
354 xmid = (1/2)*(x(1:Nx) + x(2:Nx+1)); % x-coordinates of the cell centres
355
356 b = zeros(Nx,1);
357 q = zeros(Nx,2);
358
359 for i=1:Nx
360     b(i) = 0; % b
361     if xmid(i) <= 0 % h
362         q(i,1) = 2 - b(i);
363     else
364         q(i,1) = 1 - b(i);
365     end
366     q(i,1) = 1 + exp(-xmid(i)^2); % h
367     q(i,2) = 0; % h
368 end
369
370 % set the boundary values
371 q_left_boundary = q(1,:);
372 q_right_boundary = q(Nx,:);
373 q_left_boundary(2) = 0;
374 q_right_boundary(2) = 0;
375
376 q1 = zeros(Nx+1,2);
377 qr = zeros(Nx+1,2);
378 F = zeros(Nx+1,2);
379
380 du1 = zeros(Nx,2);
381 du2 = zeros(Nx,2);
382 du3 = zeros(Nx,2);
383
384 t = 0;
385
386 while t < tmax
387
388     % calculate the timestep size
389     max_wave_speed = 0;
390     for i = 1:Nx
391         if abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1))) > max_wave_speed
392             max_wave_speed = abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1)));
393         end
394         if abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1))) > max_wave_speed
395             max_wave_speed = abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1)));
396         end
397     end
398     end
399     if max_wave_speed > wave_speed

```

```

400     wave_speed = max_wave_speed;
401     end
402     dt = CFL * (dx / (2 * max_wave_speed) );
403
404     % if the simulation arrives at the final timestep
405     if t + dt - tmax > 0
406         dt = tmax - t;
407         break_this_loop = 1;
408     end
409
410     % adjust the time
411     t = t + dt;
412
413     %q_left_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
414     %q_right_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
415     %q_left_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
416     %q_right_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
417
418
419
420     qhelp = q; % used to calculate the values for h and hu in the intermediate RK3b steps
421
422     for stage = 1:3 % RK3b has three stages
423
424
425
426         %% Begin computation state interpolation
427
428         % note that ql(1,:) corresponds to q^l_{1/2} in the report and ql(Nx+1,:)
429         corresponds to q^l_{M+1/2} in the report
430         for i = 1:Nx+1
431             if i==1
432                 ql(i,:) = q_left_boundary;
433             elseif i==2
434                 ql(i,:) = (1/2)*( qhelp(1,:) + qhelp(2,:) );
435             elseif i==Nx+1
436                 ql(i,:) = (3/2)*qhelp(Nx,:) - (1/2)*qhelp(Nx-1,:);
437             else
438                 for j=1:2
439                     ql(i,j) = qhelp(i-1,j) + (1/2) * phi_t( state_inter , qhelp(i,j) -
440                     qhelp(i-1,j) , qhelp(i-1,j) - qhelp(i-2,j) );
441                 end
442             end
443
444         % note that qr(1,:) corresponds to q^r_{1/2} in the report and qr(Nx+1,:)
445         corresponds to q^r_{M+1/2} in the report
446         for i=1:Nx+1
447             if i==1
448                 qr(i,:) = (3/2)*qhelp(1,:) - (1/2)*qhelp(2,:);
449             elseif i==Nx
450                 qr(i,:) = (1/2)*( qhelp(Nx-1,:) + qhelp(Nx,:) );
451             elseif i==Nx+1
452                 qr(i,:) = q_right_boundary;
453             else
454                 for j=1:2
455                     qr(i,j) = qhelp(i,j) + (1/2) * phi_t( state_inter , qhelp(i-1,j) -
456                     qhelp(i,j) , qhelp(i,j) - qhelp(i+1,j) );
457                 end
458             end
459
460         %% Flux method
461         for i = 1:Nx+1
462             h_bar = (1/2)*(ql(i,1)+qr(i,1));
463             c_hat = sqrt(g*h_bar);
464             u_hat = ( (sqrt(ql(i,1))*(ql(i,2)/ql(i,1)))+(sqrt(qr(i,1))*(qr(i,2)/qr(i
465             ,1))) )/(sqrt(ql(i,1))+sqrt(qr(i,1)));
466             lambda1 = u_hat - c_hat;
467             lambda2 = u_hat + c_hat;

```

```
468         sL = min(min(lambda1, lambda2), min((ql(i,2)/ql(i,1))+sqrt(g*ql(i,1)), (ql
469         (i,2)/ql(i,1))-sqrt(g*ql(i,1))));
470         sR = max(max(lambda1, lambda2), max((qr(i,2)/qr(i,1))+sqrt(g*qr(i,1)), (qr
471         (i,2)/qr(i,1))-sqrt(g*qr(i,1))));
472
473         if sL >= 0
474             F(i,1) = ql(i,2);
475             F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
476         elseif sR <= 0
477             F(i,1) = qr(i,2);
478             F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
479         else
480             F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR
481             - sL );
482             F(i,2) = ( sR*((ql(i,2)/ql(i,1))*ql(i,2) + (1/2)*g*(ql(i,1)^2)) - sL
483             *((qr(i,2)/qr(i,1))*qr(i,2) + (1/2)*g*(qr(i,1)^2)) + sR*sL*(qr(i,2)-ql(i,2)) ) / ( sR - sL
484             );
485         end
486     end
487
488     %% Perform time-integration method
489
490     if stage == 1
491         du1(1,1) = -(dt/dx)*(F(2,1)-F(1,1));
492         du1(1,2) = -(dt/dx)*(F(2,2)-F(1,2))-dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
493         (2)-b(3))/(2*dx));
494         du1(Nx,1) = -(dt/dx)*(F(Nx+1,1)-F(Nx,1));
495         du1(Nx,2) = -(dt/dx)*(F(Nx+1,2)-F(Nx,2))-dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
496         b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
497         for i=2:Nx-1
498             du1(i,1) = -(dt/dx)*(F(i+1,1)-F(i,1));
499             du1(i,2) = -(dt/dx)*(F(i+1,2)-F(i,2))-dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
500             +1)-b(i-1))/(2*dx));
501         end
502         for i=3:Nx-2
503             for j=1:2
504                 qhelp(i,j) = q(i,j)+du1(i,j);
505             end
506         end
507     elseif stage == 2
508         du2(1,1) = -(dt/dx)*(F(2,1)-F(1,1));
509         du2(1,2) = -(dt/dx)*(F(2,2)-F(1,2))-dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
510         (2)-b(3))/(2*dx));
511         du2(Nx,1) = -(dt/dx)*(F(Nx+1,1)-F(Nx,1));
512         du2(Nx,2) = -(dt/dx)*(F(Nx+1,2)-F(Nx,2))-dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
513         b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
514         for i=2:Nx-1
515             du2(i,1) = -(dt/dx)*(F(i+1,1)-F(i,1));
516             du2(i,2) = -(dt/dx)*(F(i+1,2)-F(i,2))-dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
517             +1)-b(i-1))/(2*dx));
518         end
519         for i=3:Nx-2
520             for j=1:2
521                 qhelp(i,j) = q(i,j)+(1/4)*du1(i,j)+(1/4)*du2(i,j);
522             end
523         end
524     elseif stage == 3
525         du3(1,1) = -(dt/dx)*(F(2,1)-F(1,1));
526         du3(1,2) = -(dt/dx)*(F(2,2)-F(1,2))-dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
527         (2)-b(3))/(2*dx));
528         du3(Nx,1) = -(dt/dx)*(F(Nx+1,1)-F(Nx,1));
529         du3(Nx,2) = -(dt/dx)*(F(Nx+1,2)-F(Nx,2))-dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
530         b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
531         for i=2:Nx-1
532             du3(i,1) = -(dt/dx)*(F(i+1,1)-F(i,1));
533             du3(i,2) = -(dt/dx)*(F(i+1,2)-F(i,2))-dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
534             +1)-b(i-1))/(2*dx));
535         end
536         for i=3:Nx-2
537             for j=1:2
538                 q(i,j) = q(i,j)+(1/6)*du1(i,j)+(1/6)*du2(i,j)+(4/6)*du3(i,j);
539             end
540         end
```

```

527         end
528     end
529 end
530 end
531 end
532
533
534
535
536 plot(xmid,q(:,2)./q(:,1),'-k');
537 xlabel('x');
538 ylabel('velocity u');
539 axis([xmin,xmax,-0.5,0.5]);
540 %title(['state \ interpolation =',num2str(state_inter),' CFL=',num2str(CFL),' Nx=',num2str(
541 Nx),' Tmax=',num2str(tmax),' t =',num2str(round(t,4))]);
542 grid on
543 hold off
544 legend({'Flux vector splitting ','Midpoint rule ','Trapezoidal rule ','Roe''s method ','HLL solver
545 ','HLLC solver ',' ','Location ','northwest ','Orientation ','vertical '})
546
547 %% Begin plotting
548 %if mod(count,round(Nx^(1/2))) == 0
549 %    for i = 1:Nx
550 %        freude(i) = abs(q(i,2)/q(i,1))/sqrt(g*q(i,1));
551 %    end
552 %    surface = q(:,1) + b;
553 %    for i = 1:Nx
554 %        stationary(i) = q_left_boundary(2)/q_left_boundary(1);
555 %    end
556
557 %    hold off
558 %    plot(xmid,surface,'-o',xmid,freude,'-o',xmid,b,'-o',xmid,stationary,'-');
559 %    xlabel('x');
560 %    ylabel('free surface ');
561 %    axis([xmin,xmax,0,2]);
562 %    title(['state \ interpolation =',num2str(state_inter),' flux \ evaluation =',num2str(
563 flux_eval),' CFL=',num2str(CFL),' Nx=',num2str(Nx),' Tmax=',num2str(tmax),' t =',
564 num2str(round(t,4))]);
565 %    hold on
566 %    grid on
567 %    drawnow
568 %end
569
570 %% Calculate total energy in the system
571 %energy(count) = 0;
572 %for i = 1:Nx
573 %    energy(count) = energy(count) + ((1/2)*q(i,1)*((q(i,2)/q(i,1))^2)) + (g*q(i,1)*(q(i,1)
574 +2*b(i))/2);
575 %end
576 %energy(count) = energy(count) / Nx;
577
578 %% Error calculation
579 %error(count) = 0;
580 %for i = 1:Nx
581 %    error(count) = error(count) + abs(q(i,1) + b(i) - q_left_boundary(1));
582 %end
583 %error(count) = error(count)/Nx;
584
585
586
587
588
589
590 %tiledlayout(2,1)
591
592 %% Plot error calculation
593 %nexttile
594 %plot(time,error,'-');

```



```
595 xlabel('time');
596 ylabel('error vs stationary solution');
597
598
599
600 %% Plot energy calculation
601 %nexttile
602 plot(time,energy,'-');
603 xlabel('time');
604 ylabel('energy');
605
606
607
608 %wave_speed
609 energy = 0;
610 for i = 1:Nx
611     energy = energy + ((1/2)*((q(i,2)/q(i,1))^2)) + (g*(q(i,1)+2*b(i))/2);
612 end
613 energy = energy / Nx;
614 energy
615 wave_speed
```

B.4 Test case 2: State interpolation methods

The Matlab code below simulates multiple state interpolation methods in test case 2.

```
1 clear all
2 format long
3
4
5
6 %% Select state interpolation and flux evaluation methods
7
8 % state_inter      type
9 % -----
10 % 0               first-order upwind
11 % 1               kappa = 1/3
12 % 2               with minmod limiter
13 % 3               with superbee limiter
14 % 4               with Koren limiter
15
16 % flux_eval       type
17 % -----
18 % 0               flux vector splitting
19 % 1               flux difference splitting: the midpoint rule
20 % 2               flux difference splitting: the trapezoidal rule
21 % 3               flux difference splitting: Roe's method (LeVeque 15.3.3 in Finite Volume
22 % 4               HLL Solver (Toro 10.4.1 with (10.18) in E.F. Toro – Shock-capturing methods
23 % 5               HLL Solver (LeVeque 15.3.7 in Finite Volume Methods for Hyperbolic Problems
24 %               )
25
26
27 %% Define parameters
28
29 Nx = 100; % number of finite volumes (= cells)
30 CFL = 0.45; % Courant FriedrichsLewy condition
31 tmax = 1; % time at which the simulation ends
32 g = 1; % gravitational acceleration
33
34
35
36 %% Begin initial condition initialization
37
38 xmin = -8; % x-coordinate at the left boundary of the domain
39 xmax = 8; % x-coordinate at the right boundary of the domain
40 dx = ( xmax - xmin ) / Nx; % cell size
41
42 x = xmin:dx:xmax; % x-coordinates of the cell boundaries
43 xmid = (1/2)*( x(1:Nx) + x(2:Nx+1) ); % x-coordinates of the cell centres
44
45 h = zeros(Nx,5);
46 u = zeros(Nx,5);
47
48
49
50 %% Begin simulation
51
52 for k=1:5
53
54     state_inter = k-1;
55
56     b = zeros(Nx,1);
57     q = zeros(Nx,2);
58
59     for i=1:Nx
60         b(i) = 0; % b
61         if xmid(i) <= 0 % h
62             q(i,1) = 2 - b(i);
63         else
64             q(i,1) = 1 - b(i);
65         end
66         q(i,2) = 0; % hu
```

```

67 end
68
69 % set the boundary values
70 q_left_boundary = q(1,:);
71 q_right_boundary = q(Nx,:);
72 q_left_boundary(2) = 0;
73 q_right_boundary(2) = 0;
74
75 q1 = zeros(Nx+1,2);
76 qr = zeros(Nx+1,2);
77 F = zeros(Nx+1,2);
78
79 du1 = zeros(Nx,2);
80 du2 = zeros(Nx,2);
81 du3 = zeros(Nx,2);
82
83 t = 0;
84
85 while t < tmax
86
87     % calculate the timestep size
88     max_wave_speed = 0;
89     for i = 1:Nx
90         if abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1))) > max_wave_speed
91             max_wave_speed = abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1)));
92         end
93         if abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1))) > max_wave_speed
94             max_wave_speed = abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1)));
95         end
96     end
97     %if max_wave_speed > wave_speed
98     %     wave_speed = max_wave_speed;
99     %end
100     %max_wave_speed = 1.621310199408591;
101     dt = CFL * (dx / (2 * max_wave_speed) );
102
103     % if the simulation arrives at the final timestep
104     if t + dt - tmax > 0
105         dt = tmax - t;
106         break_this_loop = 1;
107     end
108
109     % adjust the time
110     t = t + dt;
111
112     %q_left_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
113     %q_right_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
114     %q_left_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
115     %q_right_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
116
117
118
119     qhelp = q; % used to calculate the values for h and hu in the intermediate RK3b steps
120
121     for stage = 1:3 % RK3b has three stages
122
123
124
125         %% Begin computation state interpolation
126
127         % note that q1(1,:) corresponds to q^1_{1/2} in the report and q1(Nx+1,:)
128         % corresponds to q^1_{M+1/2} in the report
129         for i = 1:Nx+1
130             if i==1
131                 q1(i,:) = q_left_boundary;
132             elseif i==2
133                 q1(i,:) = (1/2)*( qhelp(1,:) + qhelp(2,:) );
134             elseif i==Nx+1
135                 q1(i,:) = (3/2)*qhelp(Nx,:) - (1/2)*qhelp(Nx-1,:);
136             else
137                 for j=1:2
138                     q1(i,j) = qhelp(i-1,j) + (1/2) * phi_t( state_inter , qhelp(i,j) -
139 qhelp(i-1,j), qhelp(i-1,j) - qhelp(i-2,j) );

```

