BACHELOR

Numerical methods for the shallow water equations

Burgerjon, Luc M.A.

Award date:
2021

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.

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Table of contents

Numerical Methods for
the Shallow Water Equations

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Table of contents

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

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

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

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 $\rho_t := \dfrac{\partial \rho}{\partial t}$.

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

$$\rho_t + u\rho_x + v\rho_y + w\rho_z + \rho\left(u_x + v_y + w_z\right) = 0, \tag{2.2a}$$

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

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

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

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

$$u_x + v_y + w_z = 0, \tag{2.3a}$$

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

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

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

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

$$z = b(x, y), \tag{2.4}$$

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

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

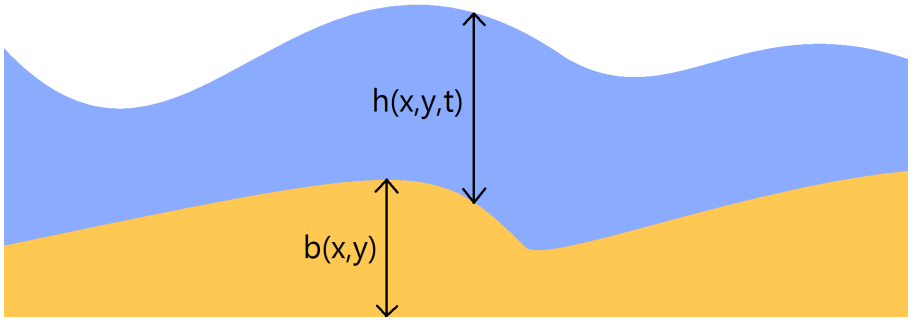

Figure 2.1: 1-dimensional sketch of the cross-section of the domain.

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

$$\psi(x, y, z, t) = 0, \tag{2.6}$$

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

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

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

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

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:

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

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

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

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

$$\frac{\mathrm{d}w}{\mathrm{d}t} = w_t + uw_x + vw_y + ww_z = 0. \tag{2.11}$$

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

$$u_x + v_y + w_z = 0, \tag{2.12a}$$

$$u_t + uu_x + vu_y + wu_z = -\frac{1}{\rho}p_x, \tag{2.12b}$$

$$v_t + uv_x + vv_y + wv_z = -\frac{1}{\rho}p_y, \tag{2.12c}$$

$$p_z = -\rho g. \tag{2.12d}$$

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

$$p = \rho g(s - z), \tag{2.13}$$

and hence

$$p_x = \rho g s_x, \tag{2.14a}$$
$$p_y = \rho g s_y. \tag{2.14b}$$

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 $\frac{\mathrm{d}u}{\mathrm{d}t}$ and $\frac{\mathrm{d}v}{\mathrm{d}t}$, 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

$$u_x + v_y + w_z = 0, \tag{2.15a}$$
$$u_t + uu_x + vu_y = -gs_x, \tag{2.15b}$$
$$v_t + uv_x + vv_y = -gs_y. \tag{2.15c}$$

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

$$\int_{z=b}^{z=s} (u_x + v_y + w_z)\,\mathrm{d}z = 0, \tag{2.16}$$

which using the Fundamental Theorem of Calculus can be written as

$$w|_{z=s} - w|_{z=b} + \int_{z=b}^{z=s} u_x\,\mathrm{d}z + \int_{z=b}^{z=s} v_y\,\mathrm{d}z = 0. \tag{2.17}$$

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

$$\int_{z=b}^{z=s} u_x\,\mathrm{d}z = \frac{\partial}{\partial x}\int_{z=b}^{z=s} u\,\mathrm{d}z - u|_{z=s}\,s_x + u|_{z=b}\,b_x, \tag{2.18a}$$
$$\int_{z=b}^{z=s} v_y\,\mathrm{d}z = \frac{\partial}{\partial y}\int_{z=b}^{z=s} v\,\mathrm{d}z - v|_{z=s}\,s_x + v|_{z=b}\,b_x. \tag{2.18b}$$

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

$$w|_{z=s} = (s_t + us_x + vs_y)\,|_{z=s}, \tag{2.19a}$$
$$w|_{z=b} = (ub_x + vb_y)\,|_{z=b}. \tag{2.19b}$$

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

$$s_t + \frac{\partial}{\partial x}\int_{z=b}^{z=s} u\,\mathrm{d}z + \frac{\partial}{\partial y}\int_{z=b}^{z=s} v\,\mathrm{d}z = 0. \tag{2.20}$$

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

$$h_t + (hu)_x + (hv)_y = 0, \tag{2.21}$$

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

$$h_t + (hu)_x + (hv)_y = 0, \tag{2.22a}$$
$$u_t + uu_x + vu_y = -gb_x, \tag{2.22b}$$
$$v_t + uv_x + vv_y = -gb_y. \tag{2.22c}$$

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

$$h\frac{\partial h}{\partial x} = \frac{\partial}{\partial x}\left(\frac{1}{2}h^2\right), \tag{2.23a}$$
$$h\frac{\partial h}{\partial y} = \frac{\partial}{\partial y}\left(\frac{1}{2}h^2\right). \tag{2.23b}$$

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

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

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

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

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:

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

where

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

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

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

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]

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

where

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

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

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

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 $x_i = i\Delta x$ $(i = 0, ..., n)$. Figure 3.1 illustrates this subdivision of the spatial domain. Furthermore, the solution is approximated at times $t_j = j\Delta t$ $(j = 1, ..., m)$ 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)

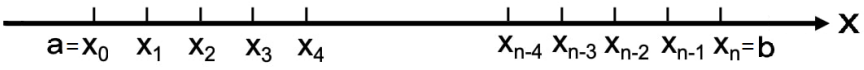

Figure 3.1: Sketch of the discretization of the spatial domain for the finite difference method.

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

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

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

Substracting 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

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

and u_x by

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

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

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

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

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

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

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

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 $x \in I_i$ $(i = 1, ..., M)$ 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

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

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

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

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

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

where

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

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.

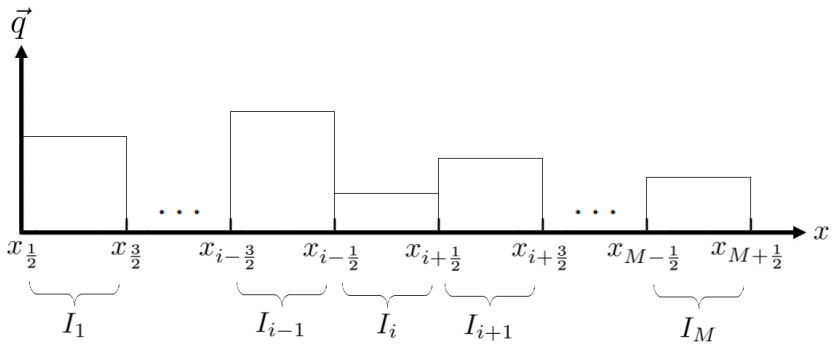

Figure 3.2: Sketch of the division of the spatial domain for the finite volume method.

3.2.3 State interpolation

In order to evaluate the fluxes $\vec{F}_{\frac{1}{2}}, \vec{F}_{\frac{3}{2}}, ..., \vec{F}_{M+\frac{1}{2}}$ in the finite volume equation (3.15), one needs the numerical solutions of $\vec{q}$ at $x_{\frac{1}{2}}, x_{\frac{3}{2}}, ..., x_{M+\frac{1}{2}}$, which we denote by $\vec{q}_{\frac{1}{2}}, \vec{q}_{\frac{3}{2}}, ..., \vec{q}_{M+\frac{1}{2}}$. 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

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

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

3.2.3.1 First-order upwind

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

$$\vec{q}^{\,l}_{i+\frac{1}{2}} = \vec{q}_i, \tag{3.18a}$$
$$\vec{q}^{\,r}_{i+\frac{1}{2}} = \vec{q}_{i+1}, \tag{3.18b}$$

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

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

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

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

where

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

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.

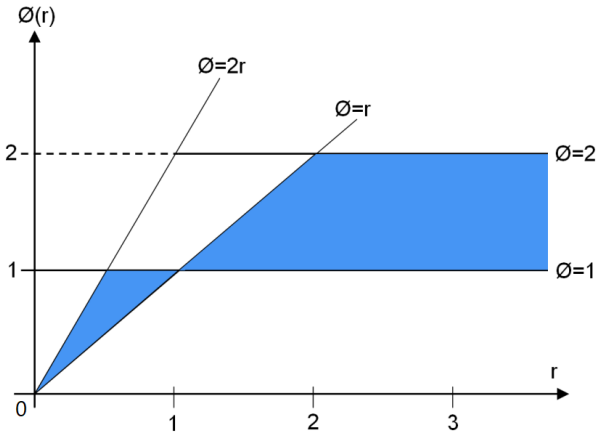

Figure 3.3: Sketch of the division of the spatial domain.

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

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

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

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

3.2.4 Flux evaluation

Once the state interpolation values have been calculated, the next step is to determine a flux evaluation function $\vec{F}_{i+\frac{1}{2}} = \vec{\mathcal{F}}(\vec{q}^{\,l}_{i+\frac{1}{2}}, \vec{q}^{\,r}_{i+\frac{1}{2}})$. 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

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

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]

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

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

$$\mathbf{A}^* = \mathbf{R}\Lambda^*\mathbf{R}^{-1}, \tag{3.27}$$

where Λ^* 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

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

where

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

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

$$\mathbf{A}_+^* := \mathbf{R}\Lambda_+^*\mathbf{R}^{*^{-1}}, \quad \mathbf{A}_-^* := \mathbf{R}\Lambda_-^*\mathbf{R}^{-1}, \tag{3.30}$$

where

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

One should note that it holds that

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

As a result, the inter cell flux is taken as

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

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

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

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

$$\mathbf{A} = \mathbf{S}\Lambda\mathbf{S}^{-1}, \tag{3.35}$$

where Λ 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

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

where

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

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

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

where

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

and again it holds that

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

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

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

A natural choice here is to take

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

Moreover, note that

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

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

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

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

$$|\mathbf{A}| := \mathbf{A}^+ - \mathbf{A}^-. \tag{3.45}$$

Hence, we consider flux difference splitting schemes of the form

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

where

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

and the consistency requirement

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

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

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

and hence the numerical flux is given by

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

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

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

and thus the numerical flux is given by

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

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

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

Here $\mathbf{Q}(\vec{q}^{\,l}_{i+\frac{1}{2}}, \vec{q}^{\,r}_{i+\frac{1}{2}})$ 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):

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

where

$$\vec{q}^{\,l}_{i+\frac{1}{2}} = \begin{pmatrix} h^l_{i+\frac{1}{2}} \\ h^l_{i+\frac{1}{2}} u^l_{i+\frac{1}{2}} \end{pmatrix}, \quad \vec{q}^{\,r}_{i+\frac{1}{2}} = \begin{pmatrix} h^r_{i+\frac{1}{2}} \\ h^r_{i+\frac{1}{2}} u^r_{i+\frac{1}{2}} \end{pmatrix}, \quad \vec{q}^{\,*}_{i+\frac{1}{2}} = \begin{pmatrix} \tilde{h}_{i+\frac{1}{2}} \\ \tilde{h}_{i+\frac{1}{2}} \tilde{u}_{i+\frac{1}{2}} \end{pmatrix} \tag{3.55}$$

and

$$\tilde{u}_{i+\frac{1}{2}} := \frac{\sqrt{h^l_{i+\frac{1}{2}}}u^l_{i+\frac{1}{2}} + \sqrt{h^r_{i+\frac{1}{2}}}u^r_{i+\frac{1}{2}}}{\sqrt{h^l_{i+\frac{1}{2}}} + \sqrt{h^r_{i+\frac{1}{2}}}}, \quad \tilde{h}_{i+\frac{1}{2}} := \frac{h^l_{i+\frac{1}{2}} + h^r_{i+\frac{1}{2}}}{2}, \quad \tilde{c}_{i+\frac{1}{2}} := \sqrt{g\tilde{h}_{i+\frac{1}{2}}}. \tag{3.56}$$

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

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

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

$$x^l \leq TS^l \leq 0, \quad x^r \geq TS^r \geq 0, \tag{3.58}$$

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

$$\int_{x^l}^{x^r} \vec{q}(x,T)\,\mathrm{d}x - \int_{x^l}^{x^r} \vec{q}(x,0)\,\mathrm{d}x + \int_0^T \vec{F}(\vec{q}(x^r,t))\,\mathrm{d}t - \int_0^T \vec{F}(\vec{q}(x^l,t))\,\mathrm{d}t = \vec{0}, \tag{3.59}$$

which can be rewritten to

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

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

$$\int_{x^l}^{x^r} \vec{q}(x,T)\,\mathrm{d}x = \int_{x^l}^{TS^l} \vec{q}(x,T)\,\mathrm{d}x + \int_{TS^l}^{TS^r} \vec{q}(x,T)\,\mathrm{d}x + \int_{TS^r}^{x^r} \vec{q}(x,T)\,\mathrm{d}x, \tag{3.61}$$

which can be evaluated as

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

Finally, comparing (3.60) and (3.62) yields

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

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

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

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

$$\int_0^{x^r} \vec{q}(x,T)\,\mathrm{d}x - \int_0^{x^r} \vec{q}(x,0)\,\mathrm{d}x + \int_0^T \vec{F}(\vec{q}(x^r,t))\,\mathrm{d}t - \int_0^T \vec{F}(\vec{q}(0,t))\,\mathrm{d}t = \vec{0}, \tag{3.65}$$

which can be rewritten to

$$\int_0^T \vec{F}(\vec{q}(0,t))\,\mathrm{d}t = \int_0^{TS^r} \vec{q}(x,T)\,\mathrm{d}x + \int_{TS^r}^{x^r} \vec{q}(x,T)\,\mathrm{d}x - \int_0^{x^r} \vec{q}(x,0)\,\mathrm{d}x + \int_0^T \vec{F}(\vec{q}(x^r,t))\,\mathrm{d}t, \tag{3.66}$$

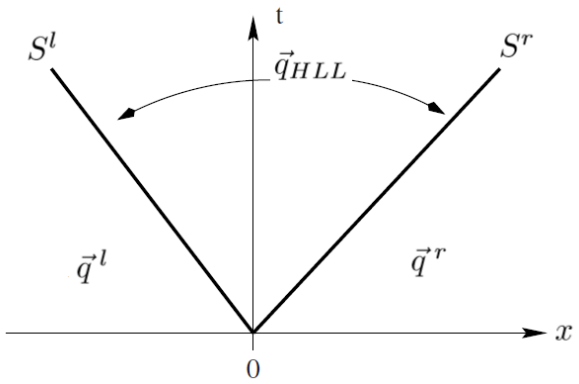

Figure 3.4: Sketch of the two waves of speeds S^l and S^r used in the HLL solver.

which evaluates to

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

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

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

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

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

and hence the numerical flux is given by

$$\vec{\mathcal{F}}_{\text{HLL}}(\vec{q}^{\,l}_{i+\frac{1}{2}}, \vec{q}^{\,r}_{i+\frac{1}{2}}) = \begin{cases} \vec{F}(\vec{q}^{\,l}_{i+\frac{1}{2}}) & \text{if } 0 \le S^l_{i+\frac{1}{2}}, \\ \dfrac{S^r_{i+\frac{1}{2}}\vec{F}(\vec{q}^{\,l}_{i+\frac{1}{2}}) - S^l_{i+\frac{1}{2}}\vec{F}(\vec{q}^{\,r}_{i+\frac{1}{2}}) + S^l_{i+\frac{1}{2}}S^r_{i+\frac{1}{2}}(\vec{q}^{\,r}_{i+\frac{1}{2}} - \vec{q}^{\,l}_{i+\frac{1}{2}})}{S^r_{i+\frac{1}{2}} - S^l_{i+\frac{1}{2}}} & \text{if } S^l_{i+\frac{1}{2}} \le 0 \le S^r_{i+\frac{1}{2}}, \\ \vec{F}(\vec{q}^{\,r}_{i+\frac{1}{2}}) & \text{if } S^r_{i+\frac{1}{2}} \ge 0. \end{cases} \tag{3.70}$$

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

The HLL solver
Toro [20] suggests from experience to take

$$S^l_{i+\frac{1}{2}} = u^l_{i+\frac{1}{2}} - p^l_{i+\frac{1}{2}}\sqrt{gh^l_{i+\frac{1}{2}}}, \quad S^r_{i+\frac{1}{2}} = u^r_{i+\frac{1}{2}} + p^r_{i+\frac{1}{2}}\sqrt{gh^r_{i+\frac{1}{2}}}, \tag{3.71}$$

where we write

$$\vec{q}^{\,l}_{i+\frac{1}{2}} = \begin{bmatrix} h^l_{i+\frac{1}{2}} \\ h^l_{i+\frac{1}{2}}u^l_{i+\frac{1}{2}} \end{bmatrix}, \quad \vec{q}^{\,r}_{i+\frac{1}{2}} = \begin{bmatrix} h^r_{i+\frac{1}{2}} \\ h^r_{i+\frac{1}{2}}u^r_{i+\frac{1}{2}} \end{bmatrix}, \tag{3.55}$$

and we define

$$p^K_{i+\frac{1}{2}} := \begin{cases} \dfrac{1}{h^K_{i+\frac{1}{2}}}\sqrt{\frac{1}{2}\hat{h}_{i+\frac{1}{2}}(\hat{h}_{i+\frac{1}{2}} + h^K_{i+\frac{1}{2}})} & \text{if } \hat{h}_{i+\frac{1}{2}} > h^K_{i+\frac{1}{2}}, \\ 1 & \text{if } \hat{h}_{i+\frac{1}{2}} \le h^K_{i+\frac{1}{2}}, \end{cases} \tag{3.72}$$

where

$$\hat{h}_{i+\frac{1}{2}} := \frac{1}{g}\left[\frac{1}{2}(\sqrt{gh^l_{i+\frac{1}{2}}} + \sqrt{gh^r_{i+\frac{1}{2}}}) + \frac{1}{4}(u^l_{i+\frac{1}{2}} - u^r_{i+\frac{1}{2}})\right]^2. \tag{3.73}$$