```

138         end
139     end
140 end
141
142 % note that qr(1,:) corresponds to  $q^{r_{1/2}}$  in the report and qr(Nx+1,:)
corresponds to  $q^{r_{M+1/2}}$  in the report
143 for i=1:Nx+1
144     if i==1
145         qr(i,:) = (3/2)*qhhelp(1,:) - (1/2)*qhhelp(2,:);
146     elseif i==Nx
147         qr(i,:) = (1/2)*( qhhelp(Nx-1,:) + qhhelp(Nx,:) );
148     elseif i==Nx+1
149         qr(i,:) = q_right_boundary;
150     else
151         for j=1:2
152             qr(i,j) = qhhelp(i,j) + (1/2) * phi_t( state_inter , qhhelp(i-1,j) -
qhhelp(i,j) , qhhelp(i,j) - qhhelp(i+1,j) );
153         end
154     end
155 end
156
157
158 %% Flux method
159
160 % The HLLC Solver
161 for i = 1:Nx+1
162     h_bar = (1/2)*(ql(i,1)+qr(i,1));
163     c_hat = sqrt(g*h_bar);
164     u_hat = ( (sqrt(ql(i,1))*(ql(i,2)/ql(i,1)))+(sqrt(qr(i,1))*(qr(i,2)/qr(i,1)))
)/(sqrt(ql(i,1))+sqrt(qr(i,1)));
165     lambda1 = u_hat - c_hat;
166     lambda2 = u_hat + c_hat;
167
168     sL = min(min(lambda1, lambda2), min((ql(i,2)/ql(i,1))+sqrt(g*ql(i,1)), (ql(i
,2)/ql(i,1))-sqrt(g*ql(i,1))));
169     sR = max(max(lambda1, lambda2), max((qr(i,2)/qr(i,1))+sqrt(g*qr(i,1)), (qr(i
,2)/qr(i,1))-sqrt(g*qr(i,1))));
170
171     if sL >= 0
172         F(i,1) = ql(i,2);
173         F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2) + ((g/2)*(ql(i,1)^2));
174     elseif sR <= 0
175         F(i,1) = qr(i,2);
176         F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2) + ((g/2)*(qr(i,1)^2));
177     else
178         F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR - sL
);
179         F(i,2) = ( sR*((ql(i,2)/ql(i,1))*ql(i,2) + (1/2)*g*(ql(i,1)^2)) - sL*((qr(i
,2)/qr(i,1))*qr(i,2) + (1/2)*g*(qr(i,1)^2)) + sR*sL*(qr(i,2)-ql(i,2)) ) / ( sR - sL );
180     end
181 end
182 end
183
184
185 %% Perform time-integration method
186
187 if stage == 1
188     du1(1,1) = -(dt/dx)*(F(2,1)-F(1,1));
189     du1(1,2) = -(dt/dx)*(F(2,2)-F(1,2)) - dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
(2)-b(3))/(2*dx));
190     du1(Nx,1) = -(dt/dx)*(F(Nx+1,1)-F(Nx,1));
191     du1(Nx,2) = -(dt/dx)*(F(Nx+1,2)-F(Nx,2)) - dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
192     for i=2:Nx-1
193         du1(i,1) = -(dt/dx)*(F(i+1,1)-F(i,1));
194         du1(i,2) = -(dt/dx)*(F(i+1,2)-F(i,2)) - dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
+1)-b(i-1))/(2*dx));
195     end
196     for i=3:Nx-2
197         for j=1:2
198             qhhelp(i,j) = q(i,j) + du1(i,j);
199         end
200

```

```

201     end
202     elseif stage == 2
203         du2(1,1) = -(dt/dx) * (F(2,1) - F(1,1));
204         du2(1,2) = -(dt/dx) * (F(2,2) - F(1,2)) - dt * g * ((1/2) * (qr(1,1) + ql(2,1))) * ((-3 * b(1) + 4 * b
(2) - b(3)) / (2 * dx));
205         du2(Nx,1) = -(dt/dx) * (F(Nx+1,1) - F(Nx,1));
206         du2(Nx,2) = -(dt/dx) * (F(Nx+1,2) - F(Nx,2)) - dt * g * ((1/2) * (qr(Nx,1) + ql(Nx+1,1))) * ((3 *
b(Nx) - 4 * b(Nx-1) + b(Nx-2)) / (2 * dx));
207         for i = 2:Nx-1
208             du2(i,1) = -(dt/dx) * (F(i+1,1) - F(i,1));
209             du2(i,2) = -(dt/dx) * (F(i+1,2) - F(i,2)) - dt * g * ((1/2) * (qr(i,1) + ql(i+1,1))) * ((b(i
+1) - b(i-1)) / (2 * dx));
210         end
211         for i = 3:Nx-2
212             for j = 1:2
213                 qhelp(i,j) = q(i,j) + (1/4) * du1(i,j) + (1/4) * du2(i,j);
214             end
215         end
216     elseif stage == 3
217         du3(1,1) = -(dt/dx) * (F(2,1) - F(1,1));
218         du3(1,2) = -(dt/dx) * (F(2,2) - F(1,2)) - dt * g * ((1/2) * (qr(1,1) + ql(2,1))) * ((-3 * b(1) + 4 * b
(2) - b(3)) / (2 * dx));
219         du3(Nx,1) = -(dt/dx) * (F(Nx+1,1) - F(Nx,1));
220         du3(Nx,2) = -(dt/dx) * (F(Nx+1,2) - F(Nx,2)) - dt * g * ((1/2) * (qr(Nx,1) + ql(Nx+1,1))) * ((3 *
b(Nx) - 4 * b(Nx-1) + b(Nx-2)) / (2 * dx));
221         for i = 2:Nx-1
222             du3(i,1) = -(dt/dx) * (F(i+1,1) - F(i,1));
223             du3(i,2) = -(dt/dx) * (F(i+1,2) - F(i,2)) - dt * g * ((1/2) * (qr(i,1) + ql(i+1,1))) * ((b(i
+1) - b(i-1)) / (2 * dx));
224         end
225         for i = 3:Nx-2
226             for j = 1:2
227                 q(i,j) = q(i,j) + (1/6) * du1(i,j) + (1/6) * du2(i,j) + (4/6) * du3(i,j);
228             end
229         end
230     end
231 end
232 end
233
234 if k == 2
235     hold on
236 end
237
238 h(:,k) = q(:,1);
239 u(:,k) = q(:,2) ./ q(:,1);
240
241 uplot = q(:,1);
242 plot(xmid, uplot, '-o');
243 xlabel('x');
244 ylabel('free surface');
245 axis([xmin, xmax, -0.5, 0.5]);
246 %title(['state \ interpolation = ', num2str(state_inter), ', CFL = ', num2str(CFL), ', Nx = ',
num2str(Nx), ', Tmax = ', num2str(tmax), ', t = ', num2str(round(t,4))]);
247 grid on
248
249 end
250
251
252
253 %% Define parameters
254
255 Nx = 100; % number of finite volumes (= cells)
256 dx = (xmax - xmin) / Nx; % cell size
257
258 x = xmin:dx:xmax; % x-coordinates of the cell boundaries
259 xmid = (1/2) * (x(1:Nx) + x(2:Nx+1)); % x-coordinates of the cell centres
260
261 h_e = zeros(Nx,1);
262 u_e = zeros(Nx,1);
263
264 c_m = 1.205753246885354;
265 x_a = - tmax * sqrt(g*2);
266 x_b = t * (2 * sqrt(g*2) - 3 * c_m);

```

```

267 x_c = t * ( 2*(c_m^2)*( sqrt(g*2) - c_m ) / ( (c_m^2) - (g*1) );
268
269 for i=1:Nx
270     if xmid(i) <= x_a
271         h_e(i) = 2;
272         u_e(i) = 0;
273     elseif xmid(i) <= x_b
274         h_e(i) = ( 4 / (9 * g) ) * (( sqrt(g*2) - (xmid(i)/(2*tmax)) ) ^2);
275         u_e(i) = (2/3) * ((xmid(i)/tmax) + sqrt(g*2));
276     elseif xmid(i) <= x_c
277         h_e(i) = (c_m^2)/g;
278         u_e(i) = 2 * ( sqrt(g*2) - c_m );
279     else
280         h_e(i) = 1;
281         u_e(i) = 0;
282     end
283 end
284
285
286
287 plot(xmid,h_e,'-k');
288 xlabel('x');
289 %ylabel(' velocity u ');
290 ylabel(' free surface s ');
291 %axis([ xmin, xmax, -0.05, 0.5]);
292 axis([ xmin, xmax, 0.9, 2.1]);
293 %title([' state \_interpolation =', num2str(state_inter), ', CFL =', num2str(CFL), ', Nx =', num2str(
    Nx), ', Tmax =', num2str(tmax), ', t =', num2str(round(t,4))] );
294 grid on
295 hold off
296 legend({' First-order upwind ', 'Kappa = 1/3 ', 'Minmod limiter ', 'Superbee limiter ', 'Koren limiter
    ', '' }, 'Location ', 'northeast ', 'Orientation ', 'vertical ')
297
298
299
300 %% Begin plotting
301 %if mod(count, round(Nx^(1/2))) == 0
302 %     for i = 1:Nx
303 %         freude(i) = abs(q(i,2)/q(i,1))/sqrt(g*q(i,1));
304 %     end
305 %     surface = q(:,1) + b;
306 %     for i = 1:Nx
307 %         stationary(i) = q_left_boundary(2)/q_left_boundary(1);
308 %     end
309
310 %     hold off
311 %     plot(xmid, surface, '-o', xmid, freude, '-o', xmid, b, '-o', xmid, stationary, '-');
312 %     xlabel('x');
313 %     ylabel(' free surface ');
314 %     axis([ xmin, xmax, 0, 2]);
315 %     title([' state \_interpolation =', num2str(state_inter), ', flux \_evaluation =', num2str(
    flux_eval), ', CFL =', num2str(CFL), ', Nx =', num2str(Nx), ', Tmax =', num2str(tmax), ', t =',
    num2str(round(t,4))] );
316 %     hold on
317 %     grid on
318 %     drawnow
319 %end
320
321
322
323 %% Calculate total energy in the system
324 %energy(count) = 0;
325 %for i = 1:Nx
326 % energy(count) = energy(count) + ((1/2)*q(i,1)*((q(i,2)/q(i,1))^2)) + (g*q(i,1)*(q(i,1)
    +2*b(i))/2);
327 %end
328 %energy(count) = energy(count) / Nx;
329
330
331
332 %% Error calculation
333 %error(count) = 0;
334 %for i = 1:Nx

```

```
335 % error(count) = error(count) + abs(q(i,1) + b(i) - q_left_boundary(1));
336 %end
337 %error(count) = error(count)/Nx;
338
339
340
341
342
343 %tiledlayout(2,1)
344
345 %% Plot error calculation
346 %nexttile
347 %plot(time,error,'-');
348 %xlabel('time');
349 %ylabel('error vs stationary solution');
350
351
352
353 %% Plot energy calculation
354 %nexttile
355 %plot(time,energy,'-');
356 %xlabel('time');
357 %ylabel('energy');
358
359
360
361 %wave_speed
362 %energy = 0;
363 %for i = 1:Nx
364 % energy = energy + ((1/2)*((q(i,2)/q(i,1))^2)) + (g*(q(i,1)+2*b(i))/2);
365 %end
366 %energy = energy / Nx;
367 %energy
368
369
370
371 for k=1:5
372     disp(k);
373     L1_h = 0;
374     L2_h = 0;
375     L3_h = 0;
376     for i=1:Nx
377         L1_h = L1_h + abs(h(i,k)-h_e(i));
378         L2_h = L2_h + ((h(i,k)-h_e(i))^2);
379         if abs(h(i,k)-h_e(i)) > L3_h
380             L3_h = abs(h(i,k)-h_e(i));
381         end
382     end
383     disp(L1_h);
384     disp(L2_h);
385     disp(L3_h);
386
387     L1_u = 0;
388     L2_u = 0;
389     L3_u = 0;
390     for i=1:Nx
391         L1_u = L1_u + abs(u(i,k)-u_e(i));
392         L2_u = L2_u + ((u(i,k)-u_e(i))^2);
393         if abs(u(i,k)-u_e(i)) > L3_u
394             L3_u = abs(u(i,k)-u_e(i));
395         end
396     end
397     disp(L1_u);
398     disp(L2_u);
399     disp(L3_u);
400 end
```

B.5 Test case 2: Flux evaluation methods

The Matlab code below simulates multiple flux evaluation methods in test case 2.

```

1 clear all
2 format long
3
4
5
6 %% Select state interpolation and flux evaluation methods
7
8 % state_inter      type
9 % -----
10 % 0               first-order upwind
11 % 1               kappa = 1/3
12 % 2               with minmod limiter
13 % 3               with superbee limiter
14 % 4               with Koren limiter
15
16 % flux_eval       type
17 % -----
18 % 0               flux vector splitting
19 % 1               flux difference splitting: the midpoint rule
20 % 2               flux difference splitting: the trapezoidal rule
21 % 3               flux difference splitting: Roe's method (LeVeque 15.3.3 in Finite Volume
22 % 4               HLL Solver (Toro 10.4.1 with (10.18) in E.F. Toro – Shock-capturing methods
23 % 5               HLL Solver (LeVeque 15.3.7 in Finite Volume Methods for Hyperbolic Problems
24 %               )
25
26
27 %% Define parameters
28
29 Nx = 100; % number of finite volumes (= cells)
30 CFL = 0.9; % Courant FriedrichsLewy condition
31 tmax = 3; % time at which the simulation ends
32 g = 1; % gravitational acceleration
33
34 state_inter = 3;
35
36
37
38 %% Begin initial condition initialization
39
40 xmin = -8; % x-coordinate at the left boundary of the domain
41 xmax = 8; % x-coordinate at the right boundary of the domain
42 dx = ( xmax - xmin ) / Nx; % cell size
43
44 x = xmin:dx:xmax; % x-coordinates of the cell boundaries
45 xmid = (1/2)*( x(1:Nx) + x(2:Nx+1) ); % x-coordinates of the cell centres
46
47 h = zeros(Nx,6);
48 u = zeros(Nx,6);
49
50
51
52 %% Begin simulation
53
54 for k=1:6
55
56     b = zeros(Nx,1);
57     q = zeros(Nx,2);
58
59     for i=1:Nx
60         b(i) = 0; % b
61         if xmid(i) <= 0 % h
62             q(i,1) = 2 - b(i);
63         else
64             q(i,1) = 1 - b(i);
65         end
66         q(i,2) = 0; % hu

```



```

67 end
68
69 % set the boundary values
70 q_left_boundary = q(1,:);
71 q_right_boundary = q(Nx,:);
72 q_left_boundary(2) = 0;
73 q_right_boundary(2) = 0;
74
75 q1 = zeros(Nx+1,2);
76 qr = zeros(Nx+1,2);
77 F = zeros(Nx+1,2);
78
79 du1 = zeros(Nx,2);
80 du2 = zeros(Nx,2);
81 du3 = zeros(Nx,2);
82
83 t = 0;
84
85 while t < tmax
86
87     % calculate the timestep size
88     max_wave_speed = 0;
89     for i = 1:Nx
90         if abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1))) > max_wave_speed
91             max_wave_speed = abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1)));
92         end
93         if abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1))) > max_wave_speed
94             max_wave_speed = abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1)));
95         end
96     end
97     %if max_wave_speed > wave_speed
98     %    wave_speed = max_wave_speed;
99     %end
100     %max_wave_speed = 1.621310199408591;
101     dt = CFL * (dx / (2 * max_wave_speed) );
102
103     % if the simulation arrives at the final timestep
104     if t + dt - tmax > 0
105         dt = tmax - t;
106         break_this_loop = 1;
107     end
108
109     % adjust the time
110     t = t + dt;
111
112     %q_left_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
113     %q_right_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
114     %q_left_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
115     %q_right_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
116
117
118
119     qhelp = q; % used to calculate the values for h and hu in the intermediate RK3b steps
120
121     for stage = 1:3 % RK3b has three stages
122
123
124
125         %% Begin computation state interpolation
126
127         % note that q1(1,:) corresponds to q^1_{1/2} in the report and q1(Nx+1,:)
128         % corresponds to q^1_{M+1/2} in the report
129         for i = 1:Nx+1
130             if i==1
131                 q1(i,:) = q_left_boundary;
132             elseif i==2
133                 q1(i,:) = (1/2)*( qhelp(1,:) + qhelp(2,:) );
134             elseif i==Nx+1
135                 q1(i,:) = (3/2)*qhelp(Nx,:) - (1/2)*qhelp(Nx-1,:);
136             else
137                 for j=1:2
138                     q1(i,j) = qhelp(i-1,j) + (1/2) * phi_t( state_inter , qhelp(i,j) -
139 qhelp(i-1,j), qhelp(i-1,j) - qhelp(i-2,j) );

```