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

$$S^l_{i+\frac{1}{2}} = \min_{p=1,2}(\min(\lambda^p_i, \tilde{\lambda}^p_{i+\frac{1}{2}})), \quad S^r_{i+\frac{1}{2}} = \max_{p=1,2}(\max(\lambda^p_{i+1}, \tilde{\lambda}^p_{i+\frac{1}{2}})), \tag{3.74}$$

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

$$S^l_{i+\frac{1}{2}} = \min_{p=1,2}(\min(\lambda^{l,p}_i, \tilde{\lambda}^p_{i+\frac{1}{2}})), \quad S^r_{i+\frac{1}{2}} = \max_{p=1,2}(\max(\lambda^{r,p}_{i+1}, \tilde{\lambda}^p_{i+\frac{1}{2}})), \tag{3.75}$$

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

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:

$$S_{i,2} = \frac{1}{\Delta x}\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} -gh(x,t)b_x(x)\,\mathrm{d}x \approx -\frac{g(b_{i+1}-b_{i-1})}{2\Delta x}\left(h^r_{i-\frac{1}{2}} + h^l_{i+\frac{1}{2}}\right), \tag{3.76}$$

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

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

where we define the following function:

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

where we calculate

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

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

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

and thus we require for the time step

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

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 $I_{i,j}$ ($i = 1, ..., M$, $j = 1, , ..., N$) is given by

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

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

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

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

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

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

$$\begin{aligned} &\frac{\mathrm{d}}{\mathrm{d}t} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} \vec{q}(x,y,t)\, \mathrm{d}x\, \mathrm{d}y + \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} \Big(\vec{F}(\vec{q}(x_{i+\frac{1}{2}},y,t)) - \vec{F}(\vec{q}(x_{i-\frac{1}{2}},y,t)) \Big)\, \mathrm{d}y \\ &+ \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \Big(\vec{G}(\vec{q}(x,y_{i+\frac{1}{2}},t)) - \vec{G}(\vec{q}(x,y_{i-\frac{1}{2}},t)) \Big)\, \mathrm{d}x = \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} \vec{S}(\vec{q}(x,y,t))\, \mathrm{d}x\, \mathrm{d}y. \end{aligned} \tag{4.4}$$

The above can be rewritten into the finite volume equation

$$\frac{\mathrm{d}\vec{q}_{i,j}}{\mathrm{d}t} + \frac{1}{\Delta x} \Big(\vec{F}_{i+\frac{1}{2},j} - \vec{F}_{i-\frac{1}{2},j} \Big) + \frac{1}{\Delta y} \Big(\vec{G}_{i,j+\frac{1}{2}} - \vec{G}_{i,j-\frac{1}{2}} \Big) = \vec{S}_{i,j}, \tag{4.5}$$

where we define the following averages:

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

In this case, we have that $\vec{q}_{i,j}$ is the *cell average*. In our numerical method, we assume the numerical solution $\vec{q}_{i,j}$ 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 $\vec{F}_{\frac{1}{2},j}, \vec{F}_{\frac{3}{2},j}, ..., \vec{F}_{M+\frac{1}{2},j}$ $(j = 1, 2, ..., N)$ in the finite volume equation (4.5), we also need to evaluate the fluxes $\vec{G}_{i,\frac{1}{2}}, \vec{G}_{i,\frac{3}{2}}, ..., \vec{G}_{i,N+\frac{1}{2}}$ $(i = 1, 2, ..., M)$.

In order to evaluate these fluxes, one needs the numerical solutions of $\vec{q}$ at both $(x_{\frac{1}{2}}, y_j), (x_{\frac{3}{2}}, y_j), ..., (x_{M+\frac{1}{2}}, y_j)$ $(j = 1, 2, ..., N)$, which we denote by $\vec{q}_{\frac{1}{2},j}, \vec{q}_{\frac{3}{2},j}, ..., \vec{q}_{M+\frac{1}{2},j}$, and $(x_i, y_{\frac{1}{2}}), (x_i, y_{\frac{3}{2}}, y_j), ..., (x_i, y_{M+\frac{1}{2}})$ $(i = 1, 2, ..., M)$, which we denote by $\vec{q}_{i,\frac{1}{2}}, \vec{q}_{i,\frac{3}{2}}, ..., \vec{q}_{i,M+\frac{1}{2}}$. 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

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

where $\vec{q}^{\,l}_{i+\frac{1}{2},j}$ is the numerical solution to the left of the right boundary of cell $I_{i,j}$ and $\vec{q}^{\,r}_{i+\frac{1}{2},j}$ 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 $\vec{\mathcal{F}}(\vec{q}^{\,l}_{i+\frac{1}{2},j}, \vec{q}^{\,r}_{i+\frac{1}{2},j})$ and $\vec{\mathcal{G}}(\vec{q}^{\,l}_{i,j+\frac{1}{2}}, \vec{q}^{\,r}_{i,j+\frac{1}{2}})$ are the flux evaluation functions which are yet to be chosen.

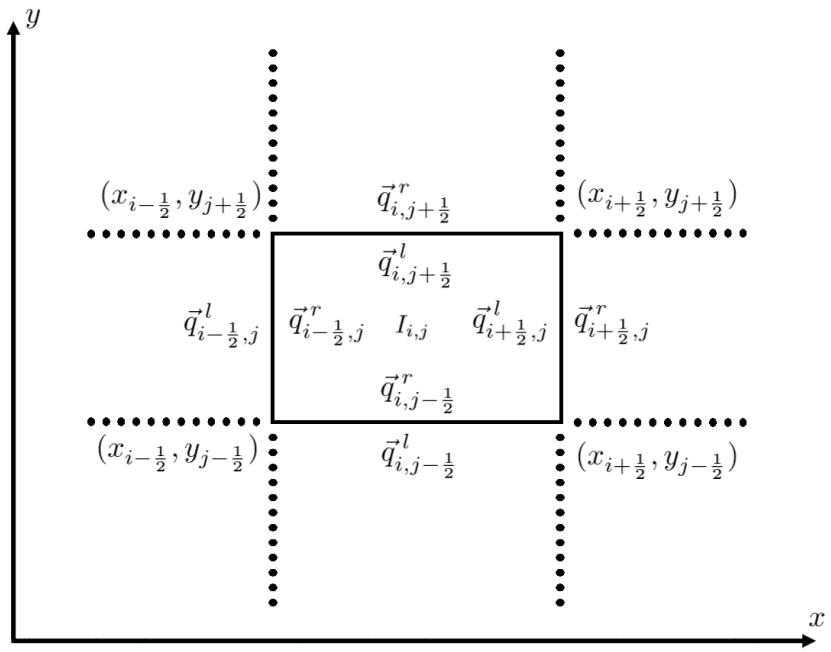

Figure 4.1: Top-down view of a two-dimensional cell.

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

$$\vec{q}^{\,l}_{i+\frac{1}{2},j} = \vec{q}_{i,j}, \tag{4.8a}$$
$$\vec{q}^{\,r}_{i+\frac{1}{2},j} = \vec{q}_{i+1,j}, \tag{4.8b}$$
$$\vec{q}^{\,l}_{i,j+\frac{1}{2}} = \vec{q}_{i,j}, \tag{4.8c}$$
$$\vec{q}^{\,r}_{i,j+\frac{1}{2}} = \vec{q}_{i,j+1}, \tag{4.8d}$$

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

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

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

$$\vec{q}^{\,l}_{i+\frac{1}{2},j} = \vec{q}_{i,j} + \frac{1}{2}\phi(r^l_{i+\frac{1}{2},j})(\vec{q}_{i,j} - \vec{q}_{i-1,j}), \tag{4.10a}$$
$$\vec{q}^{\,r}_{i+\frac{1}{2},j} = \vec{q}_{i+1,j} + \frac{1}{2}\phi(r^r_{i+\frac{1}{2},j})(\vec{q}_{i+1,j} - \vec{q}_{i+2,j}), \tag{4.10b}$$
$$\vec{q}^{\,l}_{i,j+\frac{1}{2}} = \vec{q}_{i,j} + \frac{1}{2}\phi(r^l_{i,j+\frac{1}{2}})(\vec{q}_{i,j} - \vec{q}_{i,j-1}), \tag{4.10c}$$
$$\vec{q}^{\,r}_{i,j+\frac{1}{2}} = \vec{q}_{i,j+1} + \frac{1}{2}\phi(r^r_{i,j+\frac{1}{2}})(\vec{q}_{i,j+1} - \vec{q}_{i,j+2}), \tag{4.10d}$$

where

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

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 $\vec{\mathcal{F}}(\vec{q}^{\,l}_{i+\frac{1}{2},j}, \vec{q}^{\,r}_{i+\frac{1}{2},j})$ and $\vec{\mathcal{G}}(\vec{q}^{\,l}_{i,j+\frac{1}{2}}, \vec{q}^{\,r}_{i,j+\frac{1}{2}})$. 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

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

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

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

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

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

where Λ^* 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, Σ^* 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

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

where

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

and

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

where

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

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

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

where

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

and

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

Here it holds that

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

Finally, we take the flux evaluation functions as

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

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

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

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

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

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

where Λ 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

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

where

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

and

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

where

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

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

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

where

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

and

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

Again it holds that

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

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

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

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

where

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

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

and the consistency requirement

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

is always satisfied.

The midpoint rule

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

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

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

and hence the flux evaluation function is given by

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

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

The trapezoidal rule

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

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

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

and thus the numerical flux is given by

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

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

Roe's method

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

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

where

$$\vec{q}^{\,l}_{i+\frac{1}{2},j} = \begin{pmatrix} h^l_{i+\frac{1}{2},j} \\ h^l_{i+\frac{1}{2},j}u^l_{i+\frac{1}{2},j} \\ h^l_{i+\frac{1}{2},j}v^l_{i+\frac{1}{2},j} \end{pmatrix}, \quad \vec{q}^{\,r}_{i+\frac{1}{2},j} = \begin{pmatrix} h^r_{i+\frac{1}{2},j} \\ h^r_{i+\frac{1}{2},j}u^r_{i+\frac{1}{2},j} \\ h^r_{i+\frac{1}{2},j}v^r_{i+\frac{1}{2},j} \end{pmatrix}, \quad \vec{q}^{\,*}_{i+\frac{1}{2},j} = \begin{pmatrix} \tilde{h}_{i+\frac{1}{2},j} \\ \tilde{h}_{i+\frac{1}{2},j}\tilde{u}_{i+\frac{1}{2},j} \\ \tilde{h}_{i+\frac{1}{2},j}\tilde{v}_{i+\frac{1}{2},j} \end{pmatrix}, \tag{4.42}$$

and

$$\tilde{u}_{i+\frac{1}{2},j} := \frac{\sqrt{h^l_{i+\frac{1}{2},j}}u^l_{i+\frac{1}{2},j} + \sqrt{h^r_{i+\frac{1}{2},j}}u^r_{i+\frac{1}{2},j}}{\sqrt{h^l_{i+\frac{1}{2},j}} + \sqrt{h^r_{i+\frac{1}{2},j}}}, \quad \tilde{v}_{i+\frac{1}{2},j} := \frac{\sqrt{h^l_{i+\frac{1}{2},j}}u^l_{i+\frac{1}{2},j} + \sqrt{h^r_{i+\frac{1}{2},j}}u^r_{i+\frac{1}{2},j}}{\sqrt{h^l_{i+\frac{1}{2},j}} + \sqrt{h^r_{i+\frac{1}{2},j}}},$$
$$\tilde{h}_{i+\frac{1}{2},j} := \frac{h^l_{i+\frac{1}{2},j} + h^r_{i+\frac{1}{2},j}}{2}, \quad \tilde{c}_{i+\frac{1}{2},j} := \sqrt{g\tilde{h}_{i+\frac{1}{2},j}}. \tag{4.43}$$

Similarly, we choose

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

where

$$\vec{q}^{\,l}_{i,j+\frac{1}{2}} = \begin{pmatrix} h^l_{i,j+\frac{1}{2}} \\ h^l_{i,j+\frac{1}{2}} u^l_{i,j+\frac{1}{2}} \\ h^l_{i,j+\frac{1}{2}} v^l_{i,j+\frac{1}{2}} \end{pmatrix}, \quad \vec{q}^{\,r}_{i,j+\frac{1}{2}} = \begin{pmatrix} h^r_{i,j+\frac{1}{2}} \\ h^r_{i,j+\frac{1}{2}} u^r_{i,j+\frac{1}{2}} \\ h^r_{i,j+\frac{1}{2}} v^r_{i,j+\frac{1}{2}} \end{pmatrix}, \quad \vec{q}^{\,*}_{i+\frac{1}{2},j} = \begin{pmatrix} \tilde{h}_{i,j+\frac{1}{2}} \\ \tilde{h}_{i,j+\frac{1}{2}} \tilde{u}_{i,j+\frac{1}{2}} \\ \tilde{h}_{i,j+\frac{1}{2}} \tilde{v}_{i,j+\frac{1}{2}} \end{pmatrix}, \tag{4.45}$$

and

$$\begin{aligned} \tilde{u}_{i,j+\frac{1}{2},j} &:= \frac{\sqrt{h^l_{i,j+\frac{1}{2}}}u^l_{i,j+\frac{1}{2}} + \sqrt{h^r_{i,j+\frac{1}{2}}}u^r_{i,j+\frac{1}{2}}}{\sqrt{h^l_{i,j+\frac{1}{2}}} + \sqrt{h^r_{i,j+\frac{1}{2}}}}, \quad \tilde{v}_{i,j+\frac{1}{2}} := \frac{\sqrt{h^l_{i,j+\frac{1}{2}}}u^l_{i,j+\frac{1}{2}} + \sqrt{h^r_{i,j+\frac{1}{2}}}u^r_{i,j+\frac{1}{2}}}{\sqrt{h^l_{i,j+\frac{1}{2}}} + \sqrt{h^r_{i,j+\frac{1}{2}}}}, \\ \tilde{h}_{i,j+\frac{1}{2}} &:= \frac{h^l_{i,j+\frac{1}{2}} + h^r_{i,j+\frac{1}{2}}}{2}, \quad \tilde{c}_{i,j+\frac{1}{2}} := \sqrt{g\tilde{h}_{i,j+\frac{1}{2}}}. \end{aligned} \tag{4.46}$$

4.3.3 Source term discretization

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

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

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

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:

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

where we define the following function:

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

where we calculate

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

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

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

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

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

and thus we require for the time step

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

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

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

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 $t = 3$.

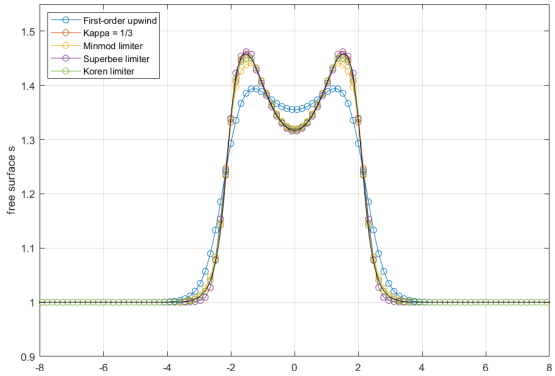

Figure 5.1: One-dimensional water hill on a flat bottom. View of the computed free surface at $t = 1$.

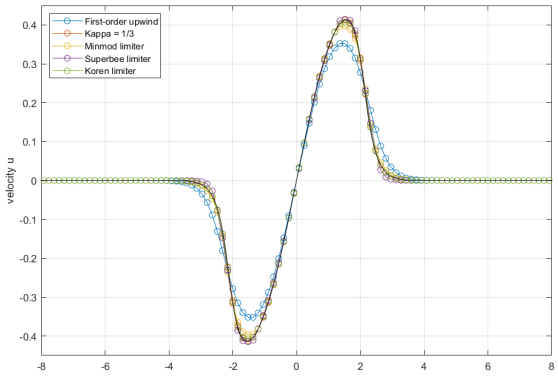

Figure 5.2: One-dimensional water hill on a flat bottom. View of the computed velocity at $t = 1$.

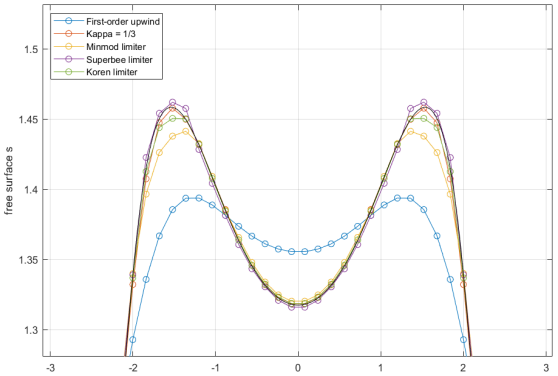

Figure 5.3: One-dimensional water hill on a flat bottom. View of the crest of both the leftward and rightward propagating waves of the computed free surface at $t = 1$.