```

138         end
139     end
140 end
141
142 % note that qr(1,:) corresponds to q^r_{1/2} in the report and qr(Nx+1,:)
corresponds to q^r_{M+1/2} in the report
143 for i=1:Nx+1
144     if i==1
145         qr(i,:) = (3/2)*qhhelp(1,:) - (1/2)*qhhelp(2,:);
146     elseif i==Nx
147         qr(i,:) = (1/2)*( qhhelp(Nx-1,:) + qhhelp(Nx,:) );
148     elseif i==Nx+1
149         qr(i,:) = q_right_boundary;
150     else
151         for j=1:2
152             qr(i,j) = qhhelp(i,j) + (1/2) * phi_t( state_inter , qhhelp(i-1,j) -
qhhelp(i,j) , qhhelp(i,j) - qhhelp(i+1,j) );
153         end
154     end
155 end
156
157
158 %% Flux method
159
160 % Flux vector splitting
161 if k == 1
162     for i = 1:Nx+1
163         lambda1L = (q1(i,2)/q1(i,1)) - sqrt(g*q1(i,1)/2);
164         lambda2L = (q1(i,2)/q1(i,1)) + sqrt(g*q1(i,1)/2);
165         diagL = diag([max(lambda1L,0) ; max(lambda2L,0)]);
166         RL = [ 1, 1; lambda1L, lambda2L ];
167         R_invL = (1/sqrt(2*g*q1(i,1))) * [ lambda2L, -1; -lambda1L, 1];
168
169         lambda1R = (qr(i,2)/qr(i,1)) - sqrt(g*qr(i,1)/2);
170         lambda2R = (qr(i,2)/qr(i,1)) + sqrt(g*qr(i,1)/2);
171         diagR = diag([min(lambda1R,0) ; min(lambda2R,0)]);
172         RR = [ 1, 1; lambda1R, lambda2R ];
173         R_invR = (1/sqrt(2*g*qr(i,1))) * [ lambda2R, -1; -lambda1R, 1];
174
175         F(i,:) = RL*diagL*R_invL*q1(i,:).' + RR*diagR*R_invR*qr(i,:).';
176     end
177
178
179 % Flux difference splitting: The midpoint rule
180 elseif k == 2
181     for i = 1:Nx+1
182         qm = (1/2)*( q1(i,:) + qr(i,:) );
183         lambda1 = (qm(2)/qm(1)) - sqrt(g*qm(1));
184         lambda2 = (qm(2)/qm(1)) + sqrt(g*qm(1));
185         diag_matrix = diag([abs(lambda1) ; abs(lambda2)]);
186         R = [ 1, 1; lambda1, lambda2 ];
187         R_inv = (1/(2*sqrt(g*qm(1)))) * [ lambda2, -1; -lambda1, 1];
188
189         F(i,:) = ( (1/2)*([q1(i,2) ; (q1(i,2)/q1(i,1))*q1(i,2)+((g/2)*(q1(i,1)^2)
)] + [qr(i,2) ; (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2))] ) - (1/2)*R*diag_matrix*
R_inv*((qr(i,:)-q1(i,:)).') ).';
190     end
191
192 % Flux difference splitting: The trapezoidal rule
193 elseif k == 3
194     for i = 1:Nx+1
195         lambda1L = (q1(i,2)/q1(i,1)) - sqrt(g*q1(i,1));
196         lambda2L = (q1(i,2)/q1(i,1)) + sqrt(g*q1(i,1));
197         diagL = diag([abs(lambda1L) ; abs(lambda2L)]);
198         RL = [ 1, 1; lambda1L, lambda2L ];
199         R_invL = (1/sqrt(2*g*q1(i,1))) * [ lambda2L, -1; -lambda1L, 1];
200
201         lambda1R = (qr(i,2)/qr(i,1)) - sqrt(g*qr(i,1));
202         lambda2R = (qr(i,2)/qr(i,1)) + sqrt(g*qr(i,1));
203         diagR = diag([abs(lambda1R) ; abs(lambda2R)]);
204         RR = [ 1, 1; lambda1R, lambda2R ];
205         R_invR = (1/sqrt(2*g*qr(i,1))) * [ lambda2R, -1; -lambda1R, 1];
206

```

```

207         F(i,:) = ( (1/2)*([ql(i,2) ; (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2)
    ] + [qr(i,2) ; (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2))] ) - (1/4)*(RL*diagL*R_invL
    + RR*diagR*R_invR)*((qr(i,:)-ql(i,:)).') ).';
208     end
209
210     % Flux difference splitting: Roe's method
211     elseif k == 4
212         for i = 1:Nx+1
213             h_bar = (1/2)*(ql(i,1)+qr(i,1));
214             c_hat = sqrt(g*h_bar);
215             u_hat = ( (sqrt(ql(i,1))*ql(i,2)/ql(i,1))+sqrt(qr(i,1))*qr(i,2)/qr(i
    ,1))) / (sqrt(ql(i,1))+sqrt(qr(i,1)));
216             lambda1 = u_hat - c_hat;
217             lambda2 = u_hat + c_hat;
218
219             diag_matrix = diag([abs(lambda1) ; abs(lambda2)]);
220             R = [ 1, 1; lambda1, lambda2 ];
221             R_inv = (1/(2*c_hat)) * [ lambda2, -1; -lambda1, 1];
222
223             F(i,:) = ( (1/2)*([ql(i,2) ; (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2)
    ] + [qr(i,2) ; (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2))] ) - (1/2)*R*diag_matrix*
    R_inv*((qr(i,:)-ql(i,:)).') ).';
224     end
225
226     % The HLL Solver
227     elseif k == 5
228         for i = 1:Nx+1
229             h_hat = (1/g)*(( (1/2)*(sqrt(g*ql(i,1))+sqrt(g*qr(i,1))) + (1/4)*((ql(i
    ,2)/ql(i,1))-qr(i,2)/qr(i,1)) ) ^2);
230
231             if h_hat > ql(i,1)
232                 pl = sqrt( (1/2)*((h_hat*(h_hat+ql(i,1)))/(ql(i,1)^2)) );
233             else
234                 pl = 1;
235             end
236
237             if h_hat > qr(i,1)
238                 pr = sqrt( (1/2)*((h_hat*(h_hat+qr(i,1)))/(qr(i,1)^2)) );
239             else
240                 pr = 1;
241             end
242
243             sL = (ql(i,2)/ql(i,1)) - (sqrt(g*ql(i,1))*pl);
244             sR = (qr(i,2)/qr(i,1)) + (sqrt(g*qr(i,1))*pr);
245
246             if sL >= 0
247                 F(i,1) = ql(i,2);
248                 F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
249             elseif sR <= 0
250                 F(i,1) = qr(i,2);
251                 F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
252             else
253                 F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR
    - sL );
254                 F(i,2) = ( sR*((ql(i,2)/ql(i,1))*ql(i,2) + (1/2)*g*(ql(i,1)^2)) - sL
    *((qr(i,2)/qr(i,1))*qr(i,2) + (1/2)*g*(qr(i,1)^2)) + sR*sL*(qr(i,2)-ql(i,2)) ) / ( sR - sL
    );
255             end
256         end
257
258     % The HLL Solver
259     elseif k == 6
260         for i = 1:Nx+1
261             h_bar = (1/2)*(ql(i,1)+qr(i,1));
262             c_hat = sqrt(g*h_bar);
263             u_hat = ( (sqrt(ql(i,1))*ql(i,2)/ql(i,1))+sqrt(qr(i,1))*qr(i,2)/qr(i
    ,1))) / (sqrt(ql(i,1))+sqrt(qr(i,1)));
264             lambda1 = u_hat - c_hat;
265             lambda2 = u_hat + c_hat;
266
267             sL = min(min(lambda1, lambda2), min((ql(i,2)/ql(i,1))+sqrt(g*ql(i,1)), (ql
    (i,2)/ql(i,1))-sqrt(g*ql(i,1))));

```

```

268         sR = max(max(lambda1 , lambda2) , max((qr(i,2)/qr(i,1))+sqrt(g*qr(i,1)) , (qr
(i,2)/qr(i,1))-sqrt(g*qr(i,1)))));
269
270         if sL >= 0
271             F(i,1) = ql(i,2);
272             F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
273         elseif sR <= 0
274             F(i,1) = qr(i,2);
275             F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
276         else
277             F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR
- sL );
278             F(i,2) = ( sR*((ql(i,2)/ql(i,1))*ql(i,2) + (1/2)*g*(ql(i,1)^2)) - sL
*((qr(i,2)/qr(i,1))*qr(i,2) + (1/2)*g*(qr(i,1)^2)) + sR*sL*(qr(i,2)-ql(i,2)) ) / ( sR - sL
);
279         end
280     end
281 end
282
283
284
285 %% Perform time-integration method
286
287 if stage == 1
288     du1(1,1)=- (dt/dx)*(F(2,1)-F(1,1));
289     du1(1,2)=- (dt/dx)*(F(2,2)-F(1,2))-dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
(2)-b(3))/(2*dx));
290     du1(Nx,1)=- (dt/dx)*(F(Nx+1,1)-F(Nx,1));
291     du1(Nx,2)=- (dt/dx)*(F(Nx+1,2)-F(Nx,2))-dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
292     for i=2:Nx-1
293         du1(i,1)=- (dt/dx)*(F(i+1,1)-F(i,1));
294         du1(i,2)=- (dt/dx)*(F(i+1,2)-F(i,2))-dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
+1)-b(i-1))/(2*dx));
295     end
296     for i=3:Nx-2
297         for j=1:2
298             qhelp(i,j)=q(i,j)+du1(i,j);
299         end
300     end
301 elseif stage == 2
302     du2(1,1)=- (dt/dx)*(F(2,1)-F(1,1));
303     du2(1,2)=- (dt/dx)*(F(2,2)-F(1,2))-dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
(2)-b(3))/(2*dx));
304     du2(Nx,1)=- (dt/dx)*(F(Nx+1,1)-F(Nx,1));
305     du2(Nx,2)=- (dt/dx)*(F(Nx+1,2)-F(Nx,2))-dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
306     for i=2:Nx-1
307         du2(i,1)=- (dt/dx)*(F(i+1,1)-F(i,1));
308         du2(i,2)=- (dt/dx)*(F(i+1,2)-F(i,2))-dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
+1)-b(i-1))/(2*dx));
309     end
310     for i=3:Nx-2
311         for j=1:2
312             qhelp(i,j)=q(i,j)+(1/4)*du1(i,j)+(1/4)*du2(i,j);
313         end
314     end
315 elseif stage == 3
316     du3(1,1)=- (dt/dx)*(F(2,1)-F(1,1));
317     du3(1,2)=- (dt/dx)*(F(2,2)-F(1,2))-dt*g*((1/2)*(qr(1,1)+ql(2,1)))*((-3*b(1)+4*b
(2)-b(3))/(2*dx));
318     du3(Nx,1)=- (dt/dx)*(F(Nx+1,1)-F(Nx,1));
319     du3(Nx,2)=- (dt/dx)*(F(Nx+1,2)-F(Nx,2))-dt*g*((1/2)*(qr(Nx,1)+ql(Nx+1,1)))*((3*
b(Nx)-4*b(Nx-1)+b(Nx-2))/(2*dx));
320     for i=2:Nx-1
321         du3(i,1)=- (dt/dx)*(F(i+1,1)-F(i,1));
322         du3(i,2)=- (dt/dx)*(F(i+1,2)-F(i,2))-dt*g*((1/2)*(qr(i,1)+ql(i+1,1)))*((b(i
+1)-b(i-1))/(2*dx));
323     end
324     for i=3:Nx-2
325         for j=1:2
326             q(i,j)=q(i,j)+(1/6)*du1(i,j)+(1/6)*du2(i,j)+(4/6)*du3(i,j);
327         end

```

```

328         end
329     end
330 end
331 end
332
333 if k == 2
334     hold on
335 end
336
337 h(:,k) = q(:,1);
338 u(:,k) = q(:,2) ./ q(:,1);
339
340 plot(xmid,q(:,1),'-o');
341 xlabel('x');
342 ylabel('velocity u');
343 axis([xmin,xmax,-0.5,0.5]);
344 %title(['state \ interpolation =',num2str(state_inter),', CFL=',num2str(CFL),', Nx=',
num2str(Nx),', Tmax=',num2str(tmax),', t=',num2str(round(t,4))]);
345 grid on
346
347 end
348
349 %% Define parameters
350
351 Nx = 100; % number of finite volumes (= cells)
352 dx = ( xmax - xmin ) / Nx; % cell size
353
354 x = xmin:dx:xmax; % x-coordinates of the cell boundaries
355 xmid = (1/2)*( x(1:Nx) + x(2:Nx+1) ); % x-coordinates of the cell centres
356
357 h_e = zeros(Nx,1);
358 u_e = zeros(Nx,1);
359
360 c_m = 1.205753246885354;
361 x_a = - tmax * sqrt(g*2);
362 x_b = t * ( 2 * sqrt(g*2) - 3 * c_m );
363 x_c = t * ( 2*(c_m^2)*( sqrt(g*2) - c_m ) / ( (c_m^2) - (g*1) ) );
364
365 for i=1:Nx
366     if xmid(i) <= x_a
367         h_e(i) = 2;
368         u_e(i) = 0;
369     elseif xmid(i) <= x_b
370         h_e(i) = ( 4 / (9 * g) ) * (( sqrt(g*2) - (xmid(i)/(2*tmax)) )^2);
371         u_e(i) = (2/3) * ((xmid(i)/tmax) + sqrt(g*2));
372     elseif xmid(i) <= x_c
373         h_e(i) = (c_m^2)/g;
374         u_e(i) = 2 * ( sqrt(g*2) - c_m );
375     else
376         h_e(i) = 1;
377         u_e(i) = 0;
378     end
379 end
380
381
382
383 plot(xmid,h_e,'-k');
384 xlabel('x');
385 ylabel('velocity u');
386 ylabel('free surface s');
387 axis([xmin,xmax,-0.05,0.5]);
388 axis([xmin,xmax,0.9,2.1]);
389 %title(['state \ interpolation =',num2str(state_inter),', CFL=',num2str(CFL),', Nx=',num2str(
Nx),', Tmax=',num2str(tmax),', t=',num2str(round(t,4))]);
390 grid on
391 hold off
392 legend({'Flux vector splitting','Midpoint rule','Trapezoidal rule','Roe''s method','HLL solver',
'HLLC solver'},'Location','northeast','Orientation','vertical')
393
394
395
396 %% Begin plotting
397 %if mod(count,round(Nx^(1/2))) == 0

```

```

398 % for i = 1:Nx
399 %     freude(i) = abs(q(i,2)/q(i,1))/sqrt(g*q(i,1));
400 % end
401 % surface = q(:,1) + b;
402 % for i = 1:Nx
403 %     stationary(i) = q_left_boundary(2)/q_left_boundary(1);
404 % end
405
406 % hold off
407 % plot(xmid, surface, '-o', xmid, freude, '-o', xmid, b, '-o', xmid, stationary, '-');
408 % xlabel('x');
409 % ylabel('free surface');
410 % axis([xmin, xmax, 0, 2]);
411 % title(['state \ interpolation =', num2str(state_inter), ', flux \ evaluation =', num2str(
412 %     flux_eval), ', CFL =', num2str(CFL), ', Nx =', num2str(Nx), ', Tmax =', num2str(tmax), ', t =',
413 %     num2str(round(t,4))] );
414 % hold on
415 % grid on
416 % drawnow
417 %end
418
419 %% Calculate total energy in the system
420 %energy(count) = 0;
421 %for i = 1:Nx
422 % energy(count) = energy(count) + ((1/2)*q(i,1)*((q(i,2)/q(i,1))^2)) + (g*q(i,1)*(q(i,1)
423 % +2*b(i))/2);
424 %end
425 %energy(count) = energy(count) / Nx;
426
427
428 %% Error calculation
429 %error(count) = 0;
430 %for i = 1:Nx
431 % error(count) = error(count) + abs(q(i,1) + b(i) - q_left_boundary(1));
432 %end
433 %error(count) = error(count)/Nx;
434
435
436
437
438
439 %tiledlayout(2,1)
440
441 %% Plot error calculation
442 %nexttile
443 %plot(time, error, '-');
444 %xlabel('time');
445 %ylabel('error vs stationary solution');
446
447
448
449 %% Plot energy calculation
450 %nexttile
451 %plot(time, energy, '-');
452 %xlabel('time');
453 %ylabel('energy');
454
455
456
457 %wave_speed
458 %energy = 0;
459 %for i = 1:Nx
460 % energy = energy + ((1/2)*((q(i,2)/q(i,1))^2)) + (g*(q(i,1)+2*b(i))/2);
461 %end
462 %energy = energy / Nx;
463 %energy
464 %wave_speed
465
466
467

```

```
468 for k=1:6
469     disp(k);
470     L1_h = 0;
471     L2_h = 0;
472     L3_h = 0;
473     for i=1:Nx
474         L1_h = L1_h + abs(h(i,k)-h_e(i));
475         L2_h = L2_h + ((h(i,k)-h_e(i))^2);
476         if abs(h(i,k)-h_e(i)) > L3_h
477             L3_h = abs(h(i,k)-h_e(i));
478         end
479     end
480     disp(L1_h);
481     disp(L2_h);
482     disp(L3_h);
483
484     L1_u = 0;
485     L2_u = 0;
486     L3_u = 0;
487     for i=1:Nx
488         L1_u = L1_u + abs(u(i,k)-u_e(i));
489         L2_u = L2_u + ((u(i,k)-u_e(i))^2);
490         if abs(u(i,k)-u_e(i)) > L3_u
491             L3_u = abs(u(i,k)-u_e(i));
492         end
493     end
494     disp(L1_u);
495     disp(L2_u);
496     disp(L3_u);
497 end
```

B.6 Test case 3: Flux vector splitting

The Matlab code below simulates the flux vector splitting method in test case 3.

```
1 clear all
2 format long
3
4 %% Select conditions
5
6 state_inter = 4;
7
8
9
10 %% Define parameters
11 Nx = 100;
12 xmin = -10;
13 xmax = 10;
14 dx = (xmax-xmin)/Nx;
15 x = xmin:dx:xmax;
16 xmid = (1/2)*(x(1:Nx)+x(2:Nx+1));
17
18 Ny = 100;
19 ymin = -10;
20 ymax = 10;
21 dy = (ymax-ymin)/Ny;
22 y = ymin:dy:ymax;
23 ymid = (1/2)*(y(1:Ny)+y(2:Ny+1));
24
25 CFL = 0.9;
26 tmax = 15;
27
28 g = 1;
29
30 break_this_loop = 0;
31 count = 0;
32
33
34
35 %% Declare sizes of matrices and vectors to lower running time
36 F = zeros(Nx+1,Ny,3);
37 G = zeros(Nx,Ny+1,3);
38
39 FL = zeros(Nx+1,Ny,3);
40 FR = zeros(Nx+1,Ny,3);
41 GL = zeros(Nx,Ny+1,3);
42 GR = zeros(Nx,Ny+1,3);
43
44 r = zeros(3,1);
45 q_initial = zeros(3,1);
46
47 du1 = zeros(Nx,Ny,3);
48 du2 = zeros(Nx,Ny,3);
49 du3 = zeros(Nx,Ny,3);
50 u = zeros(Nx,Ny,3);
51 b = zeros(Nx,Ny,1);
52 b_num = zeros(Nx,Ny,2);
53
54
55
56 %% Begin initial condition initialization
57 for i=1:Nx
58     for j=1:Ny
59         b(i,j) = (1/5)*exp(-(xmid(i)^2)-(ymid(j)^2));
60     end
61 end
62
63 for i=2:Nx-1
64     b_num(i,1,1) = ((b(i+1,1)-b(i-1,1))/(2*dx));
65     b_num(i,1,2) = ((-3*b(i,1)+4*b(i,2)-b(i,3))/(2*dy));
66     b_num(i,Ny,1) = ((b(i+1,Ny)-b(i-1,Ny))/(2*dx));
67     b_num(i,Ny,2) = ((3*b(i,Ny)-4*b(i,Ny-1)+b(i,Ny-2))/(2*dy));
68 end
69 for j=2:Ny-1
```



```

70     b_num(1,j,1) = ((-3*b(1,j)+4*b(2,j)-b(3,j))/(2*dx));
71     b_num(1,j,2) = ((b(1,j+1)-b(1,j-1))/(2*dy));
72     b_num(Nx,j,1) = ((3*b(Nx,j)-4*b(Nx-1,j)+b(Nx-2,j))/(2*dx));
73     b_num(Nx,j,2) = ((b(Nx,j+1)-b(Nx,j-1))/(2*dy));
74 end
75 b_num(1,1,1) = ((-3*b(1,1)+4*b(2,1)-b(3,1))/(2*dx));
76 b_num(1,1,2) = ((-3*b(1,1)+4*b(1,2)-b(1,3))/(2*dy));
77 b_num(1,Ny,1) = ((-3*b(1,Ny)+4*b(2,Ny)-b(3,Ny))/(2*dx));
78 b_num(1,Ny,2) = ((3*b(1,Ny)-4*b(1,Ny-1)+b(1,Ny-2))/(2*dy));
79 snapnow;
80 b_num(Nx,1,1) = ((3*b(Nx,1)-4*b(Nx-1,1)+b(Nx-2,1))/(2*dx));
81 b_num(Nx,1,2) = ((-3*b(Nx,1)+4*b(Nx,2)-b(Nx,3))/(2*dy));
82 b_num(Nx,Ny,1) = ((3*b(Nx,Ny)-4*b(Nx-1,Ny)+b(Nx-2,Ny))/(2*dx));
83 b_num(Nx,Ny,2) = ((3*b(Nx,Ny)-4*b(Nx,Ny-1)+b(Nx,Ny-2))/(2*dy));
84 for i=2:Nx-1
85     for j=2:Ny-1
86         b_num(i,j,1) = ((b(i+1,j)-b(i-1,j))/(2*dx));
87         b_num(i,j,2) = ((b(i,j+1)-b(i,j-1))/(2*dy));
88     end
89 end
90
91 for i=1:Nx
92     for j=1:Ny
93         %if (xmid(i)^2)+(ymid(j)^2) < 10
94             % u(i,j,1) = 1 - b(i,j);
95         %else
96             % u(i,j,1) = 1 - b(i,j);
97         %end
98         u(i,j,1) = 1 - b(i,j);
99         u(i,j,2) = 2*u(i,j,1);
100        u(i,j,3) = 0;
101    end
102 end
103
104 q_initial(1) = 1;
105 q_initial(2) = 2;
106 q_initial(3) = 0;
107
108
109
110 %% Begin time stepping
111
112 t=0;
113
114 while t<tmax
115
116     count = count + 1;
117
118     % Adjust time
119     max_wave_speed = 0;
120     for i = 1:Nx
121         for j = 1:Ny
122             if abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
123                 max_wave_speed = abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1)));
124             end
125             if abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
126                 max_wave_speed = abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1)));
127             end
128             if abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
129                 max_wave_speed = abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1)));
130             end
131             if abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
132                 max_wave_speed = abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1)));
133             end
134         end
135     end
136     dt = CFL * (min(dx,dy) / (2 * max_wave_speed));
137
138     t = t+dt;
139
140     if t - tmax > 0
141         t = t-dt;
142         dt = tmax - t;

```

```

143     t = t+dt;
144     break_this_loop = 1;
145 end
146
147
148
149 %% Calculate timestep
150
151 uhelp=u;
152
153 for istage=1:3
154
155     %% Begin computation net cell fluxes
156     for i=1:Nx+1
157         for j=1:Ny
158             if i==1
159                 FL(1,j,1)=q_initial(1);
160                 FL(1,j,2)=q_initial(2);
161                 FL(1,j,3)=q_initial(3);
162             elseif i==2
163                 FL(2,j,1)=(1/2)*(uhelp(1,j,1)+uhelp(2,j,1));
164                 FL(2,j,2)=(1/2)*(uhelp(1,j,2)+uhelp(2,j,2));
165                 FL(2,j,3)=(1/2)*(uhelp(1,j,3)+uhelp(2,j,3));
166             elseif i==Nx+1
167                 FL(Nx+1,j,1)=(3/2)*uhelp(Nx,j,1)-(1/2)*uhelp(Nx-1,j,1);
168                 FL(Nx+1,j,2)=(3/2)*uhelp(Nx,j,2)-(1/2)*uhelp(Nx-1,j,2);
169                 FL(Nx+1,j,3)=(3/2)*uhelp(Nx,j,3)-(1/2)*uhelp(Nx-1,j,3);
170             else
171                 for k=1:3
172                     FL(i,j,k) = uhelp(i-1,j,k) + (1/2) * phi_t( state_inter , uhelp(i,j,k)-
uhelp(i-1,j,k) , uhelp(i-1,j,k)-uhelp(i-2,j,k) );
173                 end
174             end
175         end
176     end
177
178     for i=1:Nx
179         for j=1:Ny+1
180             if j==1
181                 GL(i,1,1)=q_initial(1);
182                 GL(i,1,2)=q_initial(2);
183                 GL(i,1,3)=q_initial(3);
184             elseif j==2
185                 GL(i,2,1)=(1/2)*(uhelp(i,1,1)+uhelp(i,2,1));
186                 GL(i,2,2)=(1/2)*(uhelp(i,1,2)+uhelp(i,2,2));
187                 GL(i,2,3)=(1/2)*(uhelp(i,1,3)+uhelp(i,2,3));
188             elseif j==Ny+1
189                 GL(i,Ny+1,1)=(3/2)*uhelp(i,Ny,1)-(1/2)*uhelp(i,Ny-1,1);
190                 GL(i,Ny+1,2)=(3/2)*uhelp(i,Ny,2)-(1/2)*uhelp(i,Ny-1,2);
191                 GL(i,Ny+1,3)=(3/2)*uhelp(i,Ny,3)-(1/2)*uhelp(i,Ny-1,3);
192             else
193                 for k=1:3
194                     GL(i,j,k) = uhelp(i,j-1,k) + (1/2) * phi_t( state_inter , uhelp(i,j,k)-
uhelp(i,j-1,k) , uhelp(i,j-1,k)-uhelp(i,j-2,k) );
195                 end
196             end
197         end
198     end
199
200     for i=1:Nx+1
201         for j=1:Ny
202             if i==1
203                 FR(1,j,1)=(3/2)*uhelp(1,j,1)-(1/2)*uhelp(2,j,1);
204                 FR(1,j,2)=(3/2)*uhelp(1,j,2)-(1/2)*uhelp(2,j,2);
205                 FR(1,j,3)=(3/2)*uhelp(1,j,3)-(1/2)*uhelp(2,j,3);
206             elseif i==Nx
207                 FR(Nx,j,1)=(1/2)*(uhelp(Nx-1,j,1)+uhelp(Nx,j,1));
208                 FR(Nx,j,2)=(1/2)*(uhelp(Nx-1,j,2)+uhelp(Nx,j,2));
209                 FR(Nx,j,3)=(1/2)*(uhelp(Nx-1,j,3)+uhelp(Nx,j,3));
210             elseif i==Nx+1
211                 FR(Nx+1,j,1)=q_initial(1);
212                 FR(Nx+1,j,2)=q_initial(2);
213                 FR(Nx+1,j,3)=q_initial(3);

```

```

214         else
215             for k=1:3
216                 FR(i,j,k) = uhelp(i,j,k) + (1/2) * phi_t( state_inter , uhelp(i-1,j,k)-
uhelp(i,j,k) , uhelp(i,j,k)-uhelp(i+1,j,k) );
217                 end
218             end
219         end
220     end
221
222     for i=1:Nx
223         for j=1:Ny+1
224             if j==1
225                 GR(i,1,1)=(3/2)*uhelp(i,1,1)-(1/2)*uhelp(i,2,1);
226                 GR(i,1,2)=(3/2)*uhelp(i,1,2)-(1/2)*uhelp(i,2,2);
227                 GR(i,1,3)=(3/2)*uhelp(i,1,3)-(1/2)*uhelp(i,2,3);
228             elseif j==Ny
229                 GR(i,Ny,1)=(1/2)*(uhelp(i,Ny-1,1)+uhelp(i,Ny,1));
230                 GR(i,Ny,2)=(1/2)*(uhelp(i,Ny-1,2)+uhelp(i,Ny,2));
231                 GR(i,Ny,3)=(1/2)*(uhelp(i,Ny-1,3)+uhelp(i,Ny,3));
232             elseif j==Ny+1
233                 GR(i,Ny+1,1)=q_initial(1);
234                 GR(i,Ny+1,2)=q_initial(2);
235                 GR(i,Ny+1,3)=q_initial(3);
236             else
237                 for k=1:3
238                     GR(i,j,k) = uhelp(i,j,k) + (1/2) * phi_t( state_inter , uhelp(i,j-1,k)-
uhelp(i,j,k) , uhelp(i,j,k)-uhelp(i,j+1,k) );
239                     end
240                 end
241             end
242         end
243
244
245
246     %% Flux method: Flux vector splitting
247     for i=1:Nx+1
248         for j=1:Ny
249             lambda1L = (FL(i,j,2)/FL(i,j,1)) - sqrt(g*FL(i,j,1)/2);
250             lambda2L = (FL(i,j,2)/FL(i,j,1));
251             lambda3L = (FL(i,j,2)/FL(i,j,1)) + sqrt(g*FL(i,j,1)/2);
252             diagL = diag([max(lambda1L,0) ; max(lambda2L,0) ; max(lambda3L,0)]);
253             RL = [1 0 1; lambda1L 0 lambda3L; FL(i,j,3)/FL(i,j,1) 1 FL(i,j,3)/FL(i,j,1) ];
254
255             lambda1R = (FR(i,j,2)/(FR(i,j,1)+eps)) - sqrt(g*FR(i,j,1)/2);
256             lambda2R = (FR(i,j,2)/(FR(i,j,1)+eps));
257             lambda3R = (FR(i,j,2)/(FR(i,j,1)+eps)) + sqrt(g*FR(i,j,1)/2);
258             diagR = diag([min(lambda1R,0) ; min(lambda2R,0) ; min(lambda3R,0)]);
259             RR = [1 0 1; lambda1R 0 lambda3R; FR(i,j,3)/FR(i,j,1) 1 FR(i,j,3)/FR(i,j,1) ];
260
261             F(i,j,:) = RL*diagL*(RL\[FL(i,j,1); FL(i,j,2); FL(i,j,3)]) + RR*diagR*(RR\[FR(
i,j,1); FR(i,j,2); FR(i,j,3)]);
262         end
263     end
264
265     for i=1:Nx
266         for j=1:Ny+1
267             lambda1L = (GL(i,j,3)/GL(i,j,1)) - sqrt(g*GL(i,j,1)/2);
268             lambda2L = (GL(i,j,3)/GL(i,j,1));
269             lambda3L = (GL(i,j,3)/GL(i,j,1)) + sqrt(g*GL(i,j,1)/2);
270             diagL = diag([max(lambda1L,0) ; max(lambda2L,0) ; max(lambda3L,0)]);
271             RL = [1, 0, 1; GL(i,j,2)/GL(i,j,1), 1, GL(i,j,2)/GL(i,j,1); lambda1L, 0,
lambda3L ];
272
273             lambda1R = (GR(i,j,3)/GR(i,j,1)) - sqrt(g*GR(i,j,1)/2);
274             lambda2R = (GR(i,j,3)/GR(i,j,1));
275             lambda3R = (GR(i,j,3)/GR(i,j,1)) + sqrt(g*GR(i,j,1)/2);
276             diagR = diag([min(lambda1R,0) ; min(lambda2R,0) ; min(lambda3R,0)]);
277             RR = [1, 0, 1; GR(i,j,2)/GR(i,j,1), 1, GR(i,j,2)/GR(i,j,1); lambda1R, 0,
lambda3R ];
278
279             G(i,j,:) = RL*diagL*(RL\[GL(i,j,1); GL(i,j,2); GL(i,j,3)]) + RR*diagR*(RR\[GR(
i,j,1); GR(i,j,2); GR(i,j,3)]);
280         end

```

```

281     end
282
283
284
285     %% Perform time-integration method
286
287     if istage == 1
288         for i=1:Nx
289             for j=1:Ny
290                 du1(i,j,1)=- (dt/dx)*(F(i+1,j,1)-F(i,j,1))-(dt/dy)*(G(i,j+1,1)-G(i,j,1));
291                 du1(i,j,2)=- (dt/dx)*(F(i+1,j,2)-F(i,j,2))-(dt/dy)*(G(i,j+1,2)-G(i,j,2))-dt
292 *g*(1/2)*(FR(i,j,1)+FL(i+1,j,1))*b_num(i,j,1);
293                 du1(i,j,3)=- (dt/dx)*(F(i+1,j,3)-F(i,j,3))-(dt/dy)*(G(i,j+1,3)-G(i,j,3))-dt
294 *g*(1/2)*(GR(i,j,1)+GL(i,j+1,1))*b_num(i,j,2);
295                 for k=1:3
296                     uhelp(i,j,k)=u(i,j,k)+du1(i,j,k);
297                 end
298             end
299         end
300     elseif istage == 2
301         for i=1:Nx
302             for j=1:Ny
303                 du2(i,j,1)=- (dt/dx)*(F(i+1,j,1)-F(i,j,1))-(dt/dy)*(G(i,j+1,1)-G(i,j,1));
304                 du2(i,j,2)=- (dt/dx)*(F(i+1,j,2)-F(i,j,2))-(dt/dy)*(G(i,j+1,2)-G(i,j,2))-dt
305 *g*(1/2)*(FR(i,j,1)+FL(i+1,j,1))*b_num(i,j,1);
306                 du2(i,j,3)=- (dt/dx)*(F(i+1,j,3)-F(i,j,3))-(dt/dy)*(G(i,j+1,3)-G(i,j,3))-dt
307 *g*(1/2)*(GR(i,j,1)+GL(i,j+1,1))*b_num(i,j,2);
308                 for k=1:3
309                     uhelp(i,j,k)=u(i,j,k)+(1/4)*du1(i,j,k)+(1/4)*du2(i,j,k);
310                 end
311             end
312         end
313     elseif istage == 3
314         for i=1:Nx
315             for j=1:Ny
316                 du3(i,j,1)=- (dt/dx)*(F(i+1,j,1)-F(i,j,1))-(dt/dy)*(G(i,j+1,1)-G(i,j,1));
317                 du3(i,j,2)=- (dt/dx)*(F(i+1,j,2)-F(i,j,2))-(dt/dy)*(G(i,j+1,2)-G(i,j,2))-dt
318 *g*(1/2)*(FR(i,j,1)+FL(i+1,j,1))*b_num(i,j,1);
319                 du3(i,j,3)=- (dt/dx)*(F(i+1,j,3)-F(i,j,3))-(dt/dy)*(G(i,j+1,3)-G(i,j,3))-dt
320 *g*(1/2)*(GR(i,j,1)+GL(i,j+1,1))*b_num(i,j,2);
321                 for k=1:3
322                     u(i,j,k)=u(i,j,k)+(1/6)*du1(i,j,k)+(1/6)*du2(i,j,k)+(4/6)*du3(i,j,k);
323                 end
324             end
325         end
326     end
327
328     %% Begin plotting
329     if mod(count,10) == 0
330         set(gcf,'Position',[100 100 600 500])
331         hplot = u(:,:,1) + b(:,:,);
332
333         %surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
334         surf(xmid,ymid,transpose(u(:,:,2)./u(:,:,1)),'EdgeColor','none','LineStyle','none');
335         %hold on
336         %surf(xmid,ymid,transpose(b(:,:)));
337         %view(2)
338         %hold off
339         xlabel('x');
340         ylabel('y');
341         zlabel('free surface s');
342         xlim([xmin xmax]);
343         ylim([ymin ymax]);
344         zlim([0.75 1.25]);
345         zlim([1.75 2.25]);
346         h = colorbar;
347         h.Limits = [0.75 1.25];
348         title(['Nx=',num2str(Nx),', Ny=',num2str(Ny),', Tmax=',num2str(tmax),', t=',num2str
349 (round(t,4))] );
350         drawnow

```