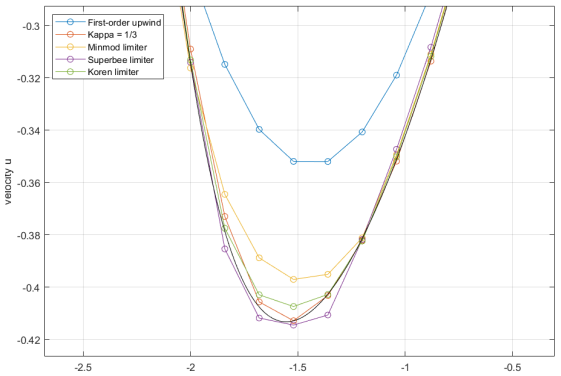

Figure 5.4: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 1$.

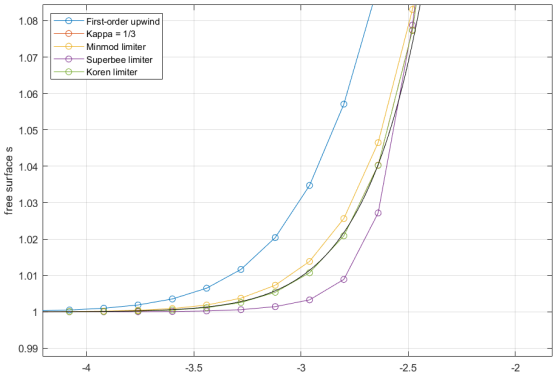

Figure 5.5: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 1$.

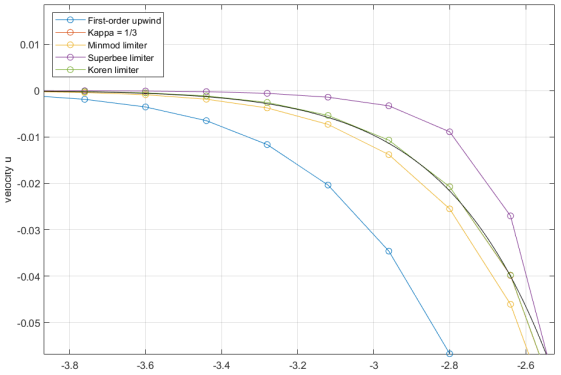

Figure 5.6: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 1$.

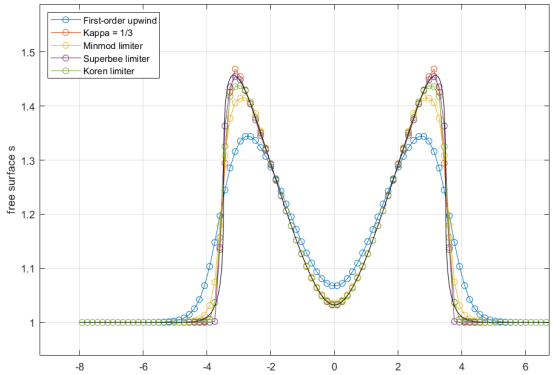

Figure 5.7: One-dimensional water hill on a flat bottom. View of the computed free surface at $t = 2$.

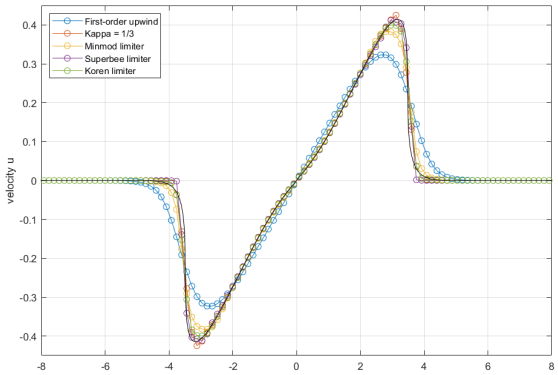

Figure 5.8: One-dimensional water hill on a flat bottom. View of the computed velocity at $t = 2$.

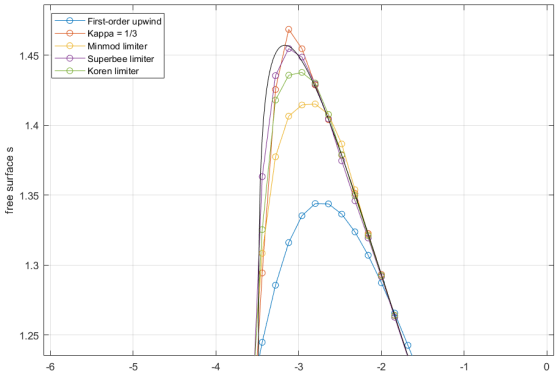

Figure 5.9: One-dimensional water hill on a flat bottom. View of the crest the leftward propagating wave of the computed free surface at $t = 2$.

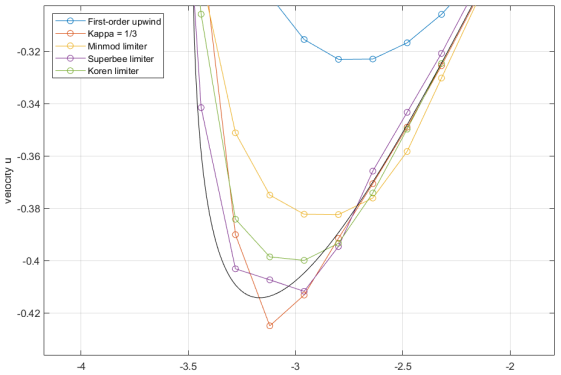

Figure 5.10: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 2$.

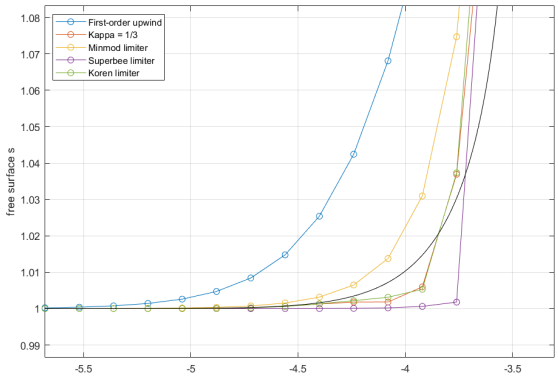

Figure 5.11: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 2$.

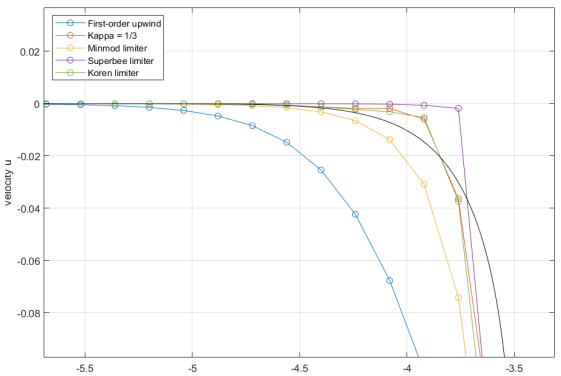

Figure 5.12: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 2$.

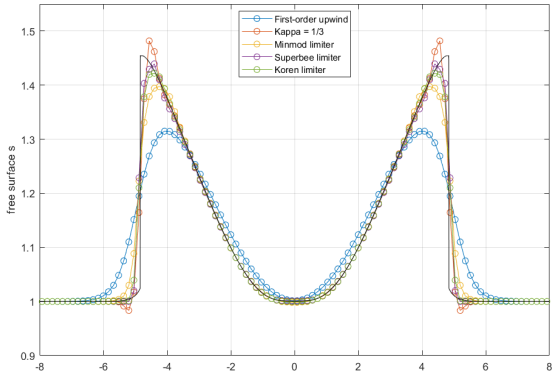

Figure 5.13: One-dimensional water hill on a flat bottom. View of the computed free surface at $t = 3$.

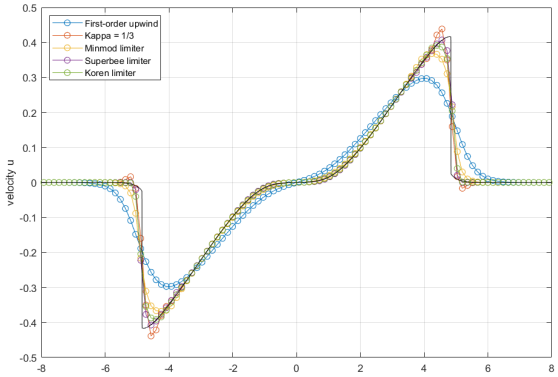

Figure 5.14: One-dimensional water hill on a flat bottom. View of the computed velocity at $t = 3$.

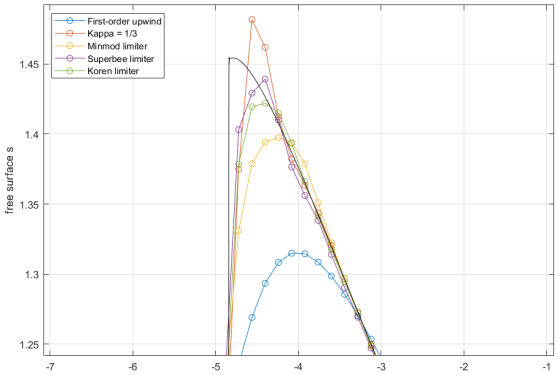

Figure 5.15: One-dimensional water hill on a flat bottom. View of the crest the leftward propagating wave of the computed free surface at $t = 3$.

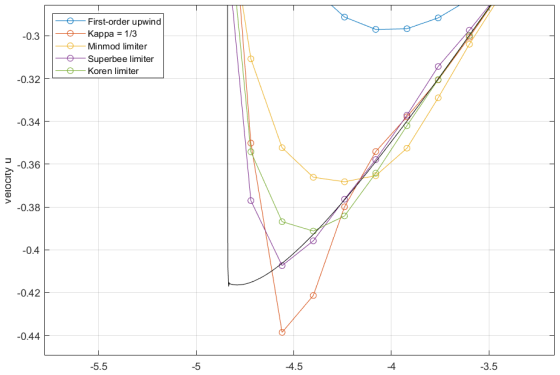

Figure 5.16: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 3$.

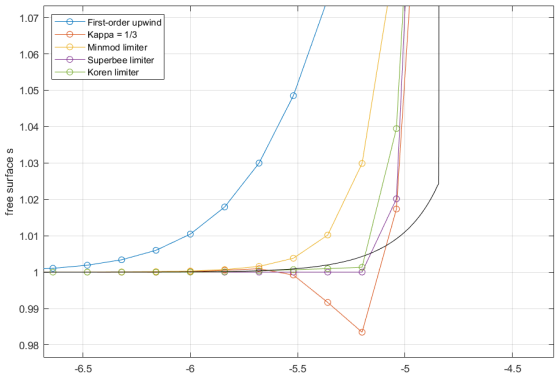

Figure 5.17: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 3$.

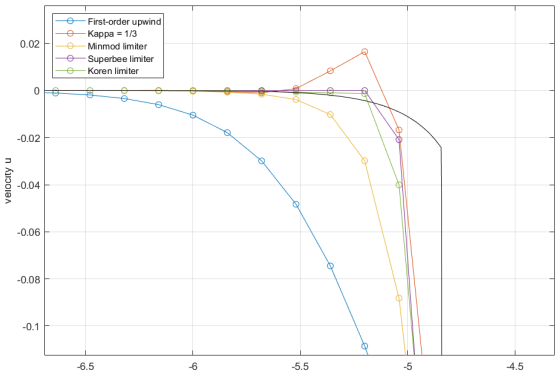

Figure 5.18: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 3$.

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.

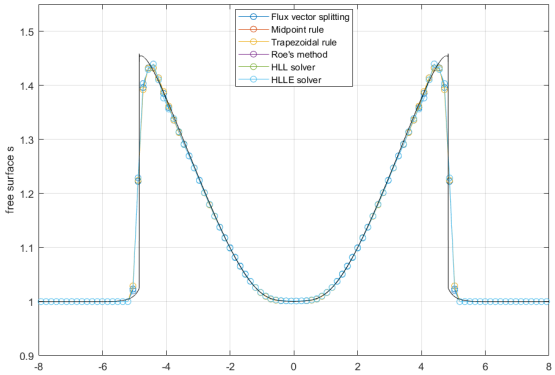

Figure 5.19: One-dimensional water hill on a flat bottom. View of the computed free surface at $t = 3$ computed using the Superbee limiter.

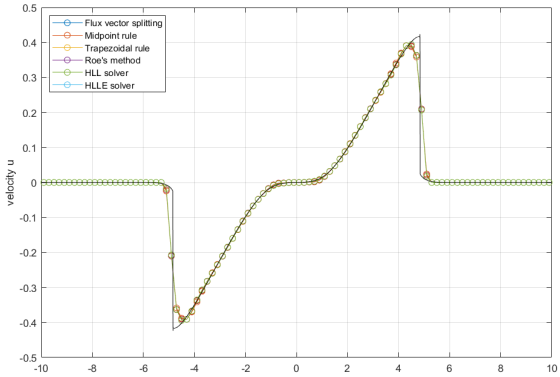

Figure 5.20: One-dimensional water hill on a flat bottom. View of the computed velocity at $t = 3$ computed using the Superbee limiter.

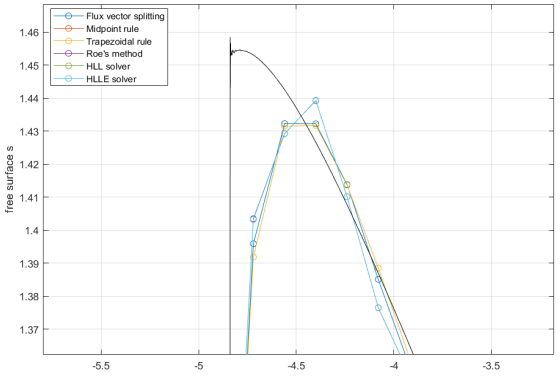

Figure 5.21: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Superbee limiter.

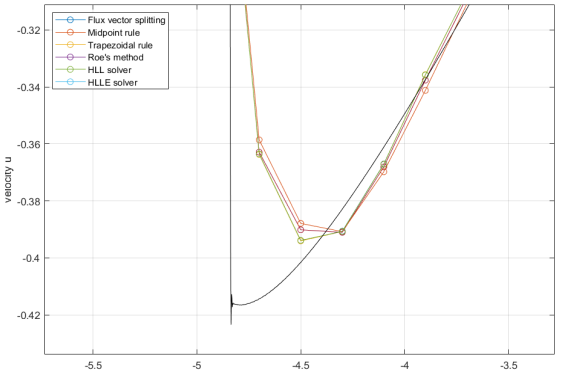

Figure 5.22: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Superbee limiter.

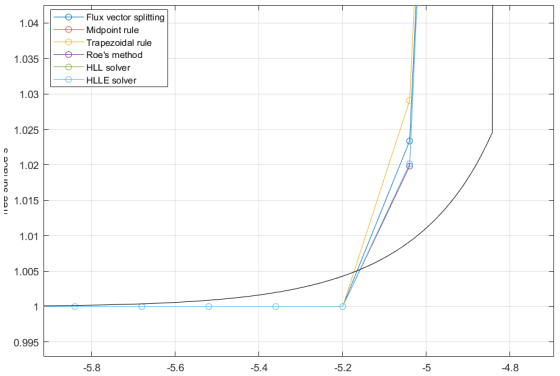

Figure 5.23: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Superbee limiter.

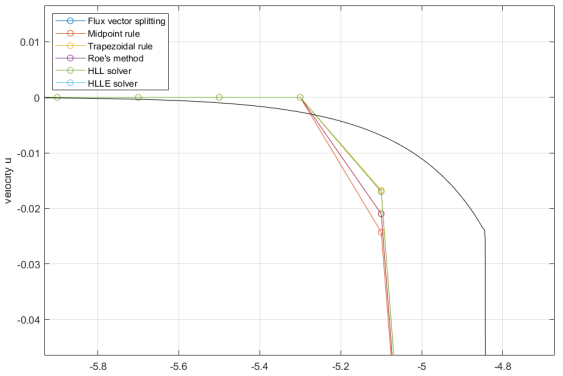

Figure 5.24: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Superbee limiter.

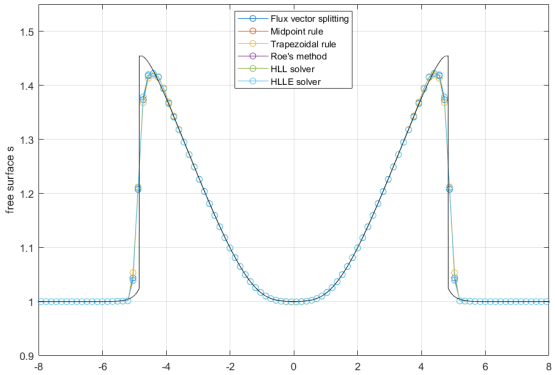

Figure 5.25: One-dimensional water hill on a flat bottom. View of the computed free surface at $t = 3$ computed using the Koren limiter.

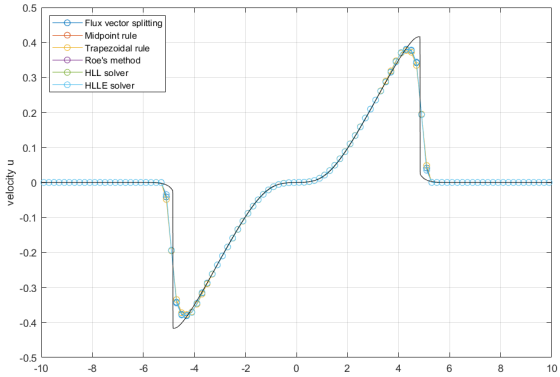

Figure 5.26: One-dimensional water hill on a flat bottom. View of the computed velocity at $t = 3$ computed using the Koren limiter.