```

347
348 % angle measurement
349 j1 = 1;
350 j2 = 1;
351 for j = round(Ny/2):Ny
352     if u(Nx,j,1) > u(Nx,j1,1)
353         j1 = j;
354     end
355 end
356 for j = round(Ny/2):Ny
357     if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)
358         j2 = j;
359     end
360 end
361 j3 = 1;
362 j4 = 1;
363 for j = round(Ny/2):Ny
364     if u(Nx,j,1) < u(Nx,j3,1)
365         j3 = j;
366     end
367 end
368 for j = round(Ny/2):Ny
369     if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
370         j4 = j;
371     end
372 end
373 disp('current time is ')
374 disp(t)
375 disp('max')
376 disp(atan4(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
377 disp(u(Nx,j1,1))
378 disp(u(round(11*Nx/20),j2,1))
379 disp('min')
380 disp(atan4(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
381 disp(u(Nx,j3,1))
382 disp(u(round(11*Nx/20),j4,1))
383
384 end
385 end
386
387
388
389 %% Begin plotting
390 set(gcf,'Position',[100 100 600 500])
391 hplot = u(:,:,1) + b(:,:,1);
392
393 surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
394 hold on
395 %surf(xmid,ymid,transpose(b(:,:)));
396 %view(2)
397 %hold off
398 xlabel('x');
399 ylabel('y');
400 zlabel('free surface s');
401 xlim([xmin xmax]);
402 ylim([ymin ymax]);
403 zlim([0.75 1.25]);
404 h = colorbar;
405 h.Limits = [0.75 1.25];
406 title(['Nx=',num2str(Nx),', Ny=',num2str(Ny),', Tmax=',num2str(tmax),', t=',num2str
(round(t,4)]]);
407 drawnow
408
409 % angle measurement
410 j1 = 1;
411 j2 = 1;
412 for j = round(Ny/2):Ny
413     if u(Nx,j,1) > u(Nx,j1,1)
414         j1 = j;
415     end
416 end
417 for j = round(Ny/2):Ny
418     if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)

```

```

419         j2 = j;
420     end
421 end
422 j3 = 1;
423 j4 = 1;
424 for j = round(Ny/2):Ny
425     if u(Nx,j,1) < u(Nx,j3,1)
426         j3 = j;
427     end
428 end
429 for j = round(Ny/2):Ny
430     if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
431         j4 = j;
432     end
433 end
434 disp('current time is ')
435 disp(t)
436 disp('max')
437 disp(atanh(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
438 disp(u(Nx,j1,1))
439 disp(u(round(11*Nx/20),j2,1))
440 disp('min')
441 disp(atanh(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
442 disp(u(Nx,j3,1))
443 disp(u(round(11*Nx/20),j4,1))

```

B.7 Test case 3: Trapezoidal rule

The Matlab code below simulates the trapezoidal rule method in test case 3.

```
1 clear all
2 format long
3
4 %% Select conditions
5
6 state_inter = 4;
7
8
9
10 %% Define parameters
11 Nx = 100;
12 xmin = -10;
13 xmax = 10;
14 dx = (xmax-xmin)/Nx;
15 x = xmin:dx:xmax;
16 xmid = (1/2)*(x(1:Nx)+x(2:Nx+1));
17
18 Ny = 100;
19 ymin = -10;
20 ymax = 10;
21 dy = (ymax-ymin)/Ny;
22 y = ymin:dy:ymax;
23 ymid = (1/2)*(y(1:Ny)+y(2:Ny+1));
24
25 CFL = 0.9;
26 tmax = 15;
27
28 g = 1;
29
30 break_this_loop = 0;
31 count = 0;
32
33
34
35 %% Declare sizes of matrices and vectors to lower running time
36 F = zeros(Nx+1,Ny,3);
37 G = zeros(Nx,Ny+1,3);
38
39 FL = zeros(Nx+1,Ny,3);
40 FR = zeros(Nx+1,Ny,3);
41 GL = zeros(Nx,Ny+1,3);
42 GR = zeros(Nx,Ny+1,3);
43
44 r = zeros(3,1);
45 q_initial = zeros(3,1);
46
47 du1 = zeros(Nx,Ny,3);
48 du2 = zeros(Nx,Ny,3);
49 du3 = zeros(Nx,Ny,3);
50 u = zeros(Nx,Ny,3);
51 b = zeros(Nx,Ny,1);
52 b_num = zeros(Nx,Ny,2);
53
54
55
56 %% Begin initial condition initialization
57 for i=1:Nx
58     for j=1:Ny
59         b(i,j) = (1/5)*exp(-(xmid(i)^2)-(ymid(j)^2));
60     end
61 end
62
63 for i=2:Nx-1
64     b_num(i,1,1) = ((b(i+1,1)-b(i-1,1))/(2*dx));
65     b_num(i,1,2) = ((-3*b(i,1)+4*b(i,2)-b(i,3))/(2*dy));
66     b_num(i,Ny,1) = ((b(i+1,Ny)-b(i-1,Ny))/(2*dx));
67     b_num(i,Ny,2) = ((3*b(i,Ny)-4*b(i,Ny-1)+b(i,Ny-2))/(2*dy));
68 end
69 for j=2:Ny-1
```

```

70     b_num(1,j,1) = ((-3*b(1,j)+4*b(2,j)-b(3,j))/(2*dx));
71     b_num(1,j,2) = ((b(1,j+1)-b(1,j-1))/(2*dy));
72     b_num(Nx,j,1) = ((3*b(Nx,j)-4*b(Nx-1,j)+b(Nx-2,j))/(2*dx));
73     b_num(Nx,j,2) = ((b(Nx,j+1)-b(Nx,j-1))/(2*dy));
74 end
75 b_num(1,1,1) = ((-3*b(1,1)+4*b(2,1)-b(3,1))/(2*dx));
76 b_num(1,1,2) = ((-3*b(1,1)+4*b(1,2)-b(1,3))/(2*dy));
77 b_num(1,Ny,1) = ((-3*b(1,Ny)+4*b(2,Ny)-b(3,Ny))/(2*dx));
78 b_num(1,Ny,2) = ((3*b(1,Ny)-4*b(1,Ny-1)+b(1,Ny-2))/(2*dy));
79 snapnow;
80 b_num(Nx,1,1) = ((3*b(Nx,1)-4*b(Nx-1,1)+b(Nx-2,1))/(2*dx));
81 b_num(Nx,1,2) = ((-3*b(Nx,1)+4*b(Nx,2)-b(Nx,3))/(2*dy));
82 b_num(Nx,Ny,1) = ((3*b(Nx,Ny)-4*b(Nx-1,Ny)+b(Nx-2,Ny))/(2*dx));
83 b_num(Nx,Ny,2) = ((3*b(Nx,Ny)-4*b(Nx,Ny-1)+b(Nx,Ny-2))/(2*dy));
84 for i=2:Nx-1
85     for j=2:Ny-1
86         b_num(i,j,1) = ((b(i+1,j)-b(i-1,j))/(2*dx));
87         b_num(i,j,2) = ((b(i,j+1)-b(i,j-1))/(2*dy));
88     end
89 end
90
91 for i=1:Nx
92     for j=1:Ny
93         %if (xmid(i)^2)+(ymid(j)^2) < 10
94             % u(i,j,1) = 1 - b(i,j);
95         %else
96             % u(i,j,1) = 1 - b(i,j);
97         %end
98         u(i,j,1) = 1 - b(i,j);
99         u(i,j,2) = 2*u(i,j,1);
100        u(i,j,3) = 0;
101    end
102 end
103
104 q_initial(1) = 1;
105 q_initial(2) = 2;
106 q_initial(3) = 0;
107
108
109
110 %% Begin time stepping
111
112 t=0;
113
114 while t<tmax
115
116     count = count + 1;
117
118     % Adjust time
119     max_wave_speed = 0;
120     for i = 1:Nx
121         for j = 1:Ny
122             if abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
123                 max_wave_speed = abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1)));
124             end
125             if abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
126                 max_wave_speed = abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1)));
127             end
128             if abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
129                 max_wave_speed = abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1)));
130             end
131             if abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
132                 max_wave_speed = abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1)));
133             end
134         end
135     end
136     dt = CFL * (min(dx,dy) / (2 * max_wave_speed));
137
138     t = t+dt;
139
140     if t - tmax > 0
141         t = t-dt;
142         dt = tmax - t;

```



```

143     t = t+dt;
144     break_this_loop = 1;
145 end
146
147
148
149 %% Calculate timestep
150
151 uhelp=u;
152
153 for istage=1:3
154
155     %% Begin computation net cell fluxes
156     for i=1:Nx+1
157         for j=1:Ny
158             if i==1
159                 FL(1,j,1)=q_initial(1);
160                 FL(1,j,2)=q_initial(2);
161                 FL(1,j,3)=q_initial(3);
162             elseif i==2
163                 FL(2,j,1)=(1/2)*(uhelp(1,j,1)+uhelp(2,j,1));
164                 FL(2,j,2)=(1/2)*(uhelp(1,j,2)+uhelp(2,j,2));
165                 FL(2,j,3)=(1/2)*(uhelp(1,j,3)+uhelp(2,j,3));
166             elseif i==Nx+1
167                 FL(Nx+1,j,1)=(3/2)*uhelp(Nx,j,1)-(1/2)*uhelp(Nx-1,j,1);
168                 FL(Nx+1,j,2)=(3/2)*uhelp(Nx,j,2)-(1/2)*uhelp(Nx-1,j,2);
169                 FL(Nx+1,j,3)=(3/2)*uhelp(Nx,j,3)-(1/2)*uhelp(Nx-1,j,3);
170             else
171                 for k=1:3
172                     FL(i,j,k) = uhelp(i-1,j,k) + (1/2) * phi_t( state_inter , uhelp(i,j,k)-
uhelp(i-1,j,k) , uhelp(i-1,j,k)-uhelp(i-2,j,k) );
173                 end
174             end
175         end
176     end
177
178     for i=1:Nx
179         for j=1:Ny+1
180             if j==1
181                 GL(i,1,1)=q_initial(1);
182                 GL(i,1,2)=q_initial(2);
183                 GL(i,1,3)=q_initial(3);
184             elseif j==2
185                 GL(i,2,1)=(1/2)*(uhelp(i,1,1)+uhelp(i,2,1));
186                 GL(i,2,2)=(1/2)*(uhelp(i,1,2)+uhelp(i,2,2));
187                 GL(i,2,3)=(1/2)*(uhelp(i,1,3)+uhelp(i,2,3));
188             elseif j==Ny+1
189                 GL(i,Ny+1,1)=(3/2)*uhelp(i,Ny,1)-(1/2)*uhelp(i,Ny-1,1);
190                 GL(i,Ny+1,2)=(3/2)*uhelp(i,Ny,2)-(1/2)*uhelp(i,Ny-1,2);
191                 GL(i,Ny+1,3)=(3/2)*uhelp(i,Ny,3)-(1/2)*uhelp(i,Ny-1,3);
192             else
193                 for k=1:3
194                     GL(i,j,k) = uhelp(i,j-1,k) + (1/2) * phi_t( state_inter , uhelp(i,j,k)-
uhelp(i,j-1,k) , uhelp(i,j-1,k)-uhelp(i,j-2,k) );
195                 end
196             end
197         end
198     end
199
200     for i=1:Nx+1
201         for j=1:Ny
202             if i==1
203                 FR(1,j,1)=(3/2)*uhelp(1,j,1)-(1/2)*uhelp(2,j,1);
204                 FR(1,j,2)=(3/2)*uhelp(1,j,2)-(1/2)*uhelp(2,j,2);
205                 FR(1,j,3)=(3/2)*uhelp(1,j,3)-(1/2)*uhelp(2,j,3);
206             elseif i==Nx
207                 FR(Nx,j,1)=(1/2)*(uhelp(Nx-1,j,1)+uhelp(Nx,j,1));
208                 FR(Nx,j,2)=(1/2)*(uhelp(Nx-1,j,2)+uhelp(Nx,j,2));
209                 FR(Nx,j,3)=(1/2)*(uhelp(Nx-1,j,3)+uhelp(Nx,j,3));
210             elseif i==Nx+1
211                 FR(Nx+1,j,1)=q_initial(1);
212                 FR(Nx+1,j,2)=q_initial(2);
213                 FR(Nx+1,j,3)=q_initial(3);

```

```

214         else
215             for k=1:3
216                 FR(i,j,k) = uhelp(i,j,k) + (1/2) * phi_t( state_inter , uhelp(i-1,j,k)-
uhelp(i,j,k) , uhelp(i,j,k)-uhelp(i+1,j,k) );
217                 end
218             end
219         end
220     end
221
222     for i=1:Nx
223         for j=1:Ny+1
224             if j==1
225                 GR(i,1,1)=(3/2)*uhelp(i,1,1)-(1/2)*uhelp(i,2,1);
226                 GR(i,1,2)=(3/2)*uhelp(i,1,2)-(1/2)*uhelp(i,2,2);
227                 GR(i,1,3)=(3/2)*uhelp(i,1,3)-(1/2)*uhelp(i,2,3);
228             elseif j==Ny
229                 GR(i,Ny,1)=(1/2)*(uhelp(i,Ny-1,1)+uhelp(i,Ny,1));
230                 GR(i,Ny,2)=(1/2)*(uhelp(i,Ny-1,2)+uhelp(i,Ny,2));
231                 GR(i,Ny,3)=(1/2)*(uhelp(i,Ny-1,3)+uhelp(i,Ny,3));
232             elseif j==Ny+1
233                 GR(i,Ny+1,1)=q_initial(1);
234                 GR(i,Ny+1,2)=q_initial(2);
235                 GR(i,Ny+1,3)=q_initial(3);
236             else
237                 for k=1:3
238                     GR(i,j,k) = uhelp(i,j,k) + (1/2) * phi_t( state_inter , uhelp(i,j-1,k)-
uhelp(i,j,k) , uhelp(i,j,k)-uhelp(i,j+1,k) );
239                     end
240                 end
241             end
242         end
243
244
245
246     %% Flux method: Flux vector splitting
247     for i=1:Nx+1
248         for j=1:Ny
249             qm = FL(i,j,:);
250             lambda1 = (qm(2)/qm(1)) - sqrt(g*qm(1));
251             lambda2 = (qm(2)/qm(1));
252             lambda3 = (qm(2)/qm(1)) + sqrt(g*qm(1));
253             diag1 = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);
254             R1 = [1 0 1; lambda1 0 lambda3; qm(3)/qm(1) 1 qm(3)/qm(1) ];
255
256             qm = FR(i,j,:);
257             lambda1 = (qm(2)/qm(1)) - sqrt(g*qm(1));
258             lambda2 = (qm(2)/qm(1));
259             lambda3 = (qm(2)/qm(1)) + sqrt(g*qm(1));
260             diag2 = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);
261             R2 = [1 0 1; lambda1 0 lambda3; qm(3)/qm(1) 1 qm(3)/qm(1) ];
262
263             F(i,j,:) = (1/2)*([FL(i,j,2); ((FL(i,j,2)^2)/FL(i,j,1))+(1/2)*g*(FL(i,j,1)^2);
FL(i,j,3)*FL(i,j,2)/(FL(i,j,1)^2)]+[FR(i,j,2); ((FR(i,j,2)^2)/FR(i,j,1))+(1/2)*g*(FR(i,j,1)^2);
FR(i,j,3)*FR(i,j,2)/(FR(i,j,1)^2)]) -(1/4)*R1*diag1*(R1\[FR(i,j,1)-FL(i,j,1); FR(i,j,2)-FL(i,j,2);
FR(i,j,3)-FL(i,j,3)]) -(1/4)*R2*diag2*(R2\[FR(i,j,1)-FL(i,j,1); FR(i,j,2)-FL(i,j,2);
FR(i,j,3)-FL(i,j,3)]);
264         end
265     end
266
267     for i=1:Nx
268         for j=1:Ny+1
269             qm = GL(i,j,:);
270             lambda1 = (qm(3)/qm(1)) - sqrt(g*qm(1));
271             lambda2 = (qm(3)/qm(1));
272             lambda3 = (qm(3)/qm(1)) + sqrt(g*qm(1));
273             diag1 = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);
274             R1 = [1 0 1; qm(2)/qm(1) 1 qm(2)/qm(1); lambda1 0 lambda3; ];
275
276             qm = GR(i,j,:);
277             lambda1 = (qm(3)/qm(1)) - sqrt(g*qm(1));
278             lambda2 = (qm(3)/qm(1));
279             lambda3 = (qm(3)/qm(1)) + sqrt(g*qm(1));
280             diag2 = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);

```

```

281         R2 = [1 0 1; qm(2)/qm(1) 1 qm(2)/qm(1); lambda1 0 lambda3; ];
282
283         G(i,j,:) = (1/2)*([GL(i,j,3); GL(i,j,3)*GL(i,j,2)/(GL(i,j,1)^2); ((GL(i,j,3)
^2)/GL(i,j,1))+(1/2)*g*(GL(i,j,1)^2)]+[GR(i,j,3); GR(i,j,3)*GR(i,j,2)/(GR(i,j,1)^2); ((GR(
i,j,3)^2)/GR(i,j,1))+(1/2)*g*(GR(i,j,1)^2)]) -(1/4)*R1*diag1*(R1\[GR(i,j,1)-GL(i,j,1); GR(
i,j,2)-GL(i,j,2); GR(i,j,3)-GL(i,j,3)]) -(1/4)*R2*diag2*(R2\[GR(i,j,1)-GL(i,j,1); GR(i,j,2)
-GL(i,j,2); GR(i,j,3)-GL(i,j,3)]);
284     end
285     end
286
287
288
289     %% Perform time-integration method
290
291     if istage == 1
292         for i=1:Nx
293             for j=1:Ny
294                 du1(i,j,1)=-(dt/dx)*(F(i+1,j,1)-F(i,j,1))-(dt/dy)*(G(i,j+1,1)-G(i,j,1));
295                 du1(i,j,2)=-(dt/dx)*(F(i+1,j,2)-F(i,j,2))-(dt/dy)*(G(i,j+1,2)-G(i,j,2))-dt
*g*(1/2)*(FR(i,j,1)+FL(i+1,j,1))*b_num(i,j,1);
296                 du1(i,j,3)=-(dt/dx)*(F(i+1,j,3)-F(i,j,3))-(dt/dy)*(G(i,j+1,3)-G(i,j,3))-dt
*g*(1/2)*(GR(i,j,1)+GL(i,j+1,1))*b_num(i,j,2);
297                 for k=1:3
298                     uhelp(i,j,k)=u(i,j,k)+du1(i,j,k);
299                 end
300             end
301         end
302     elseif istage == 2
303         for i=1:Nx
304             for j=1:Ny
305                 du2(i,j,1)=-(dt/dx)*(F(i+1,j,1)-F(i,j,1))-(dt/dy)*(G(i,j+1,1)-G(i,j,1));
306                 du2(i,j,2)=-(dt/dx)*(F(i+1,j,2)-F(i,j,2))-(dt/dy)*(G(i,j+1,2)-G(i,j,2))-dt
*g*(1/2)*(FR(i,j,1)+FL(i+1,j,1))*b_num(i,j,1);
307                 du2(i,j,3)=-(dt/dx)*(F(i+1,j,3)-F(i,j,3))-(dt/dy)*(G(i,j+1,3)-G(i,j,3))-dt
*g*(1/2)*(GR(i,j,1)+GL(i,j+1,1))*b_num(i,j,2);
308                 for k=1:3
309                     uhelp(i,j,k)=u(i,j,k)+(1/4)*du1(i,j,k)+(1/4)*du2(i,j,k);
310                 end
311             end
312         end
313     elseif istage == 3
314         for i=1:Nx
315             for j=1:Ny
316                 du3(i,j,1)=-(dt/dx)*(F(i+1,j,1)-F(i,j,1))-(dt/dy)*(G(i,j+1,1)-G(i,j,1));
317                 du3(i,j,2)=-(dt/dx)*(F(i+1,j,2)-F(i,j,2))-(dt/dy)*(G(i,j+1,2)-G(i,j,2))-dt
*g*(1/2)*(FR(i,j,1)+FL(i+1,j,1))*b_num(i,j,1);
318                 du3(i,j,3)=-(dt/dx)*(F(i+1,j,3)-F(i,j,3))-(dt/dy)*(G(i,j+1,3)-G(i,j,3))-dt
*g*(1/2)*(GR(i,j,1)+GL(i,j+1,1))*b_num(i,j,2);
319                 for k=1:3
320                     u(i,j,k)=u(i,j,k)+(1/6)*du1(i,j,k)+(1/6)*du2(i,j,k)+(4/6)*du3(i,j,k);
321                 end
322             end
323         end
324     end
325 end
326
327
328
329 %% Begin plotting
330 if mod(count,10) == 0
331     set(gcf,'Position',[100 100 600 500])
332     hplot = u(:,:,1) + b(:,:,);
333
334     surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
335     %surf(xmid,ymid,transpose(u(:,:,2))./u(:,:,1)),'EdgeColor','none','LineStyle','none');
336     %hold on
337     %surf(xmid,ymid,transpose(b(:,:)));
338     %view(2)
339     %hold off
340     xlabel('x');
341     ylabel('y');
342     zlabel('free surface s');
343     xlim([xmin xmax]);

```

```

344     ylim([ymin ymax]);
345     zlim([0.75 1.25]);
346     %zlim([1.75 2.25]);
347     h = colorbar;
348     h.Limits = [0.75 1.25];
349     title(['Nx=', num2str(Nx), ', Ny=', num2str(Ny), ', Tmax=', num2str(tmax), ', t=', num2str
(round(t,4))]);
350     drawnow
351
352     % angle measurement
353     j1 = 1;
354     j2 = 1;
355     for j = round(Ny/2):Ny
356         if u(Nx,j,1) > u(Nx,j1,1)
357             j1 = j;
358         end
359     end
360     for j = round(Ny/2):Ny
361         if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)
362             j2 = j;
363         end
364     end
365     j3 = 1;
366     j4 = 1;
367     for j = round(Ny/2):Ny
368         if u(Nx,j,1) < u(Nx,j3,1)
369             j3 = j;
370         end
371     end
372     for j = round(Ny/2):Ny
373         if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
374             j4 = j;
375         end
376     end
377     disp('current time is ')
378     disp(t)
379     disp('max')
380     disp(atan(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
381     disp(u(Nx,j1,1))
382     disp(u(round(11*Nx/20),j2,1))
383     disp('min')
384     disp(atan(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
385     disp(u(Nx,j3,1))
386     disp(u(round(11*Nx/20),j4,1))
387
388     end
389 end
390
391
392
393 %% Begin plotting
394 set(gcf,'Position',[100 100 600 500])
395 hplot = u(:, :, 1) + b(:, :);
396
397 surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
398 %hold on
399 %surf(xmid,ymid,transpose(b(:,:)));
400 %view(2)
401 %hold off
402 xlabel('x');
403 ylabel('y');
404 zlabel('free surface s');
405 xlim([xmin xmax]);
406 ylim([ymin ymax]);
407 zlim([0.75 1.25]);
408 h = colorbar;
409 h.Limits = [0.75 1.25];
410 drawnow
411
412 % angle measurement
413 j1 = 1;
414 j2 = 1;
415 for j = round(Ny/2):Ny

```

```
416     if u(Nx,j,1) > u(Nx,j1,1)
417         j1 = j;
418     end
419 end
420 for j = round(Ny/2):Ny
421     if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)
422         j2 = j;
423     end
424 end
425 j3 = 1;
426 j4 = 1;
427 for j = round(Ny/2):Ny
428     if u(Nx,j,1) < u(Nx,j3,1)
429         j3 = j;
430     end
431 end
432 for j = round(Ny/2):Ny
433     if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
434         j4 = j;
435     end
436 end
437 disp('current time is ')
438 disp(t)
439 disp('max')
440 disp(atand(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
441 disp(u(Nx,j1,1))
442 disp(u(round(11*Nx/20),j2,1))
443 disp('min')
444 disp(atand(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
445 disp(u(Nx,j3,1))
446 disp(u(round(11*Nx/20),j4,1))
```

B.8 Test case 3: Midpoint rule

The Matlab code below simulates the midpoint rule method in test case 3.

```
1 clear all
2 format long
3
4 %% Select conditions
5
6 state_inter = 4;
7
8
9
10 %% Define parameters
11 Nx = 100;
12 xmin = -10;
13 xmax = 10;
14 dx = (xmax-xmin)/Nx;
15 x = xmin:dx:xmax;
16 xmid = (1/2)*(x(1:Nx)+x(2:Nx+1));
17
18 Ny = 100;
19 ymin = -10;
20 ymax = 10;
21 dy = (ymax-ymin)/Ny;
22 y = ymin:dy:ymax;
23 ymid = (1/2)*(y(1:Ny)+y(2:Ny+1));
24
25 CFL = 0.9;
26 tmax = 15;
27
28 g = 1;
29
30 break_this_loop = 0;
31 count = 0;
32
33
34
35 %% Declare sizes of matrices and vectors to lower running time
36 F = zeros(Nx+1,Ny,3);
37 G = zeros(Nx,Ny+1,3);
38
39 FL = zeros(Nx+1,Ny,3);
40 FR = zeros(Nx+1,Ny,3);
41 GL = zeros(Nx,Ny+1,3);
42 GR = zeros(Nx,Ny+1,3);
43
44 r = zeros(3,1);
45 q_initial = zeros(3,1);
46
47 du1 = zeros(Nx,Ny,3);
48 du2 = zeros(Nx,Ny,3);
49 du3 = zeros(Nx,Ny,3);
50 u = zeros(Nx,Ny,3);
51 b = zeros(Nx,Ny,1);
52 b_num = zeros(Nx,Ny,2);
53
54
55
56 %% Begin initial condition initialization
57 for i=1:Nx
58     for j=1:Ny
59         b(i,j) = (1/5)*exp(-(xmid(i)^2)-(ymid(j)^2));
60     end
61 end
62
63 for i=2:Nx-1
64     b_num(i,1,1) = ((b(i+1,1)-b(i-1,1))/(2*dx));
65     b_num(i,1,2) = ((-3*b(i,1)+4*b(i,2)-b(i,3))/(2*dy));
66     b_num(i,Ny,1) = ((b(i+1,Ny)-b(i-1,Ny))/(2*dx));
67     b_num(i,Ny,2) = ((3*b(i,Ny)-4*b(i,Ny-1)+b(i,Ny-2))/(2*dy));
68 end
69 for j=2:Ny-1
```

```

70     b_num(1,j,1) = ((-3*b(1,j)+4*b(2,j)-b(3,j))/(2*dx));
71     b_num(1,j,2) = ((b(1,j+1)-b(1,j-1))/(2*dy));
72     b_num(Nx,j,1) = ((3*b(Nx,j)-4*b(Nx-1,j)+b(Nx-2,j))/(2*dx));
73     b_num(Nx,j,2) = ((b(Nx,j+1)-b(Nx,j-1))/(2*dy));
74 end
75 b_num(1,1,1) = ((-3*b(1,1)+4*b(2,1)-b(3,1))/(2*dx));
76 b_num(1,1,2) = ((-3*b(1,1)+4*b(1,2)-b(1,3))/(2*dy));
77 b_num(1,Ny,1) = ((-3*b(1,Ny)+4*b(2,Ny)-b(3,Ny))/(2*dx));
78 b_num(1,Ny,2) = ((3*b(1,Ny)-4*b(1,Ny-1)+b(1,Ny-2))/(2*dy));
79 snapnow;
80 b_num(Nx,1,1) = ((3*b(Nx,1)-4*b(Nx-1,1)+b(Nx-2,1))/(2*dx));
81 b_num(Nx,1,2) = ((-3*b(Nx,1)+4*b(Nx,2)-b(Nx,3))/(2*dy));
82 b_num(Nx,Ny,1) = ((3*b(Nx,Ny)-4*b(Nx-1,Ny)+b(Nx-2,Ny))/(2*dx));
83 b_num(Nx,Ny,2) = ((3*b(Nx,Ny)-4*b(Nx,Ny-1)+b(Nx,Ny-2))/(2*dy));
84 for i=2:Nx-1
85     for j=2:Ny-1
86         b_num(i,j,1) = ((b(i+1,j)-b(i-1,j))/(2*dx));
87         b_num(i,j,2) = ((b(i,j+1)-b(i,j-1))/(2*dy));
88     end
89 end
90
91 for i=1:Nx
92     for j=1:Ny
93         %if (xmid(i)^2)+(ymid(j)^2) < 10
94         %    u(i,j,1) = 1 - b(i,j);
95         %else
96         %    u(i,j,1) = 1 - b(i,j);
97         %end
98         u(i,j,1) = 1 - b(i,j);
99         u(i,j,2) = 2*u(i,j,1);
100        u(i,j,3) = 0;
101    end
102 end
103
104 q_initial(1) = 1;
105 q_initial(2) = 2;
106 q_initial(3) = 0;
107
108
109
110 %% Begin time stepping
111
112 t=0;
113
114 while t<tmax
115
116     count = count + 1;
117
118     % Adjust time
119     max_wave_speed = 0;
120     for i = 1:Nx
121         for j = 1:Ny
122             if abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
123                 max_wave_speed = abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1)));
124             end
125             if abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
126                 max_wave_speed = abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1)));
127             end
128             if abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
129                 max_wave_speed = abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1)));
130             end
131             if abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
132                 max_wave_speed = abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1)));
133             end
134         end
135     end
136     dt = CFL * (min(dx,dy) / (2 * max_wave_speed));
137
138     t = t+dt;
139
140     if t - tmax > 0
141         t = t-dt;
142         dt = tmax - t;

```

```

143     t = t+dt;
144     break_this_loop = 1;
145 end
146
147
148
149 %% Calculate timestep
150
151 uhelp=u;
152
153 for istage=1:3
154
155     %% Begin computation net cell fluxes
156     for i=1:Nx+1
157         for j=1:Ny
158             if i==1
159                 FL(1,j,1)=q_initial(1);
160                 FL(1,j,2)=q_initial(2);
161                 FL(1,j,3)=q_initial(3);
162             elseif i==2
163                 FL(2,j,1)=(1/2)*(uhelp(1,j,1)+uhelp(2,j,1));
164                 FL(2,j,2)=(1/2)*(uhelp(1,j,2)+uhelp(2,j,2));
165                 FL(2,j,3)=(1/2)*(uhelp(1,j,3)+uhelp(2,j,3));
166             elseif i==Nx+1
167                 FL(Nx+1,j,1)=(3/2)*uhelp(Nx,j,1)-(1/2)*uhelp(Nx-1,j,1);
168                 FL(Nx+1,j,2)=(3/2)*uhelp(Nx,j,2)-(1/2)*uhelp(Nx-1,j,2);
169                 FL(Nx+1,j,3)=(3/2)*uhelp(Nx,j,3)-(1/2)*uhelp(Nx-1,j,3);
170             else
171                 for k=1:3
172                     FL(i,j,k) = uhelp(i-1,j,k) + (1/2) * phi_t( state_inter , uhelp(i,j,k)-
uhelp(i-1,j,k) , uhelp(i-1,j,k)-uhelp(i-2,j,k) );
173                 end
174             end
175         end
176     end
177
178     for i=1:Nx
179         for j=1:Ny+1
180             if j==1
181                 GL(i,1,1)=q_initial(1);
182                 GL(i,1,2)=q_initial(2);
183                 GL(i,1,3)=q_initial(3);
184             elseif j==2
185                 GL(i,2,1)=(1/2)*(uhelp(i,1,1)+uhelp(i,2,1));
186                 GL(i,2,2)=(1/2)*(uhelp(i,1,2)+uhelp(i,2,2));
187                 GL(i,2,3)=(1/2)*(uhelp(i,1,3)+uhelp(i,2,3));
188             elseif j==Ny+1
189                 GL(i,Ny+1,1)=(3/2)*uhelp(i,Ny,1)-(1/2)*uhelp(i,Ny-1,1);
190                 GL(i,Ny+1,2)=(3/2)*uhelp(i,Ny,2)-(1/2)*uhelp(i,Ny-1,2);
191                 GL(i,Ny+1,3)=(3/2)*uhelp(i,Ny,3)-(1/2)*uhelp(i,Ny-1,3);
192             else
193                 for k=1:3
194                     GL(i,j,k) = uhelp(i,j-1,k) + (1/2) * phi_t( state_inter , uhelp(i,j,k)-
uhelp(i,j-1,k) , uhelp(i,j-1,k)-uhelp(i,j-2,k) );
195                 end
196             end
197         end
198     end
199
200     for i=1:Nx+1
201         for j=1:Ny
202             if i==1
203                 FR(1,j,1)=(3/2)*uhelp(1,j,1)-(1/2)*uhelp(2,j,1);
204                 FR(1,j,2)=(3/2)*uhelp(1,j,2)-(1/2)*uhelp(2,j,2);
205                 FR(1,j,3)=(3/2)*uhelp(1,j,3)-(1/2)*uhelp(2,j,3);
206             elseif i==Nx
207                 FR(Nx,j,1)=(1/2)*(uhelp(Nx-1,j,1)+uhelp(Nx,j,1));
208                 FR(Nx,j,2)=(1/2)*(uhelp(Nx-1,j,2)+uhelp(Nx,j,2));
209                 FR(Nx,j,3)=(1/2)*(uhelp(Nx-1,j,3)+uhelp(Nx,j,3));
210             elseif i==Nx+1
211                 FR(Nx+1,j,1)=q_initial(1);
212                 FR(Nx+1,j,2)=q_initial(2);
213                 FR(Nx+1,j,3)=q_initial(3);

```