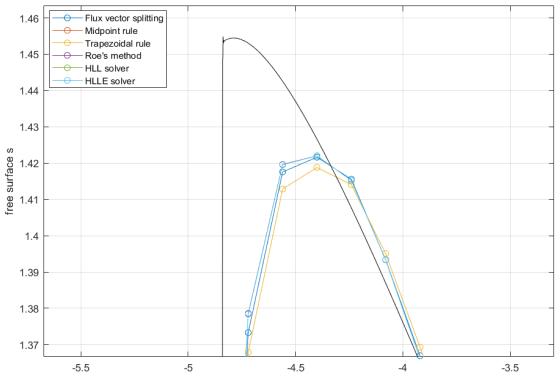

Figure 5.27: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Koren limiter.

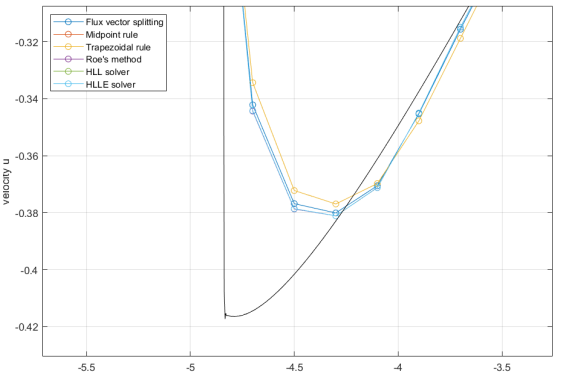

Figure 5.28: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Koren limiter.

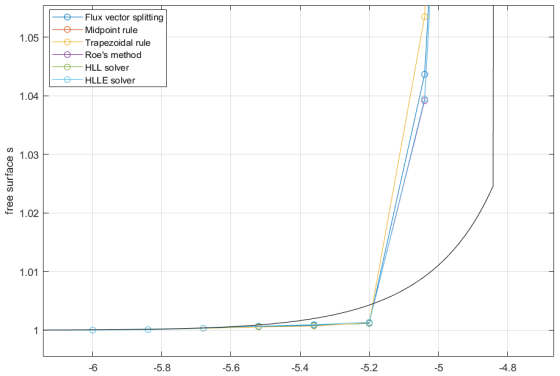

Figure 5.29: One-dimensional water hill on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Koren limiter.

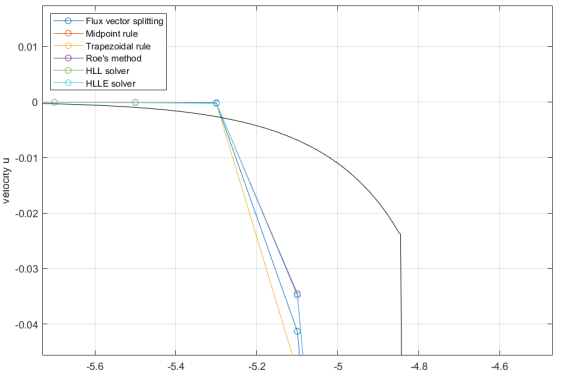

Figure 5.30: One-dimensional water hill on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Koren limiter.

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 ($i = 1, ..., N$) 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:

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

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 HLLE 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
HLLE 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

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

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 phenomenom dissapears 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 under performs 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.

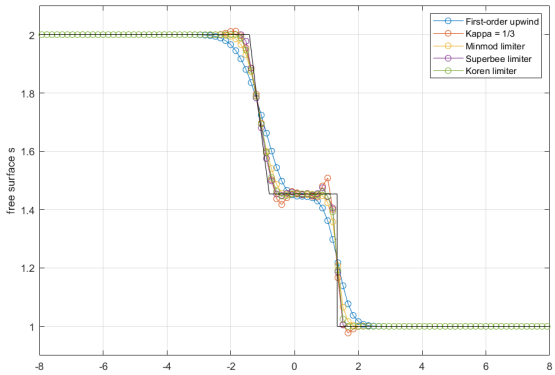

Figure 5.31: One-dimensional dam-break problem on a flat bottom. View of the computed free surface at $t = 1$.

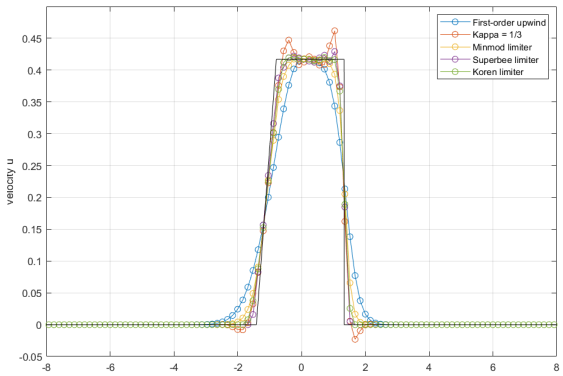

Figure 5.32: One-dimensional dam-break problem on a flat bottom. View of the computed velocity at $t = 1$.

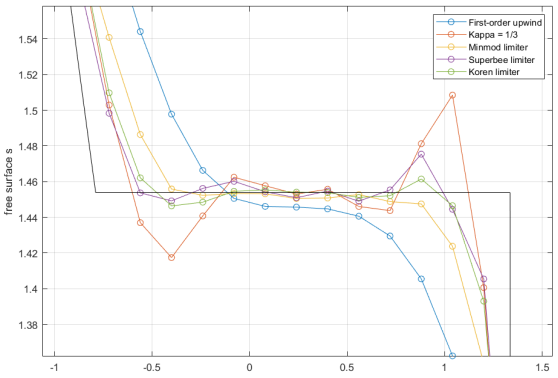

Figure 5.33: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave and the crest of the rightward propagating wave of the computed free surface at $t = 1$.

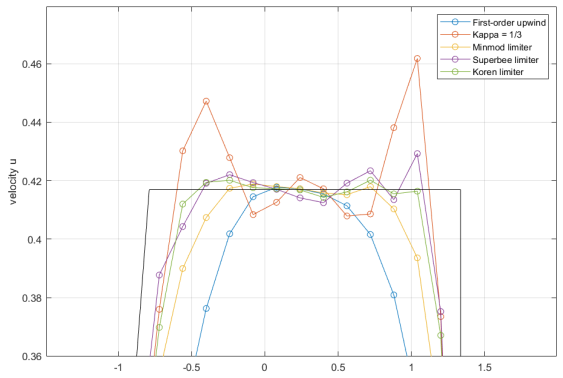

Figure 5.34: One-dimensional dam-break problem on a flat bottom. View of the crest of both the leftward and rightward propagating waves of the computed velocity at $t = 1$.

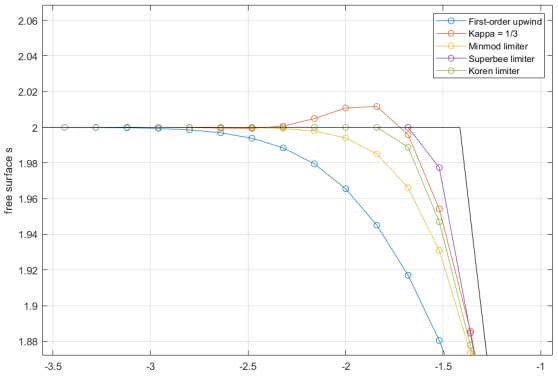

Figure 5.35: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 1$.

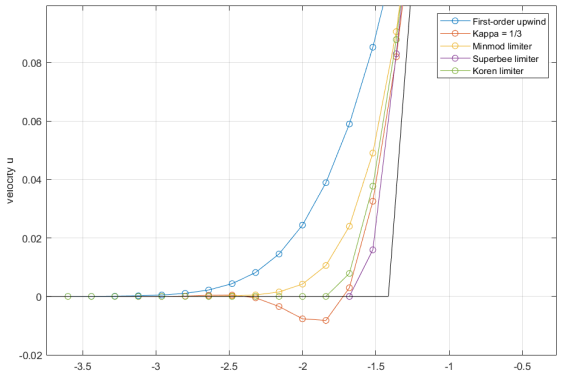

Figure 5.36: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 1$.

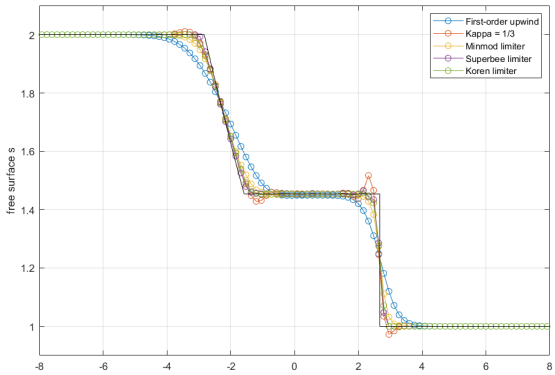

Figure 5.37: One-dimensional dam-break problem on a flat bottom. View of the computed free surface at $t = 2$.

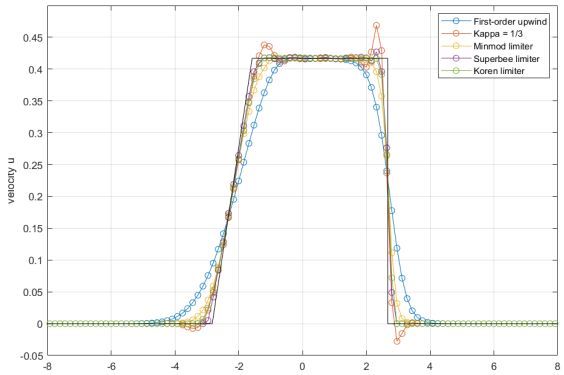

Figure 5.38: One-dimensional dam-break problem on a flat bottom. View of the computed velocity at $t = 2$.

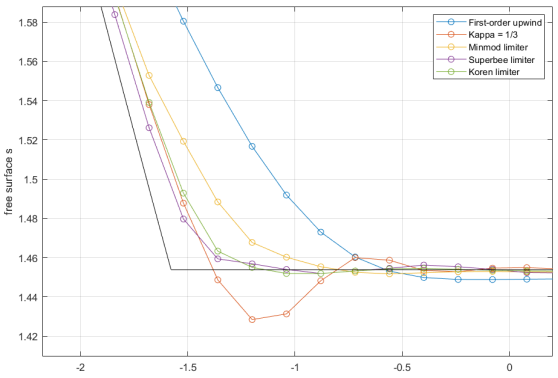

Figure 5.39: One-dimensional dam-break problem on a flat bottom. View of the trough the leftward propagating wave of the computed free surface at $t = 2$.

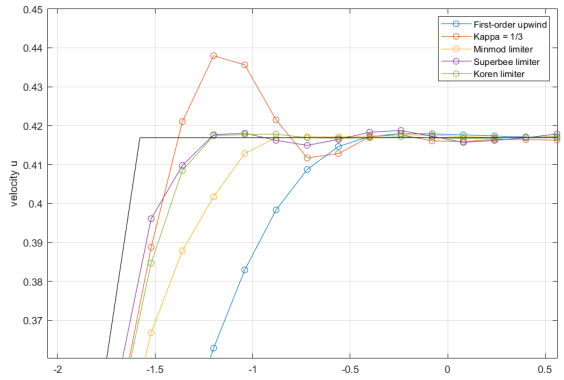

Figure 5.40: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 2$.

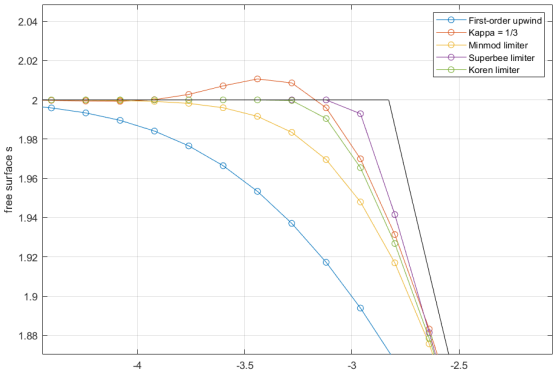

Figure 5.41: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 2$.

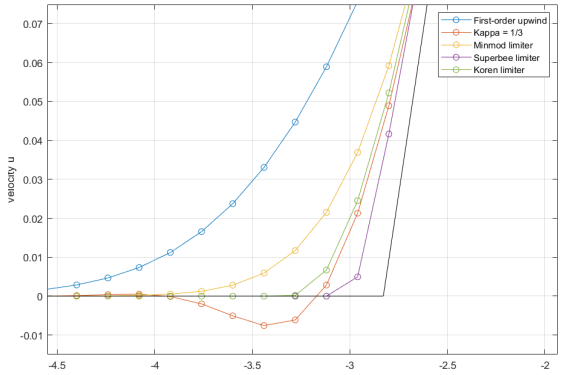

Figure 5.42: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 2$.

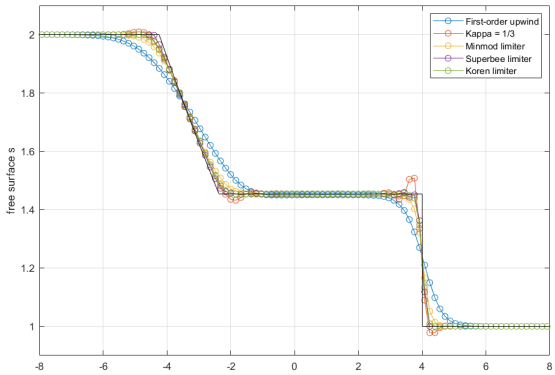

Figure 5.43: One-dimensional dam-break problem on a flat bottom. View of the computed free surface at $t = 3$.

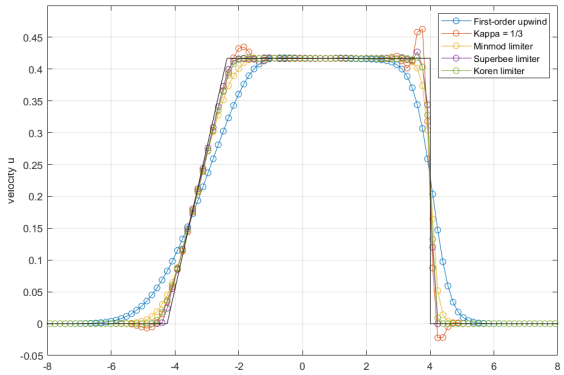

Figure 5.44: One-dimensional dam-break problem on a flat bottom. View of the computed velocity at $t = 3$.

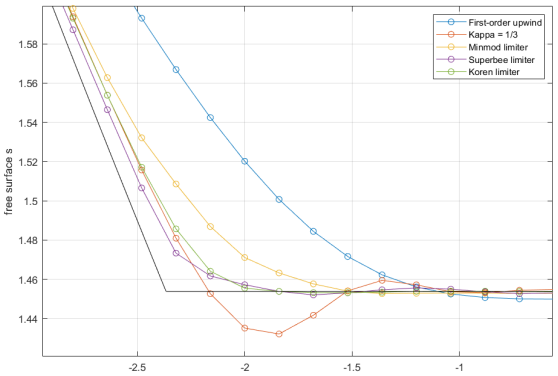

Figure 5.45: One-dimensional dam-break problem on a flat bottom. View of the trough the leftward propagating wave of the computed free surface at $t = 3$.

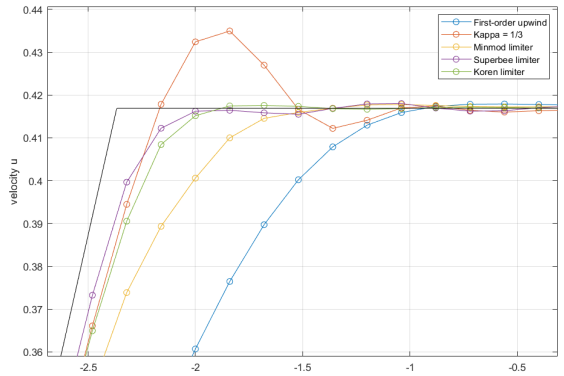

Figure 5.46: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 3$.

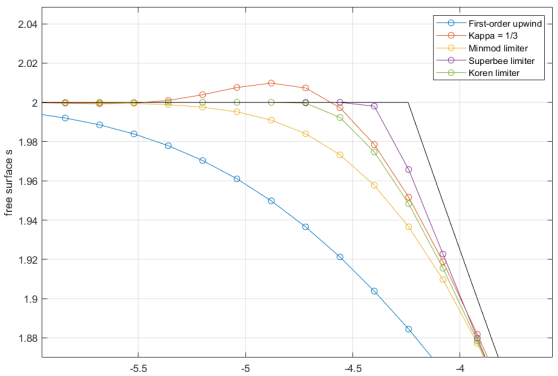

Figure 5.47: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 3$.

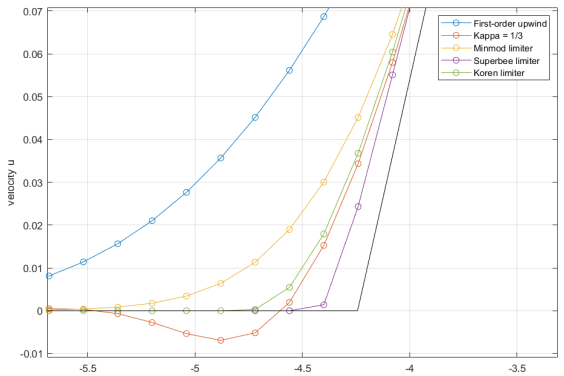

Figure 5.48: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 3$.

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 ($i = 1, ..., N$) 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 ($i = 1, ..., N$). We consider the L_1, L_2 and L_∞ errors of the computed free surface and velocity, given by

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

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

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

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.

Tables 5.5shows 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	HLLE 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 $t = 3$.

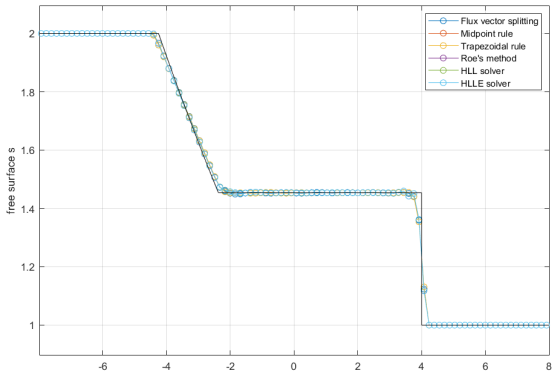

Figure 5.49: One-dimensional dam-break problem on a flat bottom. View of the computed free surface at $t = 3$ computed using the Superbee limiter.

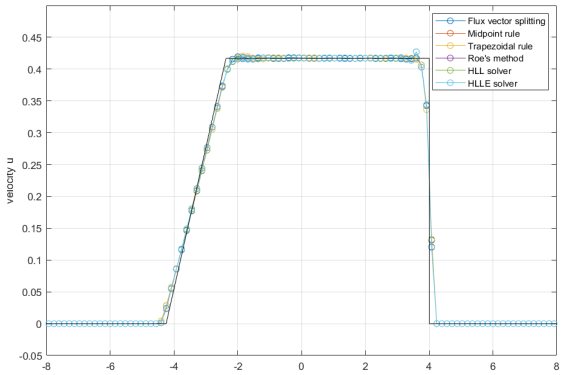

Figure 5.50: One-dimensional dam-break problem on a flat bottom. View of the computed velocity at $t = 3$ computed using the Superbee limiter.

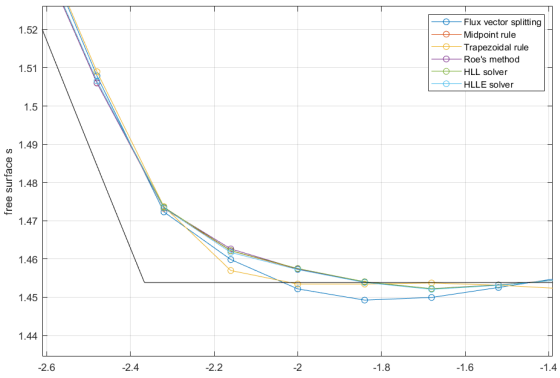

Figure 5.51: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Superbee limiter.

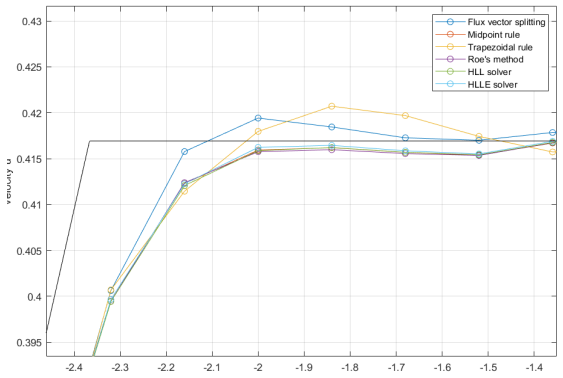

Figure 5.52: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Superbee limiter.

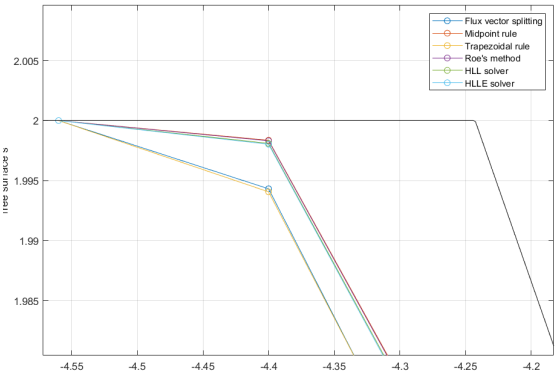

Figure 5.53: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Superbee limiter.

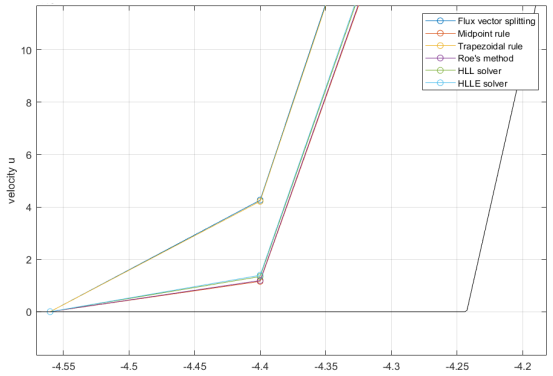

Figure 5.54: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Superbee limiter.

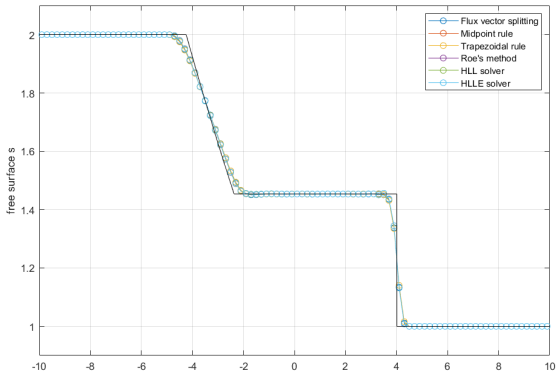

Figure 5.55: One-dimensional dam-break problem on a flat bottom. View of the computed free surface at $t = 3$ computed using the Koren limiter.

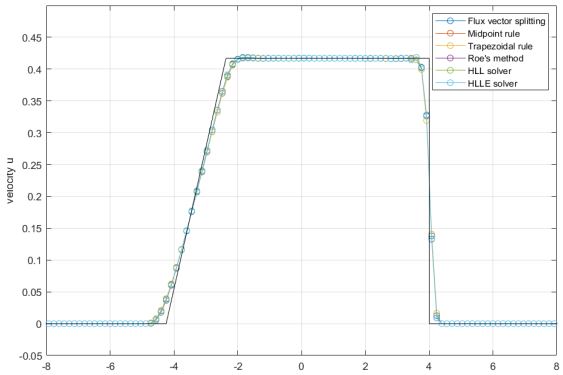

Figure 5.56: One-dimensional dam-break problem on a flat bottom. View of the computed velocity at $t = 3$ computed using the Koren limiter.

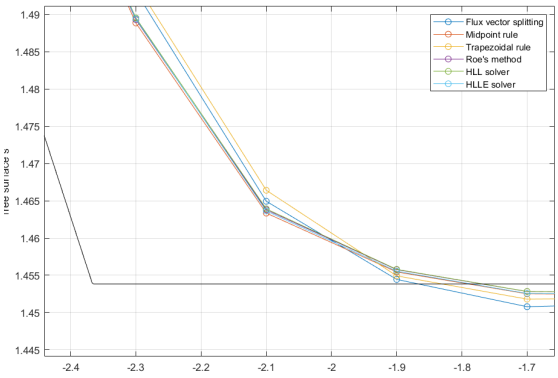

Figure 5.57: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Koren limiter.

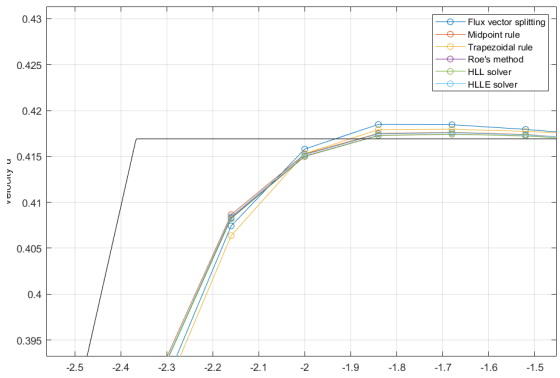

Figure 5.58: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Koren limiter.

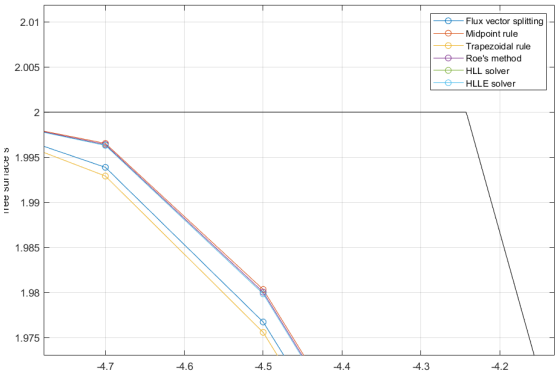

Figure 5.59: One-dimensional dam-break problem on a flat bottom. View of the crest of the leftward propagating wave of the computed free surface at $t = 3$ computed using the Koren limiter.

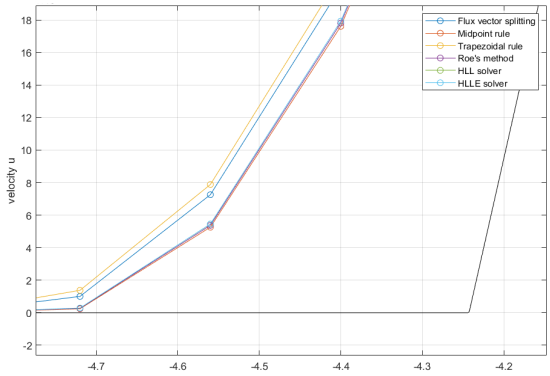

Figure 5.60: One-dimensional dam-break problem on a flat bottom. View of the trough of the leftward propagating wave of the computed velocity at $t = 3$ computed using the Koren limiter.

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 physicial phenomenom 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 heighest and lowest height values in one part of the V-shaped wave.

As expected, since the Froude-number $Fr = \dfrac{u}{\sqrt{gh}}$ 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 $t = 15$.

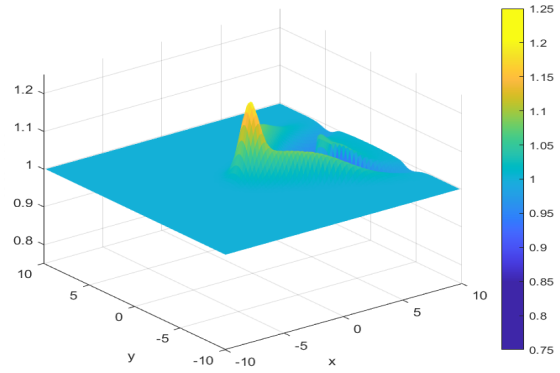

Figure 5.61: Two-dimensional supercritical flow over a bump. View of the computed free surface at $t = 5$ computed using the Koren limiter and Roe's method.

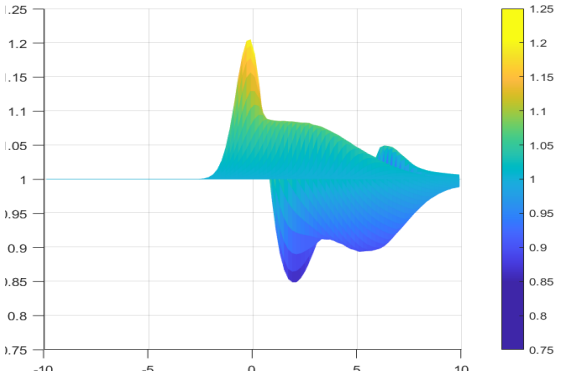

Figure 5.62: Two-dimensional supercritical flow over a bump. Side view of the computed free surface at $t = 5$ computed using the Koren limiter and Roe's method.

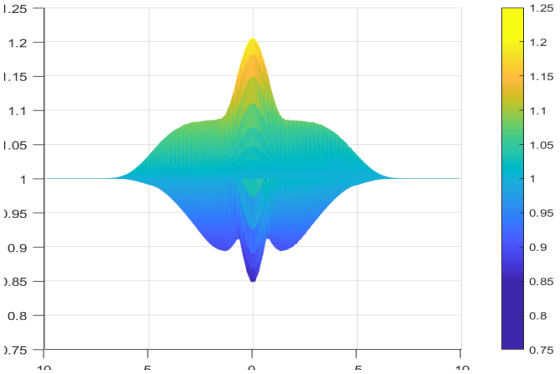

Figure 5.63: Two-dimensional supercritical flow over a bump. Front view of the computed free surface at $t = 5$ computed using the Koren limiter and Roe's method.

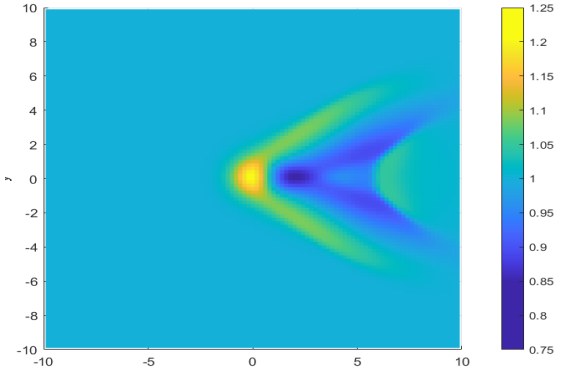

Figure 5.64: Two-dimensional supercritical flow over a bump. Top view of the computed free surface at $t = 5$ computed using the Koren limiter and Roe's method.

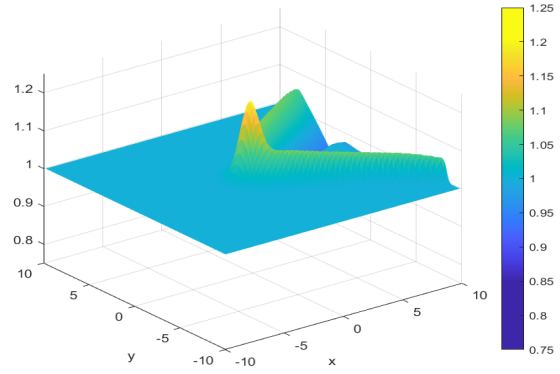

Figure 5.65: Two-dimensional supercritical flow over a bump. View of the computed free surface at $t = 10$ computed using the Koren limiter and Roe's method.

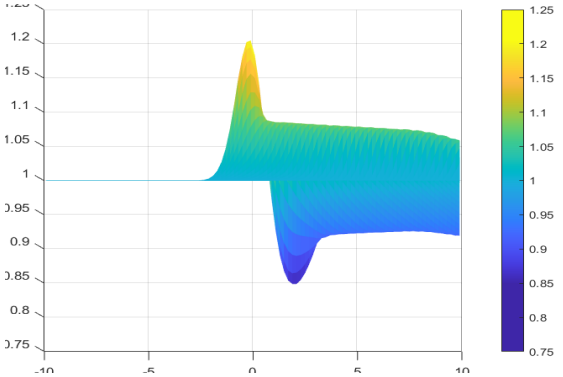

Figure 5.66: Two-dimensional supercritical flow over a bump. Side view of the computed free surface at $t = 10$ computed using the Koren limiter and Roe's method.

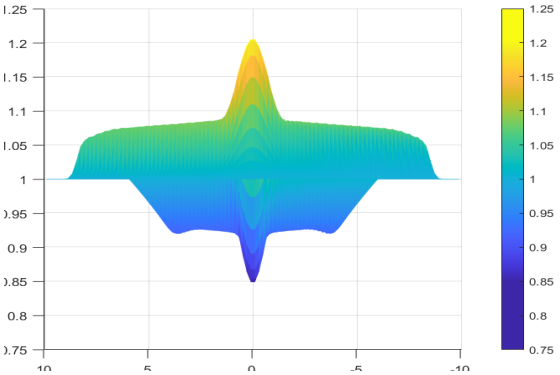

Figure 5.67: Two-dimensional supercritical flow over a bump. View of the computed free surface at $t = 10$ computed using the Koren limiter and Roe's method.

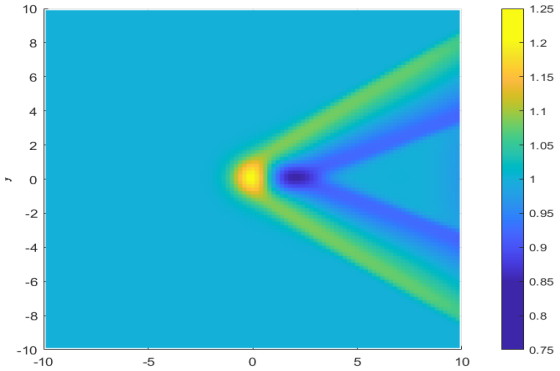

Figure 5.68: Two-dimensional supercritical flow over a bump. Side view of the computed free surface at $t = 10$ computed using the Koren limiter and Roe's method.

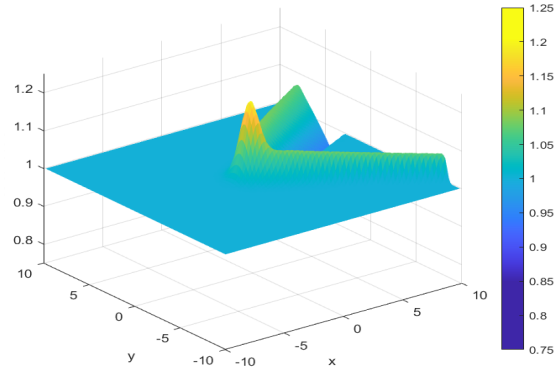

Figure 5.69: Two-dimensional supercritical flow over a bump. View of the computed free surface at $t = 15$ computed using the Koren limiter and Roe's method.