```

214         else
215             for k=1:3
216                 FR(i,j,k) = uhelp(i,j,k) + (1/2) * phi_t( state_inter , uhelp(i-1,j,k)-
uhelp(i,j,k) , uhelp(i,j,k)-uhelp(i+1,j,k) );
217                 end
218             end
219         end
220     end
221
222     for i=1:Nx
223         for j=1:Ny+1
224             if j==1
225                 GR(i,1,1)=(3/2)*uhelp(i,1,1)-(1/2)*uhelp(i,2,1);
226                 GR(i,1,2)=(3/2)*uhelp(i,1,2)-(1/2)*uhelp(i,2,2);
227                 GR(i,1,3)=(3/2)*uhelp(i,1,3)-(1/2)*uhelp(i,2,3);
228             elseif j==Ny
229                 GR(i,Ny,1)=(1/2)*(uhelp(i,Ny-1,1)+uhelp(i,Ny,1));
230                 GR(i,Ny,2)=(1/2)*(uhelp(i,Ny-1,2)+uhelp(i,Ny,2));
231                 GR(i,Ny,3)=(1/2)*(uhelp(i,Ny-1,3)+uhelp(i,Ny,3));
232             elseif j==Ny+1
233                 GR(i,Ny+1,1)=q_initial(1);
234                 GR(i,Ny+1,2)=q_initial(2);
235                 GR(i,Ny+1,3)=q_initial(3);
236             else
237                 for k=1:3
238                     GR(i,j,k) = uhelp(i,j,k) + (1/2) * phi_t( state_inter , uhelp(i,j-1,k)-
uhelp(i,j,k) , uhelp(i,j,k)-uhelp(i,j+1,k) );
239                     end
240                 end
241             end
242         end
243
244
245
246     %% Flux method: Flux vector splitting
247     for i=1:Nx+1
248         for j=1:Ny
249             qm = (1/2)*( FL(i,j,:) + FR(i,j,:) );
250             lambda1 = (qm(2)/qm(1)) - sqrt(g*qm(1));
251             lambda2 = (qm(2)/qm(1));
252             lambda3 = (qm(2)/qm(1)) + sqrt(g*qm(1));
253             diagm = diag( [abs(lambda1) ; abs(lambda2) ; abs(lambda3)] );
254             Rm = [ 1 0 1; lambda1 0 lambda3; qm(3)/qm(1) 1 qm(3)/qm(1) ];
255
256             F(i,j,:) = (1/2)*([FL(i,j,2); ((FL(i,j,2)^2)/FL(i,j,1))+(1/2)*g*(FL(i,j,1)^2);
FL(i,j,3)*FL(i,j,2)/(FL(i,j,1)^2)+[FR(i,j,2); ((FR(i,j,2)^2)/FR(i,j,1))+(1/2)*g*(FR(i,j,1)^2);
FR(i,j,3)*FR(i,j,2)/(FR(i,j,1)^2)] - (1/2)*Rm*diagm*(Rm\[FR(i,j,1)-FL(i,j,1); FR(i,j,2)-FL(i,j,2); FR(i,j,3)-FL(i,j,3)]);
257             end
258         end
259
260     for i=1:Nx
261         for j=1:Ny+1
262             qm = (1/2)*( GL(i,j,:) + GR(i,j,:) );
263             lambda1 = (qm(3)/qm(1)) - sqrt(g*qm(1));
264             lambda2 = (qm(3)/qm(1));
265             lambda3 = (qm(3)/qm(1)) + sqrt(g*qm(1));
266             diagm = diag( [abs(lambda1) ; abs(lambda2) ; abs(lambda3)] );
267             Rm = [ 1 0 1; qm(2)/qm(1) 1 qm(2)/qm(1); lambda1 0 lambda3 ];
268
269             G(i,j,:) = (1/2)*([GL(i,j,3); GL(i,j,3)*GL(i,j,2)/(GL(i,j,1)^2); ((GL(i,j,3)^2)/GL(i,j,1))+(1/2)*g*(GL(i,j,1)^2)+[GR(i,j,3); GR(i,j,3)*GR(i,j,2)/(GR(i,j,1)^2); ((GR(i,j,3)^2)/GR(i,j,1))+(1/2)*g*(GR(i,j,1)^2)] - (1/2)*Rm*diagm*(Rm\[GR(i,j,1)-GL(i,j,1); GR(i,j,2)-GL(i,j,2); GR(i,j,3)-GL(i,j,3)]);
270             end
271         end
272
273
274
275     %% Perform time-integration method
276
277     if istage == 1
278         for i=1:Nx

```

```

279     for j=1:Ny
280         du1(i,j,1)=-(dt/dx)*(F(i+1,j,1)-F(i,j,1))-(dt/dy)*(G(i,j+1,1)-G(i,j,1));
281         du1(i,j,2)=-(dt/dx)*(F(i+1,j,2)-F(i,j,2))-(dt/dy)*(G(i,j+1,2)-G(i,j,2))-dt
282 *g*(1/2)*(FR(i,j,1)+FL(i+1,j,1))*b_num(i,j,1);
283         du1(i,j,3)=-(dt/dx)*(F(i+1,j,3)-F(i,j,3))-(dt/dy)*(G(i,j+1,3)-G(i,j,3))-dt
284 *g*(1/2)*(GR(i,j,1)+GL(i,j+1,1))*b_num(i,j,2);
285         for k=1:3
286             uhelp(i,j,k)=u(i,j,k)+du1(i,j,k);
287         end
288     end
289     elseif istage == 2
290         for i=1:Nx
291             for j=1:Ny
292                 du2(i,j,1)=-(dt/dx)*(F(i+1,j,1)-F(i,j,1))-(dt/dy)*(G(i,j+1,1)-G(i,j,1));
293                 du2(i,j,2)=-(dt/dx)*(F(i+1,j,2)-F(i,j,2))-(dt/dy)*(G(i,j+1,2)-G(i,j,2))-dt
294 *g*(1/2)*(FR(i,j,1)+FL(i+1,j,1))*b_num(i,j,1);
295                 du2(i,j,3)=-(dt/dx)*(F(i+1,j,3)-F(i,j,3))-(dt/dy)*(G(i,j+1,3)-G(i,j,3))-dt
296 *g*(1/2)*(GR(i,j,1)+GL(i,j+1,1))*b_num(i,j,2);
297                 for k=1:3
298                     uhelp(i,j,k)=u(i,j,k)+(1/4)*du1(i,j,k)+(1/4)*du2(i,j,k);
299                 end
300             end
301         end
302     elseif istage == 3
303         for i=1:Nx
304             for j=1:Ny
305                 du3(i,j,1)=-(dt/dx)*(F(i+1,j,1)-F(i,j,1))-(dt/dy)*(G(i,j+1,1)-G(i,j,1));
306                 du3(i,j,2)=-(dt/dx)*(F(i+1,j,2)-F(i,j,2))-(dt/dy)*(G(i,j+1,2)-G(i,j,2))-dt
307 *g*(1/2)*(FR(i,j,1)+FL(i+1,j,1))*b_num(i,j,1);
308                 du3(i,j,3)=-(dt/dx)*(F(i+1,j,3)-F(i,j,3))-(dt/dy)*(G(i,j+1,3)-G(i,j,3))-dt
309 *g*(1/2)*(GR(i,j,1)+GL(i,j+1,1))*b_num(i,j,2);
310                 for k=1:3
311                     u(i,j,k)=u(i,j,k)+(1/6)*du1(i,j,k)+(1/6)*du2(i,j,k)+(4/6)*du3(i,j,k);
312                 end
313             end
314         end
315     end
316 end
317
318 %% Begin plotting
319 if mod(count,10) == 0
320     set(gcf,'Position',[100 100 600 500])
321     hplot = u(:, :, 1) + b(:, :);
322
323     surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
324     %surf(xmid,ymid,transpose(u(:, :, 2) ./ u(:, :, 1)),'EdgeColor','none','LineStyle','none');
325     %hold on
326     %surf(xmid,ymid,transpose(b(:, :)));
327     %view(2)
328     %hold off
329     xlabel('x');
330     ylabel('y');
331     zlabel('free surface s');
332     xlim([xmin xmax]);
333     ylim([ymin ymax]);
334     zlim([0.75 1.25]);
335     %zlim([1.75 2.25]);
336     h = colorbar;
337     h.Limits = [0.75 1.25];
338     title(['Nx=',num2str(Nx),', Ny=',num2str(Ny),', Tmax=',num2str(tmax),', t=',num2str
339 (round(t,4))] );
340     drawnow
341
342     % angle measurement
343     j1 = 1;
344     j2 = 1;
345     for j = round(Ny/2):Ny
346         if u(Nx,j,1) > u(Nx,j1,1)
347             j1 = j;
348         end
349     end

```

```

345 end
346 for j = round(Ny/2):Ny
347     if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)
348         j2 = j;
349     end
350 end
351 j3 = 1;
352 j4 = 1;
353 for j = round(Ny/2):Ny
354     if u(Nx,j,1) < u(Nx,j3,1)
355         j3 = j;
356     end
357 end
358 for j = round(Ny/2):Ny
359     if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
360         j4 = j;
361     end
362 end
363 disp('current time is ')
364 disp(t)
365 disp('max')
366 disp(atanh(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
367 disp(u(Nx,j1,1))
368 disp(u(round(11*Nx/20),j2,1))
369 disp('min')
370 disp(atanh(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
371 disp(u(Nx,j3,1))
372 disp(u(round(11*Nx/20),j4,1))
373
374 end
375 end
376
377
378
379 %% Begin plotting
380 set(gcf,'Position',[100 100 600 500])
381 hplot = u(:,:,1) + b(:,:,1);
382
383 surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
384 %hold on
385 %surf(xmid,ymid,transpose(b(:,:)));
386 %view(2)
387 %hold off
388 xlabel('x');
389 ylabel('y');
390 zlabel('free surface s');
391 xlim([xmin xmax]);
392 ylim([ymin ymax]);
393 zlim([0.75 1.25]);
394 h = colorbar;
395 h.Limits = [0.75 1.25];
396 title(['Nx=',num2str(Nx),', Ny=',num2str(Ny),', Tmax=',num2str(tmax),', t=',num2str
(round(t,4))]');
397 drawnow
398
399 % angle measurement
400 j1 = 1;
401 j2 = 1;
402 for j = round(Ny/2):Ny
403     if u(Nx,j,1) > u(Nx,j1,1)
404         j1 = j;
405     end
406 end
407 for j = round(Ny/2):Ny
408     if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)
409         j2 = j;
410     end
411 end
412 j3 = 1;
413 j4 = 1;
414 for j = round(Ny/2):Ny
415     if u(Nx,j,1) < u(Nx,j3,1)
416         j3 = j;

```

```
417     end
418   end
419   for j = round(Ny/2):Ny
420     if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
421       j4 = j;
422     end
423   end
424   disp('current time is ')
425   disp(t)
426   disp('max')
427   disp(atan2(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
428   disp(u(Nx,j1,1))
429   disp(u(round(11*Nx/20),j2,1))
430   disp('min')
431   disp(atan2(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
432   disp(u(Nx,j3,1))
433   disp(u(round(11*Nx/20),j4,1))
```

B.9 Test case 3: Roe's method

The Matlab code below simulates Roe's method in test case 3.

```
1 clear all
2 format long
3
4 %% Select conditions
5
6 state_inter = 4;
7
8
9
10 %% Define parameters
11 Nx = 100;
12 xmin = -10;
13 xmax = 10;
14 dx = (xmax-xmin)/Nx;
15 x = xmin:dx:xmax;
16 xmid = (1/2)*(x(1:Nx)+x(2:Nx+1));
17
18 Ny = 100;
19 ymin = -10;
20 ymax = 10;
21 dy = (ymax-ymin)/Ny;
22 y = ymin:dy:ymax;
23 ymid = (1/2)*(y(1:Ny)+y(2:Ny+1));
24
25 CFL = 0.9;
26 tmax = 5;
27
28 g = 1;
29
30 break_this_loop = 0;
31 count = 0;
32
33
34
35 %% Declare sizes of matrices and vectors to lower running time
36 F = zeros(Nx+1,Ny,3);
37 G = zeros(Nx,Ny+1,3);
38
39 FL = zeros(Nx+1,Ny,3);
40 FR = zeros(Nx+1,Ny,3);
41 GL = zeros(Nx,Ny+1,3);
42 GR = zeros(Nx,Ny+1,3);
43
44 r = zeros(3,1);
45 q_initial = zeros(3,1);
46
47 du1 = zeros(Nx,Ny,3);
48 du2 = zeros(Nx,Ny,3);
49 du3 = zeros(Nx,Ny,3);
50 u = zeros(Nx,Ny,3);
51 b = zeros(Nx,Ny,1);
52 b_num = zeros(Nx,Ny,2);
53
54
55
56 %% Begin initial condition initialization
57 for i=1:Nx
58     for j=1:Ny
59         b(i,j) = (1/5)*exp(-(xmid(i)^2)-(ymid(j)^2));
60     end
61 end
62
63 for i=2:Nx-1
64     b_num(i,1,1) = ((b(i+1,1)-b(i-1,1))/(2*dx));
65     b_num(i,1,2) = ((-3*b(i,1)+4*b(i,2)-b(i,3))/(2*dy));
66     b_num(i,Ny,1) = ((b(i+1,Ny)-b(i-1,Ny))/(2*dx));
67     b_num(i,Ny,2) = ((3*b(i,Ny)-4*b(i,Ny-1)+b(i,Ny-2))/(2*dy));
68 end
69 for j=2:Ny-1
```

```

70     b_num(1,j,1) = ((-3*b(1,j)+4*b(2,j)-b(3,j))/(2*dx));
71     b_num(1,j,2) = ((b(1,j+1)-b(1,j-1))/(2*dy));
72     b_num(Nx,j,1) = ((3*b(Nx,j)-4*b(Nx-1,j)+b(Nx-2,j))/(2*dx));
73     b_num(Nx,j,2) = ((b(Nx,j+1)-b(Nx,j-1))/(2*dy));
74 end
75 b_num(1,1,1) = ((-3*b(1,1)+4*b(2,1)-b(3,1))/(2*dx));
76 b_num(1,1,2) = ((-3*b(1,1)+4*b(1,2)-b(1,3))/(2*dy));
77 b_num(1,Ny,1) = ((-3*b(1,Ny)+4*b(2,Ny)-b(3,Ny))/(2*dx));
78 b_num(1,Ny,2) = ((3*b(1,Ny)-4*b(1,Ny-1)+b(1,Ny-2))/(2*dy));
79 snapnow;
80 b_num(Nx,1,1) = ((3*b(Nx,1)-4*b(Nx-1,1)+b(Nx-2,1))/(2*dx));
81 b_num(Nx,1,2) = ((-3*b(Nx,1)+4*b(Nx,2)-b(Nx,3))/(2*dy));
82 b_num(Nx,Ny,1) = ((3*b(Nx,Ny)-4*b(Nx-1,Ny)+b(Nx-2,Ny))/(2*dx));
83 b_num(Nx,Ny,2) = ((3*b(Nx,Ny)-4*b(Nx,Ny-1)+b(Nx,Ny-2))/(2*dy));
84 for i=2:Nx-1
85     for j=2:Ny-1
86         b_num(i,j,1) = ((b(i+1,j)-b(i-1,j))/(2*dx));
87         b_num(i,j,2) = ((b(i,j+1)-b(i,j-1))/(2*dy));
88     end
89 end
90
91 for i=1:Nx
92     for j=1:Ny
93         %if (xmid(i)^2)+(ymid(j)^2) < 10
94         %     u(i,j,1) = 1 - b(i,j);
95         %else
96         %     u(i,j,1) = 1 - b(i,j);
97         %end
98         u(i,j,1) = 1 - b(i,j);
99         u(i,j,2) = 2*u(i,j,1);
100        u(i,j,3) = 0;
101    end
102 end
103
104 q_initial(1) = 1;
105 q_initial(2) = 2;
106 q_initial(3) = 0;
107
108
109
110 %% Begin time stepping
111
112 t=0;
113
114 while t<tmax
115
116     count = count + 1;
117
118     % Adjust time
119     max_wave_speed = 0;
120     for i = 1:Nx
121         for j = 1:Ny
122             if abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
123                 max_wave_speed = abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1)));
124             end
125             if abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
126                 max_wave_speed = abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1)));
127             end
128             if abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
129                 max_wave_speed = abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1)));
130             end
131             if abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
132                 max_wave_speed = abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1)));
133             end
134         end
135     end
136     dt = CFL * (min(dx,dy) / (2 * max_wave_speed));
137
138     t = t+dt;
139
140     if t - tmax > 0
141         t = t-dt;
142         dt = tmax - t;

```

```

143     t = t+dt;
144     break_this_loop = 1;
145 end
146
147
148
149 %% Calculate timestep
150
151 uhelp=u;
152
153 for istage=1:3
154
155     %% Begin computation net cell fluxes
156     for i=1:Nx+1
157         for j=1:Ny
158             if i==1
159                 FL(1,j,1)=q_initial(1);
160                 FL(1,j,2)=q_initial(2);
161                 FL(1,j,3)=q_initial(3);
162             elseif i==2
163                 FL(2,j,1)=(1/2)*(uhelp(1,j,1)+uhelp(2,j,1));
164                 FL(2,j,2)=(1/2)*(uhelp(1,j,2)+uhelp(2,j,2));
165                 FL(2,j,3)=(1/2)*(uhelp(1,j,3)+uhelp(2,j,3));
166             elseif i==Nx+1
167                 FL(Nx+1,j,1)=(3/2)*uhelp(Nx,j,1)-(1/2)*uhelp(Nx-1,j,1);
168                 FL(Nx+1,j,2)=(3/2)*uhelp(Nx,j,2)-(1/2)*uhelp(Nx-1,j,2);
169                 FL(Nx+1,j,3)=(3/2)*uhelp(Nx,j,3)-(1/2)*uhelp(Nx-1,j,3);
170             else
171                 for k=1:3
172                     FL(i,j,k) = uhelp(i-1,j,k) + (1/2) * phi_t( state_inter , uhelp(i,j,k)-
uhelp(i-1,j,k) , uhelp(i-1,j,k)-uhelp(i-2,j,k) );
173                 end
174             end
175         end
176     end
177
178     for i=1:Nx
179         for j=1:Ny+1
180             if j==1
181                 GL(i,1,1)=q_initial(1);
182                 GL(i,1,2)=q_initial(2);
183                 GL(i,1,3)=q_initial(3);
184             elseif j==2
185                 GL(i,2,1)=(1/2)*(uhelp(i,1,1)+uhelp(i,2,1));
186                 GL(i,2,2)=(1/2)*(uhelp(i,1,2)+uhelp(i,2,2));
187                 GL(i,2,3)=(1/2)*(uhelp(i,1,3)+uhelp(i,2,3));
188             elseif j==Ny+1
189                 GL(i,Ny+1,1)=(3/2)*uhelp(i,Ny,1)-(1/2)*uhelp(i,Ny-1,1);
190                 GL(i,Ny+1,2)=(3/2)*uhelp(i,Ny,2)-(1/2)*uhelp(i,Ny-1,2);
191                 GL(i,Ny+1,3)=(3/2)*uhelp(i,Ny,3)-(1/2)*uhelp(i,Ny-1,3);
192             else
193                 for k=1:3
194                     GL(i,j,k) = uhelp(i,j-1,k) + (1/2) * phi_t( state_inter , uhelp(i,j,k)-
uhelp(i,j-1,k) , uhelp(i,j-1,k)-uhelp(i,j-2,k) );
195                 end
196             end
197         end
198     end
199
200     for i=1:Nx+1
201         for j=1:Ny
202             if i==1
203                 FR(1,j,1)=(3/2)*uhelp(1,j,1)-(1/2)*uhelp(2,j,1);
204                 FR(1,j,2)=(3/2)*uhelp(1,j,2)-(1/2)*uhelp(2,j,2);
205                 FR(1,j,3)=(3/2)*uhelp(1,j,3)-(1/2)*uhelp(2,j,3);
206             elseif i==Nx
207                 FR(Nx,j,1)=(1/2)*(uhelp(Nx-1,j,1)+uhelp(Nx,j,1));
208                 FR(Nx,j,2)=(1/2)*(uhelp(Nx-1,j,2)+uhelp(Nx,j,2));
209                 FR(Nx,j,3)=(1/2)*(uhelp(Nx-1,j,3)+uhelp(Nx,j,3));
210             elseif i==Nx+1
211                 FR(Nx+1,j,1)=q_initial(1);
212                 FR(Nx+1,j,2)=q_initial(2);
213                 FR(Nx+1,j,3)=q_initial(3);

```

```

214         else
215             for k=1:3
216                 FR(i,j,k) = uhelp(i,j,k) + (1/2) * phi_t( state_inter , uhelp(i-1,j,k)-
uhelp(i,j,k) , uhelp(i,j,k)-uhelp(i+1,j,k) );
217                 end
218             end
219         end
220     end
221
222     for i=1:Nx
223         for j=1:Ny+1
224             if j==1
225                 GR(i,1,1)=(3/2)*uhelp(i,1,1)-(1/2)*uhelp(i,2,1);
226                 GR(i,1,2)=(3/2)*uhelp(i,1,2)-(1/2)*uhelp(i,2,2);
227                 GR(i,1,3)=(3/2)*uhelp(i,1,3)-(1/2)*uhelp(i,2,3);
228             elseif j==Ny
229                 GR(i,Ny,1)=(1/2)*(uhelp(i,Ny-1,1)+uhelp(i,Ny,1));
230                 GR(i,Ny,2)=(1/2)*(uhelp(i,Ny-1,2)+uhelp(i,Ny,2));
231                 GR(i,Ny,3)=(1/2)*(uhelp(i,Ny-1,3)+uhelp(i,Ny,3));
232             elseif j==Ny+1
233                 GR(i,Ny+1,1)=q_initial(1);
234                 GR(i,Ny+1,2)=q_initial(2);
235                 GR(i,Ny+1,3)=q_initial(3);
236             else
237                 for k=1:3
238                     GR(i,j,k) = uhelp(i,j,k) + (1/2) * phi_t( state_inter , uhelp(i,j-1,k)-
uhelp(i,j,k) , uhelp(i,j,k)-uhelp(i,j+1,k) );
239                     end
240                 end
241             end
242         end
243
244
245
246     %% Flux method: Flux vector splitting
247     for i=1:Nx+1
248         for j=1:Ny
249             h_bar = (1/2)*(FL(i,j,1)+FR(i,j,1));
250             c_hat = sqrt(g*h_bar);
251             u_hat = ( (sqrt(FL(i,j,1))*(FL(i,j,2)/FL(i,j,1)))+(sqrt(FR(i,j,1))*(FR(i,j,2)/
FR(i,j,1))) )/(sqrt(FL(i,j,1))+sqrt(FR(i,j,1)));
252             v_hat = ( (sqrt(FL(i,j,1))*(FL(i,j,3)/FL(i,j,1)))+(sqrt(FR(i,j,1))*(FR(i,j,3)/
FR(i,j,1))) )/(sqrt(FL(i,j,1))+sqrt(FR(i,j,1)));
253
254             lambda1 = (u_hat) - c_hat;
255             lambda2 = (u_hat);
256             lambda3 = (u_hat) + c_hat;
257             diagm = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);
258             Rm = [1 0 1; lambda1 0 lambda3; v_hat 1 v_hat ];
259
260             F(i,j,:) = (1/2)*([FL(i,j,2); ((FL(i,j,2)^2)/FL(i,j,1))+(1/2)*g*(FL(i,j,1)^2);
FL(i,j,3)*FL(i,j,2)/(FL(i,j,1)^2)]+[FR(i,j,2); ((FR(i,j,2)^2)/FR(i,j,1))+(1/2)*g*(FR(i,j,1)^2);
FR(i,j,3)*FR(i,j,2)/(FR(i,j,1)^2)]) -(1/2)*Rm*diagm*(Rm\[FR(i,j,1)-FL(i,j,1); FR(i,
j,2)-FL(i,j,2); FR(i,j,3)-FL(i,j,3)]);
261         end
262     end
263
264     for i=1:Nx
265         for j=1:Ny+1
266             h_bar = (1/2)*(GL(i,j,1)+GR(i,j,1));
267             c_hat = sqrt(g*h_bar);
268             u_hat = ( (sqrt(GL(i,j,1))*(GL(i,j,2)/GL(i,j,1)))+(sqrt(GR(i,j,1))*(GR(i,j,2)/
GR(i,j,1))) )/(sqrt(GL(i,j,1))+sqrt(GR(i,j,1)));
269             v_hat = ( (sqrt(GL(i,j,1))*(GL(i,j,3)/GL(i,j,1)))+(sqrt(GR(i,j,1))*(GR(i,j,3)/
GR(i,j,1))) )/(sqrt(GL(i,j,1))+sqrt(GR(i,j,1)));
270
271             lambda1 = (v_hat) - c_hat;
272             lambda2 = (v_hat);
273             lambda3 = (v_hat) + c_hat;
274             diagm = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);
275             Rm = [1 0 1; u_hat 1 u_hat; lambda1 0 lambda3; ];
276

```



```

277         G(i,j,:) = (1/2)*([GL(i,j,3); GL(i,j,3)*GL(i,j,2)/(GL(i,j,1)^2); ((GL(i,j,3)
^2)/GL(i,j,1))+(1/2)*g*(GL(i,j,1)^2)]+[GR(i,j,3); GR(i,j,3)*GR(i,j,2)/(GR(i,j,1)^2); ((GR(
i,j,3)^2)/GR(i,j,1))+(1/2)*g*(GR(i,j,1)^2)]) -(1/2)*Rm*diagm*(Rm\[GR(i,j,1)-GL(i,j,1); GR(
i,j,2)-GL(i,j,2); GR(i,j,3)-GL(i,j,3)]);
278     end
279     end
280
281
282
283     %% Perform time-integration method
284
285     if istage == 1
286         for i=1:Nx
287             for j=1:Ny
288                 du1(i,j,1)=-(dt/dx)*(F(i+1,j,1)-F(i,j,1))-(dt/dy)*(G(i,j+1,1)-G(i,j,1));
289                 du1(i,j,2)=-(dt/dx)*(F(i+1,j,2)-F(i,j,2))-(dt/dy)*(G(i,j+1,2)-G(i,j,2))-dt
*g*(1/2)*(FR(i,j,1)+FL(i+1,j,1))*b_num(i,j,1);
290                 du1(i,j,3)=-(dt/dx)*(F(i+1,j,3)-F(i,j,3))-(dt/dy)*(G(i,j+1,3)-G(i,j,3))-dt
*g*(1/2)*(GR(i,j,1)+GL(i,j+1,1))*b_num(i,j,2);
291                 for k=1:3
292                     uhelp(i,j,k)=u(i,j,k)+du1(i,j,k);
293                 end
294             end
295         end
296         elseif istage == 2
297             for i=1:Nx
298                 for j=1:Ny
299                     du2(i,j,1)=-(dt/dx)*(F(i+1,j,1)-F(i,j,1))-(dt/dy)*(G(i,j+1,1)-G(i,j,1));
300                     du2(i,j,2)=-(dt/dx)*(F(i+1,j,2)-F(i,j,2))-(dt/dy)*(G(i,j+1,2)-G(i,j,2))-dt
*g*(1/2)*(FR(i,j,1)+FL(i+1,j,1))*b_num(i,j,1);
301                     du2(i,j,3)=-(dt/dx)*(F(i+1,j,3)-F(i,j,3))-(dt/dy)*(G(i,j+1,3)-G(i,j,3))-dt
*g*(1/2)*(GR(i,j,1)+GL(i,j+1,1))*b_num(i,j,2);
302                     for k=1:3
303                         uhelp(i,j,k)=u(i,j,k)+(1/4)*du1(i,j,k)+(1/4)*du2(i,j,k);
304                     end
305                 end
306             end
307         elseif istage == 3
308             for i=1:Nx
309                 for j=1:Ny
310                     du3(i,j,1)=-(dt/dx)*(F(i+1,j,1)-F(i,j,1))-(dt/dy)*(G(i,j+1,1)-G(i,j,1));
311                     du3(i,j,2)=-(dt/dx)*(F(i+1,j,2)-F(i,j,2))-(dt/dy)*(G(i,j+1,2)-G(i,j,2))-dt
*g*(1/2)*(FR(i,j,1)+FL(i+1,j,1))*b_num(i,j,1);
312                     du3(i,j,3)=-(dt/dx)*(F(i+1,j,3)-F(i,j,3))-(dt/dy)*(G(i,j+1,3)-G(i,j,3))-dt
*g*(1/2)*(GR(i,j,1)+GL(i,j+1,1))*b_num(i,j,2);
313                     for k=1:3
314                         u(i,j,k)=u(i,j,k)+(1/6)*du1(i,j,k)+(1/6)*du2(i,j,k)+(4/6)*du3(i,j,k);
315                     end
316                 end
317             end
318         end
319     end
320
321
322
323     %% Begin plotting
324     if mod(count,10) == 0
325         set(gcf,'Position',[100 100 600 500])
326         hplot = u(:,:,1) + b(:,:,);
327
328         surf(xmid,ymid,transpose(hplot(:,:)), 'EdgeColor','none','LineStyle','none');
329         %surf(xmid,ymid,transpose(u(:,:,2) ./ u(:,:,1)), 'EdgeColor','none','LineStyle','none');
330         %hold on
331         %surf(xmid,ymid,transpose(b(:,:)));
332         %view(2)
333         %hold off
334         xlabel('x');
335         ylabel('y');
336         zlabel('free surface s');
337         xlim([xmin xmax]);
338         ylim([ymin ymax]);
339         zlim([0.75 1.25]);
340         %zlim([1.75 2.25]);

```

```

341     h = colorbar;
342     h.Limits = [0.75 1.25];
343     title(['Nx=', num2str(Nx), ',   Ny=', num2str(Ny), ',   Tmax=', num2str(tmax), ',   t=', num2str
(round(t,4) )]);
344     drawnow
345
346     % angle measurement
347     j1 = 1;
348     j2 = 1;
349     for j = round(Ny/2):Ny
350         if u(Nx,j,1) > u(Nx,j1,1)
351             j1 = j;
352         end
353     end
354     for j = round(Ny/2):Ny
355         if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)
356             j2 = j;
357         end
358     end
359     j3 = 1;
360     j4 = 1;
361     for j = round(Ny/2):Ny
362         if u(Nx,j,1) < u(Nx,j3,1)
363             j3 = j;
364         end
365     end
366     for j = round(Ny/2):Ny
367         if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
368             j4 = j;
369         end
370     end
371     disp('current time is ')
372     disp(t)
373     disp('max ')
374     disp(atan2(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
375     disp(u(Nx,j1,1))
376     disp(u(round(11*Nx/20),j2,1))
377     disp('min ')
378     disp(atan2(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
379     disp(u(Nx,j3,1))
380     disp(u(round(11*Nx/20),j4,1))
381
382 end
383 end
384
385
386
387 %% Begin plotting
388 set(gcf,'Position',[100 100 600 500])
389 hplot = u(:,:,1) + b(:,:,1);
390
391 surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
392 hold on
393 %surf(xmid,ymid,transpose(b(:,:)));
394 %view(2)
395 %hold off
396 xlabel('x');
397 ylabel('y');
398 zlabel('free surface s');
399 xlim([xmin xmax]);
400 ylim([ymin ymax]);
401 zlim([0.75 1.25]);
402 h = colorbar;
403 h.Limits = [0.75 1.25];
404 %title(['Nx=', num2str(Nx), ',   Ny=', num2str(Ny), ',   Tmax=', num2str(tmax), ',   t=',
num2str(round(t,4) )]);
405 drawnow
406
407 % angle measurement
408 j1 = 1;
409 j2 = 1;
410 for j = round(Ny/2):Ny
411     if u(Nx,j,1) > u(Nx,j1,1)

```

```
412         j1 = j;
413     end
414 end
415 for j = round(Ny/2):Ny
416     if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)
417         j2 = j;
418     end
419 end
420 j3 = 1;
421 j4 = 1;
422 for j = round(Ny/2):Ny
423     if u(Nx,j,1) < u(Nx,j3,1)
424         j3 = j;
425     end
426 end
427 for j = round(Ny/2):Ny
428     if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
429         j4 = j;
430     end
431 end
432 disp('current time is ')
433 disp(t)
434 disp('max')
435 disp(atan(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
436 disp(u(Nx,j1,1))
437 disp(u(round(11*Nx/20),j2,1))
438 disp('min')
439 disp(atan(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20))))))
440 disp(u(Nx,j3,1))
441 disp(u(round(11*Nx/20),j4,1))
```