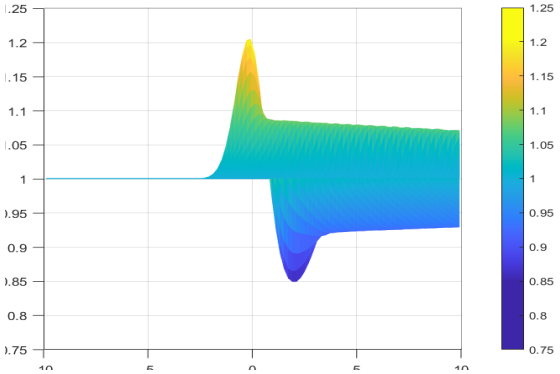

Figure 5.70: Two-dimensional supercritical flow over a bump. Side view of the computed free surface at $t = 15$ computed using the Koren limiter and Roe's method.

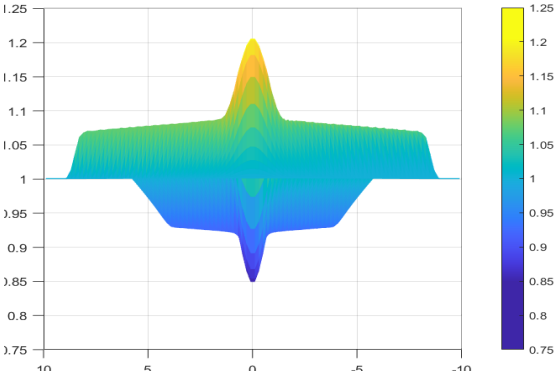

Figure 5.71: Two-dimensional supercritical flow over a bump. Front view of the computed free surface at $t = 15$ computed using the Koren limiter and Roe's method.

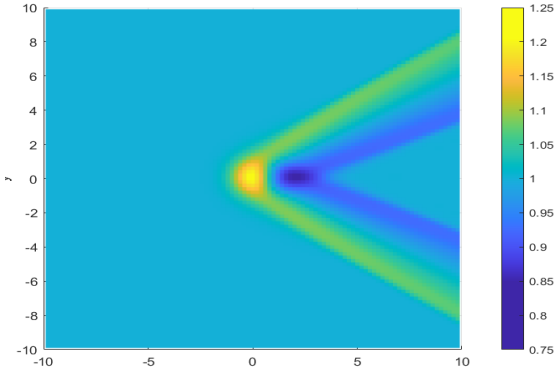

Figure 5.72: Two-dimensional supercritical flow over a bump. Top view of the computed free surface at $t = 15$ computed using the Koren limiter and Roe's method.

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 beter at representing reality than the Superbee limiter. Moreover, the flux vector splitting worked better than the other flux evluation 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 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 noticable) 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., KSINANT, 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 schrittweiten-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] HUNDSDORFER, 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

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

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

where

$$r^l_{i+\frac{1}{2}} = \frac{\vec{q}_{i+1} - \vec{q}_i}{\vec{q}_i - \vec{q}_{i-1}}, \quad r^r_{i+\frac{1}{2}} = \frac{\vec{q}_i - \vec{q}_{i+1}}{\vec{q}_{i+1} - \vec{q}_{i+2}}. \tag{3.21}$$

Furthermore, recall that the Superbee limiter is given by

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

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:

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

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 $\vec{q}_{i-1}$, $\vec{q}_i$ and $\vec{q}_{i+1}$ have the same value and we obviously want $\vec{q}^{\,l}_{i+\frac{1}{2}}$ 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 $\phi_s b = 0$ or $\phi_s b = 2$ 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

```
function f = phi_t(number,a,b)

    aa = a * a;
    ab = a * b;

    % first-order upwind
    if number == 0
        f = 0;

    % kappa = 1/3
    elseif number == 1
      f = b * (1/3) + a * (2/3);

    % minmod limiter
    elseif number  == 2
        if ab <= 0
            f = 0;
        elseif aa < ab
            f = a;
        else
            f = b;
        end

    % superbee limiter
    elseif number == 3
        if ab <= 0
            f = 0;
        elseif aa < (1/2) * ab
            f = 2 * a;
        elseif aa < ab
            f = b;
        elseif aa < 2 * ab
            f = a;
        else
            f = 2 * b;
        end

  % Koren limiter
   elseif number == 4
        if ab <= 0
            f = 0;
        elseif aa < (1/4) * ab
            f = 2 * a;
        elseif aa < (5/2) * ab
            f = (1/3) * b + (2/3) * a;
        else
            f = 2 * b;
        end
    end

end
```

B.2 Test case 1: State interpolation methods

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

```
clear all
format long



%% Select state interpolation and flux evaluation methods

%  state_inter       type
%  -----------       ----
%   0                first-order upwind
%   1                kappa = 1/3
%   2                with minmod limiter
%   3                with superbee limiter
%   4                with Koren limiter



%% Define parameters

Nx = 100; % number of finite volumes (= cells)
CFL = 0.9;  %  C o u r a n t FriedrichsLewy  condition
tmax = 3;  % time at which the simulation ends
g = 1;      % gravitational acceleration



%% Begin initial condition initialization

xmin = -8;                  % x-coordinate at the left boundary of the domain
xmax = 8;                   % x-coordinate at the right boundary of the domain
dx = ( xmax - xmin ) / Nx;  % cell size

x = xmin:dx:xmax;                         % x-coordinates of the cell boundaries
xmid = (1/2)*( x(1:Nx) + x(2:Nx+1) ); % x-coordinates of the cell centres

% set the initial values for h, hu and b at the cell centres
b = zeros(Nx,1);
q = zeros(Nx,2);

for i=1:Nx
    b(i) = 0;                       % b
    if xmid(i) <= 0                 % h
        q(i,1) = 2 - b(i);
    else
        q(i,1) = 1 - b(i);
    end
    q(i,1) = 1 + exp(-xmid(i)^2);   % h
    q(i,2) = 0;                     % hu
end



%% Begin simulation

for k=1:5

    state_inter = k-1;

    b = zeros(Nx,1);
    q = zeros(Nx,2);

    for i=1:Nx
        b(i) = 0;                       % b
        if xmid(i) <= 0                 % h
            q(i,1) = 2 - b(i);
        else
            q(i,1) = 1 - b(i);
        end
        q(i,1) = 1 + exp(-xmid(i)^2);   % h
```

```
        q(i,2) = 0;                         % hu
    end

    % set the boundary values
    q_left_boundary = q(1,:);
    q_right_boundary = q(Nx,:);
    q_left_boundary(2) = 0;
    q_right_boundary(2) = 0;

    ql = zeros(Nx+1,2);
    qr = zeros(Nx+1,2);
    F = zeros(Nx+1,2);

    du1 = zeros(Nx,2);
    du2 = zeros(Nx,2);
    du3 = zeros(Nx,2);

    t = 0;

    while t < tmax

        % calculate the timestep size
        max_wave_speed = 0;
        for i = 1:Nx
            if abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1))) > max_wave_speed
                max_wave_speed = abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1)));
            end
            if abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1))) > max_wave_speed
                max_wave_speed = abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1)));
            end
        end
        %if max_wave_speed > wave_speed
        %    wave_speed = max_wave_speed;
        %end
        %max_wave_speed = 1.621310199408591;
        dt = CFL * (dx / (2 * max_wave_speed) );

        % if the simulation arrives at the final timestep
        if t + dt - tmax > 0
            dt = tmax - t;
            break_this_loop = 1;
        end

        % adjust the time
        t = t + dt;

        %q_left_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
        %q_right_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
        %q_left_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
        %q_right_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);



        qhelp = q; % used to calculate the values for h and hu in the intermediate RK3b steps

        for stage = 1:3    % RK3b has three stages



            %% Begin computation state interpolation

            % note that ql(1,:) corresponds to q^l_{1/2} in the report and ql(Nx+1,:)
    corresponds to q^l_{M+1/2} in the report
            for i = 1:Nx+1
                if i==1
                    ql(i,:) = q_left_boundary;
                elseif i==2
                    ql(i,:) = (1/2)*( qhelp(1,:) + qhelp(2,:) );
                elseif i==Nx+1
                    ql(i,:) = (3/2)*qhelp(Nx,:) - (1/2)*qhelp(Nx-1,:);
                else
                    for j=1:2
```

```
                    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) );
                end
            end
        end

        % 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
        for i=1:Nx+1
            if i==1
                qr(i,:) = (3/2)*qhelp(1,:) - (1/2)*qhelp(2,:);
            elseif i==Nx
                qr(i,:) = (1/2)*( qhelp(Nx-1,:) + qhelp(Nx,:) );
            elseif i==Nx+1
                qr(i,:) = q_right_boundary;
            else
                for j=1:2
                    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) );
                end
            end
        end



        %% Flux method

        % The HLLE Solver
        for i = 1:Nx+1
            h_bar = (1/2)*(ql(i,1)+qr(i,1));
            c_hat = sqrt(g*h_bar);
            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)));
            lambda1 = u_hat - c_hat;
            lambda2 = u_hat + c_hat;

            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))));
            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))));

            if sL >= 0
                F(i,1) = ql(i,2);
                F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
            elseif sR <= 0
                F(i,1) = qr(i,2);
                F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
            else
                F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR - sL );
                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 );
            end
        end



        %% Perform time-integration method

        if stage == 1
            du1(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
            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));
            du1(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
            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));
            for i=2:Nx-1
                du1(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                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));
            end
            for i=3:Nx-2
                for j=1:2
```

```
                        qhelp(i,j)=q(i,j)+du1(i,j);
                    end
                end
            elseif stage == 2
                du2(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du2(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du2(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        qhelp(i,j)=q(i,j)+(1/4)*du1(i,j)+(1/4)*du2(i,j);
                    end
                end
            elseif stage == 3
                du3(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du3(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du3(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        q(i,j)=q(i,j)+(1/6)*du1(i,j)+(1/6)*du2(i,j)+(4/6)*du3(i,j);
                    end
                end
            end
        end
    end

    if k == 2
        hold on
    end
    uplot = q(:,2)./q(:,1);
    plot(xmid,uplot,'-o');
    xlabel('x');
    ylabel('free surface');
    axis([xmin,xmax,-0.5,0.5]);
    %title(['state\_interpolation=',num2str(state_inter),',  CFL=',num2str(CFL),',  Nx=',num2str(Nx),',  Tmax=',num2str(tmax),',  t=',num2str(round(t,4))]);
    grid on

end

%% Define parameters

Nx = 10000; % number of finite volumes (= cells)
state_inter = 4;



%% Begin initial condition initialization
dx = ( xmax - xmin ) / Nx;  % cell size

x = xmin:dx:xmax;                       % x-coordinates of the cell boundaries
xmid = (1/2)*( x(1:Nx) + x(2:Nx+1) ); % x-coordinates of the cell centres

b = zeros(Nx,1);
q = zeros(Nx,2);

for i=1:Nx
    b(i) = 0;                          % b
```

```
    if xmid(i) <= 0                 % h
        q(i,1) = 2 - b(i);
    else
        q(i,1) = 1 - b(i);
    end
    q(i,1) = 1 + exp(-xmid(i)^2);   % h
    q(i,2) = 0;                     % h
end

    % set the boundary values
    q_left_boundary = q(1,:);
    q_right_boundary = q(Nx,:);
    q_left_boundary(2) = 0;
    q_right_boundary(2) = 0;

    ql = zeros(Nx+1,2);
    qr = zeros(Nx+1,2);
    F = zeros(Nx+1,2);

    du1 = zeros(Nx,2);
    du2 = zeros(Nx,2);
    du3 = zeros(Nx,2);

    t = 0;

    while t < tmax

        % calculate the timestep size
        max_wave_speed = 0;
        for i = 1:Nx
            if abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1))) > max_wave_speed
                max_wave_speed = abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1)));
            end
            if abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1))) > max_wave_speed
                max_wave_speed = abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1)));
            end
        end
        %if max_wave_speed > wave_speed
        %    wave_speed = max_wave_speed;
        %end
        %max_wave_speed = 1.621310199408591;
        dt = CFL * (dx / (2 * max_wave_speed) );

        % if the simulation arrives at the final timestep
        if t + dt - tmax > 0
            dt = tmax - t;
            break_this_loop = 1;
        end

        % adjust the time
        t = t + dt;

        %q_left_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
        %q_right_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
        %q_left_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
        %q_right_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);



        qhelp = q; % used to calculate the values for h and hu in the intermediate RK3b steps

        for stage = 1:3    % RK3b has three stages



            %% Begin computation state interpolation

            % note that ql(1,:) corresponds to q^l_{1/2} in the report and ql(Nx+1,:)
    corresponds to q^l_{M+1/2} in the report
            for i = 1:Nx+1
                if i==1
                    ql(i,:) = q_left_boundary;
                elseif i==2
```

```
                ql(i,:) = (1/2)*( qhelp(1,:) + qhelp(2,:) );
            elseif i==Nx+1
                ql(i,:) = (3/2)*qhelp(Nx,:) - (1/2)*qhelp(Nx-1,:);
            else
                for j=1:2
                    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) );
                end
            end
        end

        % 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
        for i=1:Nx+1
            if i==1
                qr(i,:) = (3/2)*qhelp(1,:) - (1/2)*qhelp(2,:);
            elseif i==Nx
                qr(i,:) = (1/2)*( qhelp(Nx-1,:) + qhelp(Nx,:) );
            elseif i==Nx+1
                qr(i,:) = q_right_boundary;
            else
                for j=1:2
                    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) );
                end
            end
        end



        %% Flux method

        % The HLLE Solver
        for i = 1:Nx+1
            h_bar = (1/2)*(ql(i,1)+qr(i,1));
            c_hat = sqrt(g*h_bar);
            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)));
            lambda1 = u_hat - c_hat;
            lambda2 = u_hat + c_hat;

            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))));
            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))));

            if sL >= 0
                F(i,1) = ql(i,2);
                F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
            elseif sR <= 0
                F(i,1) = qr(i,2);
                F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
            else
                F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR - sL );
                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 );
            end
        end



        %% Perform time-integration method

        if stage == 1
            du1(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
            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));
            du1(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
            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));
            for i=2:Nx-1
                du1(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
```

```
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        qhelp(i,j)=q(i,j)+du1(i,j);
                    end
                end
            elseif stage == 2
                du2(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du2(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du2(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        qhelp(i,j)=q(i,j)+(1/4)*du1(i,j)+(1/4)*du2(i,j);
                    end
                end
            elseif stage == 3
                du3(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du3(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du3(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        q(i,j)=q(i,j)+(1/6)*du1(i,j)+(1/6)*du2(i,j)+(4/6)*du3(i,j);
                    end
                end
            end
        end
    end



uplot = q(:,2)./q(:,1);
plot(xmid,uplot,'-k');
xlabel('x');
ylabel('velocity u');
axis([xmin,xmax,-0.5,0.5]);
%title(['state\_interpolation=',num2str(state_inter),',  CFL=',num2str(CFL),',  Nx=',num2str(
    Nx),',  Tmax=',num2str(tmax),',  t=',num2str(round(t,4))]);
grid on
hold off
legend({'First-order upwind','Kappa = 1/3','Minmod limiter','Superbee limiter','Koren limiter
    ',''},'Location','northwest','Orientation','vertical')



  %% Begin plotting
    %if mod(count,round(Nx^(1/2))) == 0
    %    for i = 1:Nx
    %        freude(i) = abs(q(i,2)/q(i,1))/sqrt(g*q(i,1));
    %    end
    %    surface = q(:,1) + b;
    %    for i = 1:Nx
    %        stationary(i) = q_left_boundary(2)/q_left_boundary(1);
    %    end

    %    hold off
```

```
    %     plot(xmid,surface,'-o',xmid,freude,'-o',xmid,b,'-o',xmid,stationary,'-');
    %     xlabel('x');
    %     ylabel('free surface');
    %     axis([xmin,xmax,0,2]);
    %     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))]);
    %     hold on
    %     grid on
    %     drawnow
    %end



    %% Calculate total energy in the system
    %energy(count) = 0;
    %for i = 1:Nx
    % 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);
    %end
    %energy(count) = energy(count) / Nx;



    %% Error calculation
    %error(count) = 0;
    %for i = 1:Nx
    %     error(count) = error(count) + abs(q(i,1) + b(i) - q_left_boundary(1));
    %end
    %error(count) = error(count)/Nx;




%tiledlayout(2,1)

%% Plot error calculation
%nexttile
%plot(time,error,'-');
%xlabel('time');
%ylabel('error vs stationary solution');



%% Plot energy calculation
%nexttile
%plot(time,energy,'-');
%xlabel('time');
%ylabel('energy');



%wave_speed
%energy = 0;
%for i = 1:Nx
% energy = energy + ((1/2)*((q(i,2)/q(i,1))^2)) + (g*(q(i,1)+2*b(i))/2);
%end
%energy = energy / Nx;
%energy
```

B.3 Test case 1: Flux evaluation methods

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

```
clear all
format long



%% Select state interpolation and flux evaluation methods

%  state_inter       type
%  -----------       ----
%   0                first-order upwind
%   1                kappa = 1/3
%   2                with minmod limiter
%   3                with superbee limiter
%   4                with Koren limiter

%  flux_eval       type
%  ---------       ----
%   0              flux vector splitting
%   1              flux difference splitting: the midpoint rule
%   2              flux difference splitting: the trapezoidal rule
%   3              flux difference splitting: Roe's method (LeVeque 15.3.3 in Finite Volume Methods for Hyperbolic Problems)
%   4              HLL Solver (Toro 10.4.1 with (10.18) in E.F. Toro - Shock-capturing methods for free-surface shallow flows)
%   5              HLLE Solver (LeVeque 15.3.7 in Finite Volume Methods for Hyperbolic Problems)



%% Define parameters

Nx = 100; % number of finite volumes (= cells)
CFL = 0.9;  %  C o u r a n t FriedrichsLewy  condition
tmax = 3;  % time at which the simulation ends
g = 1;      % gravitational acceleration

state_inter = 4;



%% Begin initial condition initialization

xmin = -10;                 % x-coordinate at the left boundary of the domain
xmax = 10;                  % x-coordinate at the right boundary of the domain
dx = ( xmax - xmin ) / Nx;  % cell size

x = xmin:dx:xmax;                       % x-coordinates of the cell boundaries
xmid = (1/2)*( x(1:Nx) + x(2:Nx+1) ); % x-coordinates of the cell centres



%% Begin simulation

for k=1:6

    b = zeros(Nx,1);
    q = zeros(Nx,2);

    for i=1:Nx
        b(i) = 0;                       % b
        q(i,1) = 1 + exp(-xmid(i)^2);   % h
        q(i,2) = 0;                     % hu
    end

    % set the boundary values
    q_left_boundary = q(1,:);
    q_right_boundary = q(Nx,:);
    q_left_boundary(2) = 0;
    q_right_boundary(2) = 0;
```

```
67
68      ql = 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 ql(1,:) corresponds to q^l_{1/2} in the report and ql(Nx+1,:)
    corresponds to q^l_{M+1/2} in the report
128             for i = 1:Nx+1
129                 if i==1
130                     ql(i,:) = q_left_boundary;
131                 elseif i==2
132                     ql(i,:) = (1/2)*( qhelp(1,:) + qhelp(2,:) );
133                 elseif i==Nx+1
134                     ql(i,:) = (3/2)*qhelp(Nx,:) - (1/2)*qhelp(Nx-1,:);
135                 else
136                     for j=1:2
137                         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) );
```

```
                end
            end
        end

        % 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
        for i=1:Nx+1
            if i==1
                qr(i,:) = (3/2)*qhelp(1,:) - (1/2)*qhelp(2,:);
            elseif i==Nx
                qr(i,:) = (1/2)*( qhelp(Nx-1,:) + qhelp(Nx,:) );
            elseif i==Nx+1
                qr(i,:) = q_right_boundary;
            else
                for j=1:2
                    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) );
                end
            end
        end



        %% Flux method

        % Flux vector splitting
        if k == 1
            for i = 1:Nx+1
                lambda1L = (ql(i,2)/ql(i,1)) - sqrt(g*ql(i,1)/2);
                lambda2L = (ql(i,2)/ql(i,1)) + sqrt(g*ql(i,1)/2);
                diagL = diag([max(lambda1L,0) ; max(lambda2L,0)]);
                RL = [ 1, 1; lambda1L, lambda2L ];
                R_invL = (1/sqrt(2*g*ql(i,1))) * [ lambda2L, -1; -lambda1L, 1];

                lambda1R = (qr(i,2)/qr(i,1)) - sqrt(g*qr(i,1)/2);
                lambda2R = (qr(i,2)/qr(i,1)) + sqrt(g*qr(i,1)/2);
                diagR = diag([min(lambda1R,0) ; min(lambda2R,0)]);
                RR = [ 1, 1; lambda1R, lambda2R ];
                R_invR = (1/sqrt(2*g*qr(i,1))) * [ lambda2R, -1; -lambda1R, 1];

                F(i,:) = RL*diagL*R_invL*ql(i,:).' + RR*diagR*R_invR*qr(i,:).';
            end

        % Flux difference splitting: The midpoint rule
        elseif k == 2
            for i = 1:Nx+1
                qm = (1/2)*( ql(i,:) + qr(i,:) );
                lambda1 = (qm(2)/qm(1)) - sqrt(g*qm(1));
                lambda2 = (qm(2)/qm(1)) + sqrt(g*qm(1));
                diag_matrix = diag([abs(lambda1) ; abs(lambda2)]);
                R = [ 1, 1; lambda1, lambda2 ];
                R_inv = (1/(2*sqrt(g*qm(1)))) * [ lambda2, -1; -lambda1, 1];

                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,:)).') ).';
            end

        % Flux difference splitting: The trapezoidal rule
        elseif k == 3
            for i = 1:Nx+1
                lambda1L = (ql(i,2)/ql(i,1)) - sqrt(g*ql(i,1));
                lambda2L = (ql(i,2)/ql(i,1)) + sqrt(g*ql(i,1));
                diagL = diag([abs(lambda1L) ; abs(lambda2L)]);
                RL = [ 1, 1; lambda1L, lambda2L ];
                R_invL = (1/sqrt(2*g*ql(i,1))) * [ lambda2L, -1; -lambda1L, 1];

                lambda1R = (qr(i,2)/qr(i,1)) - sqrt(g*qr(i,1));
                lambda2R = (qr(i,2)/qr(i,1)) + sqrt(g*qr(i,1));
                diagR = diag([abs(lambda1R) ; abs(lambda2R)]);
                RR = [ 1, 1; lambda1R, lambda2R ];
                R_invR = (1/sqrt(2*g*qr(i,1))) * [ lambda2R, -1; -lambda1R, 1];

```

```
                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,:)).') ).';
            end

        % Flux difference splitting: Roe's method
        elseif k == 4
            for i = 1:Nx+1
                h_bar = (1/2)*(ql(i,1)+ql(i,1));
                c_hat = sqrt(g*h_bar);
                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)));
                lambda1 = u_hat - c_hat;
                lambda2 = u_hat + c_hat;

                diag_matrix = diag([abs(lambda1) ; abs(lambda2)]);
                R = [ 1, 1; lambda1, lambda2 ];
                R_inv = (1/(2*c_hat)) * [ lambda2, -1; -lambda1, 1];

                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,:)).') ).';
            end

        % The HLL Solver
        elseif k == 5
            for i = 1:Nx+1
                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);

                if h_hat > ql(i,1)
                    pl = sqrt( (1/2)*( (h_hat*(h_hat+ql(i,1)))/(ql(i,1)^2) ) );
                else
                    pl = 1;
                end

                if h_hat > qr(i,1)
                    pr = sqrt( (1/2)*( (h_hat*(h_hat+qr(i,1)))/(qr(i,1)^2) ) );
                else
                    pr = 1;
                end

                sL = (ql(i,2)/ql(i,1)) - (sqrt(g*ql(i,1))*pl);
                sR = (qr(i,2)/qr(i,1)) + (sqrt(g*qr(i,1))*pr);

                if sL >= 0
                    F(i,1) = ql(i,2);
                    F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
                elseif sR <= 0
                    F(i,1) = qr(i,2);
                    F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
                else
                    F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR
- sL );
                    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
);
                end
            end

        % The HLLE Solver
        elseif k == 6
            for i = 1:Nx+1
                h_bar = (1/2)*(ql(i,1)+qr(i,1));
                c_hat = sqrt(g*h_bar);
                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)));
                lambda1 = u_hat - c_hat;
                lambda2 = u_hat + c_hat;

                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))));
```

```
                    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))));

                    if sL >= 0
                        F(i,1) = ql(i,2);
                        F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
                    elseif sR <= 0
                        F(i,1) = qr(i,2);
                        F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
                    else
                        F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR - sL );
                        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 );
                    end
                end
            end



            %% Perform time-integration method

            if stage == 1
                du1(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du1(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du1(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        qhelp(i,j)=q(i,j)+du1(i,j);
                    end
                end
            elseif stage == 2
                du2(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du2(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du2(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        qhelp(i,j)=q(i,j)+(1/4)*du1(i,j)+(1/4)*du2(i,j);
                    end
                end
            elseif stage == 3
                du3(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du3(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du3(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        q(i,j)=q(i,j)+(1/6)*du1(i,j)+(1/6)*du2(i,j)+(4/6)*du3(i,j);
                    end
```

```
                end
            end
        end
    end

    if k == 2
        hold on
    end
    plot(xmid,q(:,2)./q(:,1),'-o');
    xlabel('x');
    ylabel('velocity u');
    axis([xmin,xmax,-0.5,0.5]);
    %title(['state\_interpolation=',num2str(state_inter),',  CFL=',num2str(CFL),',  Nx=',
    num2str(Nx),',  Tmax=',num2str(tmax),',  t=',num2str(round(t,4))]);
    grid on

end

%% Define parameters

Nx = 10000; % number of finite volumes (= cells)
wave_speed = 0;


%% Begin initial condition initialization
dx = ( xmax - xmin ) / Nx;  % cell size

x = xmin:dx:xmax;                       % x-coordinates of the cell boundaries
xmid = (1/2)*( x(1:Nx) + x(2:Nx+1) ); % x-coordinates of the cell centres

b = zeros(Nx,1);
q = zeros(Nx,2);

for i=1:Nx
    b(i) = 0;                          % b
    if xmid(i) <= 0                    % h
        q(i,1) = 2 - b(i);
    else
        q(i,1) = 1 - b(i);
    end
    q(i,1) = 1 + exp(-xmid(i)^2);   % h
    q(i,2) = 0;                        % h
end

    % set the boundary values
    q_left_boundary = q(1,:);
    q_right_boundary = q(Nx,:);
    q_left_boundary(2) = 0;
    q_right_boundary(2) = 0;

    ql = zeros(Nx+1,2);
    qr = zeros(Nx+1,2);
    F = zeros(Nx+1,2);

    du1 = zeros(Nx,2);
    du2 = zeros(Nx,2);
    du3 = zeros(Nx,2);

    t = 0;

    while t < tmax

        % calculate the timestep size
        max_wave_speed = 0;
        for i = 1:Nx
            if abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1))) > max_wave_speed
                max_wave_speed = abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1)));
            end
            if abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1))) > max_wave_speed
                max_wave_speed = abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1)));
            end
        end
        if max_wave_speed > wave_speed
```

```
            wave_speed = max_wave_speed;
        end
        dt = CFL * (dx / (2 * max_wave_speed) );

        % if the simulation arrives at the final timestep
        if t + dt - tmax > 0
            dt = tmax - t;
            break_this_loop = 1;
        end

        % adjust the time
        t = t + dt;

        %q_left_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
        %q_right_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
        %q_left_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
        %q_right_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);



        qhelp = q; % used to calculate the values for h and hu in the intermediate RK3b steps

        for stage = 1:3    % RK3b has three stages



            %% Begin computation state interpolation

            % note that ql(1,:) corresponds to q^l_{1/2} in the report and ql(Nx+1,:)
corresponds to q^l_{M+1/2} in the report
            for i = 1:Nx+1
                if i==1
                    ql(i,:) = q_left_boundary;
                elseif i==2
                    ql(i,:) = (1/2)*( qhelp(1,:) + qhelp(2,:) );
                elseif i==Nx+1
                    ql(i,:) = (3/2)*qhelp(Nx,:) - (1/2)*qhelp(Nx-1,:);
                else
                    for j=1:2
                        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) );
                    end
                end
            end

            % 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
            for i=1:Nx+1
                if i==1
                    qr(i,:) = (3/2)*qhelp(1,:) - (1/2)*qhelp(2,:);
                elseif i==Nx
                    qr(i,:) = (1/2)*( qhelp(Nx-1,:) + qhelp(Nx,:) );
                elseif i==Nx+1
                    qr(i,:) = q_right_boundary;
                else
                    for j=1:2
                        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) );
                    end
                end
            end



            %% Flux method
                for i = 1:Nx+1
                    h_bar = (1/2)*(ql(i,1)+qr(i,1));
                    c_hat = sqrt(g*h_bar);
                    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)));
                    lambda1 = u_hat - c_hat;
                    lambda2 = u_hat + c_hat;

```

```
                    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))));
                    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))));

                    if sL >= 0
                        F(i,1) = ql(i,2);
                        F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
                    elseif sR <= 0
                        F(i,1) = qr(i,2);
                        F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
                    else
                        F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR - sL );
                        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 );
                    end
                end



            %% Perform time-integration method

            if stage == 1
                du1(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du1(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du1(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        qhelp(i,j)=q(i,j)+du1(i,j);
                    end
                end
            elseif stage == 2
                du2(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du2(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du2(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        qhelp(i,j)=q(i,j)+(1/4)*du1(i,j)+(1/4)*du2(i,j);
                    end
                end
            elseif stage == 3
                du3(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du3(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du3(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        q(i,j)=q(i,j)+(1/6)*du1(i,j)+(1/6)*du2(i,j)+(4/6)*du3(i,j);
```

```
                    end
                end
            end
        end
    end




plot(xmid,q(:,2)./q(:,1),'-k');
xlabel('x');
ylabel('velocity u');
axis([xmin,xmax,-0.5,0.5]);
%title(['state\_interpolation=',num2str(state_inter),',  CFL=',num2str(CFL),',  Nx=',num2str(
    Nx),',  Tmax=',num2str(tmax),',  t=',num2str(round(t,4))]);
grid on
hold off
legend({'Flux vector splitting','Midpoint rule','Trapezoidal rule','Roe''s method','HLL solver
    ','HLLE solver',''},'Location','northwest','Orientation','vertical')



  %% Begin plotting
    %if mod(count,round(Nx^(1/2))) == 0
    %    for i = 1:Nx
    %        freude(i) = abs(q(i,2)/q(i,1))/sqrt(g*q(i,1));
    %    end
    %    surface = q(:,1) + b;
    %    for i = 1:Nx
    %        stationary(i) = q_left_boundary(2)/q_left_boundary(1);
    %    end

    %    hold off
    %    plot(xmid,surface,'-o',xmid,freude,'-o',xmid,b,'-o',xmid,stationary,'-');
    %    xlabel('x');
    %    ylabel('free surface');
    %    axis([xmin,xmax,0,2]);
    %    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))]);
    %    hold on
    %    grid on
    %    drawnow
    %end



    %% Calculate total energy in the system
    %energy(count) = 0;
    %for i = 1:Nx
    % 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);
    %end
    %energy(count) = energy(count) / Nx;



    %% Error calculation
    %error(count) = 0;
    %for i = 1:Nx
    %    error(count) = error(count) + abs(q(i,1) + b(i) - q_left_boundary(1));
    %end
    %error(count) = error(count)/Nx;




%tiledlayout(2,1)

%% Plot error calculation
%nexttile
%plot(time,error,'-');
```

```
%xlabel('time');
%ylabel('error vs stationary solution');



%% Plot energy calculation
%nexttile
%plot(time,energy,'-');
%xlabel('time');
%ylabel('energy');



%wave_speed
energy = 0;
for i = 1:Nx
  energy = energy + ((1/2)*((q(i,2)/q(i,1))^2)) + (g*(q(i,1)+2*b(i))/2);
end
energy = energy / Nx;
energy
wave_speed
```

B.4 Test case 2: State interpolation methods

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

```
clear all
format long



%% Select state interpolation and flux evaluation methods

%  state_inter       type
%  -----------       ----
%   0                first-order upwind
%   1                kappa = 1/3
%   2                with minmod limiter
%   3                with superbee limiter
%   4                with Koren limiter

%  flux_eval       type
%  ---------       ----
%   0              flux vector splitting
%   1              flux difference splitting: the midpoint rule
%   2              flux difference splitting: the trapezoidal rule
%   3              flux difference splitting: Roe's method (LeVeque 15.3.3 in Finite Volume
    Methods for Hyperbolic Problems)
%   4             HLL Solver (Toro 10.4.1 with (10.18) in E.F. Toro - Shock-capturing methods
    for free-surface shallow flows)
%   5             HLLE Solver (LeVeque 15.3.7 in Finite Volume Methods for Hyperbolic Problems
    )



%% Define parameters

Nx = 100; % number of finite volumes (= cells)
CFL = 0.45;  %  C o u r a n t FriedrichsLewy  condition
tmax = 1;  % time at which the simulation ends
g = 1;      % gravitational acceleration



%% Begin initial condition initialization

xmin = -8;                % x-coordinate at the left boundary of the domain
xmax = 8;                 % x-coordinate at the right boundary of the domain
dx = ( xmax - xmin ) / Nx;  % cell size

x = xmin:dx:xmax;                       % x-coordinates of the cell boundaries
xmid = (1/2)*( x(1:Nx) + x(2:Nx+1) ); % x-coordinates of the cell centres

h = zeros(Nx,5);
u = zeros(Nx,5);



%% Begin simulation

for k=1:5

    state_inter = k-1;

    b = zeros(Nx,1);
    q = zeros(Nx,2);

    for i=1:Nx
        b(i) = 0;                       % b
        if xmid(i) <= 0                 % h
            q(i,1) = 2 - b(i);
        else
            q(i,1) = 1 - b(i);
        end
        q(i,2) = 0;                     % hu
```

```
    end

    % set the boundary values
    q_left_boundary = q(1,:);
    q_right_boundary = q(Nx,:);
    q_left_boundary(2) = 0;
    q_right_boundary(2) = 0;

    ql = zeros(Nx+1,2);
    qr = zeros(Nx+1,2);
    F = zeros(Nx+1,2);

    du1 = zeros(Nx,2);
    du2 = zeros(Nx,2);
    du3 = zeros(Nx,2);

    t = 0;

    while t < tmax

        % calculate the timestep size
        max_wave_speed = 0;
        for i = 1:Nx
            if abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1))) > max_wave_speed
                max_wave_speed = abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1)));
            end
            if abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1))) > max_wave_speed
                max_wave_speed = abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1)));
            end
        end
        %if max_wave_speed > wave_speed
        %    wave_speed = max_wave_speed;
        %end
        %max_wave_speed = 1.621310199408591;
        dt = CFL * (dx / (2 * max_wave_speed) );

        % if the simulation arrives at the final timestep
        if t + dt - tmax > 0
            dt = tmax - t;
            break_this_loop = 1;
        end

        % adjust the time
        t = t + dt;

        %q_left_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
        %q_right_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
        %q_left_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
        %q_right_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);



        qhelp = q; % used to calculate the values for h and hu in the intermediate RK3b steps

        for stage = 1:3    % RK3b has three stages



            %% Begin computation state interpolation

            % note that ql(1,:) corresponds to q^l_{1/2} in the report and ql(Nx+1,:)
corresponds to q^l_{M+1/2} in the report
            for i = 1:Nx+1
                if i==1
                    ql(i,:) = q_left_boundary;
                elseif i==2
                    ql(i,:) = (1/2)*( qhelp(1,:) + qhelp(2,:) );
                elseif i==Nx+1
                    ql(i,:) = (3/2)*qhelp(Nx,:) - (1/2)*qhelp(Nx-1,:);
                else
                    for j=1:2
                        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) );
```

```
                end
            end
        end

        % 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
        for i=1:Nx+1
            if i==1
                qr(i,:) = (3/2)*qhelp(1,:) - (1/2)*qhelp(2,:);
            elseif i==Nx
                qr(i,:) = (1/2)*( qhelp(Nx-1,:) + qhelp(Nx,:) );
            elseif i==Nx+1
                qr(i,:) = q_right_boundary;
            else
                for j=1:2
                    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) );
                end
            end
        end



        %% Flux method

        % The HLLE Solver
        for i = 1:Nx+1
            h_bar = (1/2)*(ql(i,1)+qr(i,1));
            c_hat = sqrt(g*h_bar);
            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)));
            lambda1 = u_hat - c_hat;
            lambda2 = u_hat + c_hat;

            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))));
            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))));

            if sL >= 0
                F(i,1) = ql(i,2);
                F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
            elseif sR <= 0
                F(i,1) = qr(i,2);
                F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
            else
                F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR - sL );
                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 );
            end
        end



        %% Perform time-integration method

        if stage == 1
            du1(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
            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));
            du1(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
            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));
            for i=2:Nx-1
                du1(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                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));
            end
            for i=3:Nx-2
                for j=1:2
                    qhelp(i,j)=q(i,j)+du1(i,j);
                end
```

```
                end
            elseif stage == 2
                du2(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du2(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du2(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        qhelp(i,j)=q(i,j)+(1/4)*du1(i,j)+(1/4)*du2(i,j);
                    end
                end
            elseif stage == 3
                du3(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du3(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du3(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        q(i,j)=q(i,j)+(1/6)*du1(i,j)+(1/6)*du2(i,j)+(4/6)*du3(i,j);
                    end
                end
            end
        end
    end

    if k == 2
        hold on
    end

    h(:,k) = q(:,1);
    u(:,k) = q(:,2)./q(:,1);

    uplot = q(:,1);
    plot(xmid,uplot,'-o');
    xlabel('x');
    ylabel('free surface');
    axis([xmin,xmax,-0.5,0.5]);
    %title(['state\_interpolation=',num2str(state_inter),',  CFL=',num2str(CFL),',  Nx=',num2str(Nx),',  Tmax=',num2str(tmax),',  t=',num2str(round(t,4))]);
    grid on

end



%% Define parameters

Nx = 100; % number of finite volumes (= cells)
dx = ( xmax - xmin ) / Nx;  % cell size

x = xmin:dx:xmax;                       % x-coordinates of the cell boundaries
xmid = (1/2)*( x(1:Nx) + x(2:Nx+1) ); % x-coordinates of the cell centres

h_e = zeros(Nx,1);
u_e = zeros(Nx,1);

c_m = 1.205753246885354;
x_a = - tmax * sqrt(g*2);
x_b = t * ( 2 * sqrt(g*2) - 3 * c_m );
```

```
x_c = t * ( 2*(c_m^2)*( sqrt(g*2) - c_m) ) / ( (c_m^2) - (g*1) );

for i=1:Nx
    if xmid(i) <= x_a
        h_e(i) = 2;
        u_e(i) = 0;
    elseif xmid(i) <= x_b
        h_e(i) = ( 4 / (9 * g) ) * (( sqrt(g*2) - (xmid(i)/(2*tmax)) )^2);
        u_e(i) = (2/3) * ((xmid(i)/tmax) + sqrt(g*2));
    elseif xmid(i) <= x_c
        h_e(i) = (c_m^2)/g;
        u_e(i) = 2 * ( sqrt(g*2) - c_m );
    else
        h_e(i) = 1;
        u_e(i) = 0;
    end
end



plot(xmid,h_e,'-k');
xlabel('x');
%ylabel('velocity u');
ylabel('free surface s');
%axis([xmin,xmax,-0.05,0.5]);
axis([xmin,xmax,0.9,2.1]);
%title(['state\_interpolation=',num2str(state_inter),',  CFL=',num2str(CFL),',  Nx=',num2str(
    Nx),',  Tmax=',num2str(tmax),',  t=',num2str(round(t,4))]);
grid on
hold off
legend({'First-order upwind','Kappa = 1/3','Minmod limiter','Superbee limiter','Koren limiter
    ',''},'Location','northeast','Orientation','vertical')



 %% Begin plotting
   %if mod(count,round(Nx^(1/2))) == 0
   %    for i = 1:Nx
   %        freude(i) = abs(q(i,2)/q(i,1))/sqrt(g*q(i,1));
   %    end
   %    surface = q(:,1) + b;
   %    for i = 1:Nx
   %        stationary(i) = q_left_boundary(2)/q_left_boundary(1);
   %    end

   %    hold off
   %    plot(xmid,surface,'-o',xmid,freude,'-o',xmid,b,'-o',xmid,stationary,'-');
   %    xlabel('x');
   %    ylabel('free surface');
   %    axis([xmin,xmax,0,2]);
   %    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))]);
   %    hold on
   %    grid on
   %    drawnow
   %end



   %% Calculate total energy in the system
   %energy(count) = 0;
   %for i = 1:Nx
   % 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);
   %end
   %energy(count) = energy(count) / Nx;



   %% Error calculation
   %error(count) = 0;
   %for i = 1:Nx
```

```
    %    error(count) = error(count) + abs(q(i,1) + b(i) - q_left_boundary(1));
    %end
    %error(count) = error(count)/Nx;




%tiledlayout(2,1)

%% Plot error calculation
%nexttile
%plot(time,error,'-');
%xlabel('time');
%ylabel('error vs stationary solution');



%% Plot energy calculation
%nexttile
%plot(time,energy,'-');
%xlabel('time');
%ylabel('energy');



%wave_speed
%energy = 0;
%for i = 1:Nx
% energy = energy + ((1/2)*((q(i,2)/q(i,1))^2)) + (g*(q(i,1)+2*b(i))/2);
%end
%energy = energy / Nx;
%energy



for k=1:5
    disp(k);
    L1_h = 0;
    L2_h = 0;
    L3_h = 0;
    for i=1:Nx
        L1_h = L1_h + abs(h(i,k)-h_e(i));
        L2_h = L2_h + ((h(i,k)-h_e(i))^2);
        if abs(h(i,k)-h_e(i)) > L3_h
            L3_h = abs(h(i,k)-h_e(i));
        end
    end
    disp(L1_h);
    disp(L2_h);
    disp(L3_h);

    L1_u = 0;
    L2_u = 0;
    L3_u = 0;
    for i=1:Nx
        L1_u = L1_u + abs(u(i,k)-u_e(i));
        L2_u = L2_u + ((u(i,k)-u_e(i))^2);
        if abs(u(i,k)-u_e(i)) > L3_u
            L3_u = abs(u(i,k)-u_e(i));
        end
    end
    disp(L1_u);
    disp(L2_u);
    disp(L3_u);
end
```

B.5 Test case 2: Flux evaluation methods

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

```
clear all
format long



%% Select state interpolation and flux evaluation methods

%  state_inter       type
%  -----------       ----
%   0                first-order upwind
%   1                kappa = 1/3
%   2                with minmod limiter
%   3                with superbee limiter
%   4                with Koren limiter

%  flux_eval       type
%  ---------       ----
%   0              flux vector splitting
%   1              flux difference splitting: the midpoint rule
%   2              flux difference splitting: the trapezoidal rule
%   3              flux difference splitting: Roe's method (LeVeque 15.3.3 in Finite Volume
    Methods for Hyperbolic Problems)
%   4             HLL Solver (Toro 10.4.1 with (10.18) in E.F. Toro - Shock-capturing methods
    for free-surface shallow flows)
%   5             HLLE Solver (LeVeque 15.3.7 in Finite Volume Methods for Hyperbolic Problems
    )



%% Define parameters

Nx = 100; % number of finite volumes (= cells)
CFL = 0.9;  %  C o u r a n t FriedrichsLewy  condition
tmax = 3;  % time at which the simulation ends
g = 1;      % gravitational acceleration

state_inter = 3;



%% Begin initial condition initialization

xmin = -8;                % x-coordinate at the left boundary of the domain
xmax = 8;                 % x-coordinate at the right boundary of the domain
dx = ( xmax - xmin ) / Nx;  % cell size

x = xmin:dx:xmax;                       % x-coordinates of the cell boundaries
xmid = (1/2)*( x(1:Nx) + x(2:Nx+1) ); % x-coordinates of the cell centres

h = zeros(Nx,6);
u = zeros(Nx,6);



%% Begin simulation

for k=1:6

    b = zeros(Nx,1);
    q = zeros(Nx,2);

    for i=1:Nx
        b(i) = 0;                       % b
        if xmid(i) <= 0                 % h
            q(i,1) = 2 - b(i);
        else
            q(i,1) = 1 - b(i);
        end
        q(i,2) = 0;                     % hu
```

```
    end

    % set the boundary values
    q_left_boundary = q(1,:);
    q_right_boundary = q(Nx,:);
    q_left_boundary(2) = 0;
    q_right_boundary(2) = 0;

    ql = zeros(Nx+1,2);
    qr = zeros(Nx+1,2);
    F = zeros(Nx+1,2);

    du1 = zeros(Nx,2);
    du2 = zeros(Nx,2);
    du3 = zeros(Nx,2);

    t = 0;

    while t < tmax

        % calculate the timestep size
        max_wave_speed = 0;
        for i = 1:Nx
            if abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1))) > max_wave_speed
                max_wave_speed = abs((q(i,2)/q(i,1)) + sqrt(g*q(i,1)));
            end
            if abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1))) > max_wave_speed
                max_wave_speed = abs((q(i,2)/q(i,1)) - sqrt(g*q(i,1)));
            end
        end
        %if max_wave_speed > wave_speed
        %    wave_speed = max_wave_speed;
        %end
        %max_wave_speed = 1.621310199408591;
        dt = CFL * (dx / (2 * max_wave_speed) );

        % if the simulation arrives at the final timestep
        if t + dt - tmax > 0
            dt = tmax - t;
            break_this_loop = 1;
        end

        % adjust the time
        t = t + dt;

        %q_left_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
        %q_right_boundary(2) = (1/4)*(tanh((1/5)*(t-10))+1);
        %q_left_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);
        %q_right_boundary(2) = 0.1*(tanh((1/20)*(t-80))+10);



        qhelp = q; % used to calculate the values for h and hu in the intermediate RK3b steps

        for stage = 1:3    % RK3b has three stages



            %% Begin computation state interpolation

            % note that ql(1,:) corresponds to q^l_{1/2} in the report and ql(Nx+1,:)
corresponds to q^l_{M+1/2} in the report
            for i = 1:Nx+1
                if i==1
                    ql(i,:) = q_left_boundary;
                elseif i==2
                    ql(i,:) = (1/2)*( qhelp(1,:) + qhelp(2,:) );
                elseif i==Nx+1
                    ql(i,:) = (3/2)*qhelp(Nx,:) - (1/2)*qhelp(Nx-1,:);
                else
                    for j=1:2
                        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) );
```

```
                end
            end
        end

        % 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
        for i=1:Nx+1
            if i==1
                qr(i,:) = (3/2)*qhelp(1,:) - (1/2)*qhelp(2,:);
            elseif i==Nx
                qr(i,:) = (1/2)*( qhelp(Nx-1,:) + qhelp(Nx,:) );
            elseif i==Nx+1
                qr(i,:) = q_right_boundary;
            else
                for j=1:2
                    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) );
                end
            end
        end



        %% Flux method

        % Flux vector splitting
        if k == 1
            for i = 1:Nx+1
                lambda1L = (ql(i,2)/ql(i,1)) - sqrt(g*ql(i,1)/2);
                lambda2L = (ql(i,2)/ql(i,1)) + sqrt(g*ql(i,1)/2);
                diagL = diag([max(lambda1L,0) ; max(lambda2L,0)]);
                RL = [ 1, 1; lambda1L, lambda2L ];
                R_invL = (1/sqrt(2*g*ql(i,1))) * [ lambda2L, -1; -lambda1L, 1];

                lambda1R = (qr(i,2)/qr(i,1)) - sqrt(g*qr(i,1)/2);
                lambda2R = (qr(i,2)/qr(i,1)) + sqrt(g*qr(i,1)/2);
                diagR = diag([min(lambda1R,0) ; min(lambda2R,0)]);
                RR = [ 1, 1; lambda1R, lambda2R ];
                R_invR = (1/sqrt(2*g*qr(i,1))) * [ lambda2R, -1; -lambda1R, 1];

                F(i,:) = RL*diagL*R_invL*ql(i,:).' + RR*diagR*R_invR*qr(i,:).';
            end

        % Flux difference splitting: The midpoint rule
        elseif k == 2
            for i = 1:Nx+1
                qm = (1/2)*( ql(i,:) + qr(i,:) );
                lambda1 = (qm(2)/qm(1)) - sqrt(g*qm(1));
                lambda2 = (qm(2)/qm(1)) + sqrt(g*qm(1));
                diag_matrix = diag([abs(lambda1) ; abs(lambda2)]);
                R = [ 1, 1; lambda1, lambda2 ];
                R_inv = (1/(2*sqrt(g*qm(1)))) * [ lambda2, -1; -lambda1, 1];

                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,:)).') ).';
            end

        % Flux difference splitting: The trapezoidal rule
        elseif k == 3
            for i = 1:Nx+1
                lambda1L = (ql(i,2)/ql(i,1)) - sqrt(g*ql(i,1));
                lambda2L = (ql(i,2)/ql(i,1)) + sqrt(g*ql(i,1));
                diagL = diag([abs(lambda1L) ; abs(lambda2L)]);
                RL = [ 1, 1; lambda1L, lambda2L ];
                R_invL = (1/sqrt(2*g*ql(i,1))) * [ lambda2L, -1; -lambda1L, 1];

                lambda1R = (qr(i,2)/qr(i,1)) - sqrt(g*qr(i,1));
                lambda2R = (qr(i,2)/qr(i,1)) + sqrt(g*qr(i,1));
                diagR = diag([abs(lambda1R) ; abs(lambda2R)]);
                RR = [ 1, 1; lambda1R, lambda2R ];
                R_invR = (1/sqrt(2*g*qr(i,1))) * [ lambda2R, -1; -lambda1R, 1];

```

```
                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,:)).') ).';
            end

        % Flux difference splitting: Roe's method
        elseif k == 4
            for i = 1:Nx+1
                h_bar = (1/2)*(ql(i,1)+ql(i,1));
                c_hat = sqrt(g*h_bar);
                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)));
                lambda1 = u_hat - c_hat;
                lambda2 = u_hat + c_hat;

                diag_matrix = diag([abs(lambda1) ; abs(lambda2)]);
                R = [ 1, 1; lambda1, lambda2 ];
                R_inv = (1/(2*c_hat)) * [ lambda2, -1; -lambda1, 1];

                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,:)).') ).';
            end

        % The HLL Solver
        elseif k == 5
            for i = 1:Nx+1
                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);

                if h_hat > ql(i,1)
                    pl = sqrt( (1/2)*( (h_hat*(h_hat+ql(i,1)))/(ql(i,1)^2) ) );
                else
                    pl = 1;
                end

                if h_hat > qr(i,1)
                    pr = sqrt( (1/2)*( (h_hat*(h_hat+qr(i,1)))/(qr(i,1)^2) ) );
                else
                    pr = 1;
                end

                sL = (ql(i,2)/ql(i,1)) - (sqrt(g*ql(i,1))*pl);
                sR = (qr(i,2)/qr(i,1)) + (sqrt(g*qr(i,1))*pr);

                if sL >= 0
                    F(i,1) = ql(i,2);
                    F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
                elseif sR <= 0
                    F(i,1) = qr(i,2);
                    F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
                else
                    F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR
- sL );
                    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
);
                end
            end

        % The HLLE Solver
        elseif k == 6
            for i = 1:Nx+1
                h_bar = (1/2)*(ql(i,1)+qr(i,1));
                c_hat = sqrt(g*h_bar);
                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)));
                lambda1 = u_hat - c_hat;
                lambda2 = u_hat + c_hat;

                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))));
```

```
                    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))));

                    if sL >= 0
                        F(i,1) = ql(i,2);
                        F(i,2) = (ql(i,2)/ql(i,1))*ql(i,2)+((g/2)*(ql(i,1)^2));
                    elseif sR <= 0
                        F(i,1) = qr(i,2);
                        F(i,2) = (qr(i,2)/qr(i,1))*qr(i,2)+((g/2)*(qr(i,1)^2));
                    else
                        F(i,1) = ( sR*ql(i,2) - sL*qr(i,2) + sR*sL*(qr(i,1)-ql(i,1)) ) / ( sR - sL );
                        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 );
                    end
                end
            end



            %% Perform time-integration method

            if stage == 1
                du1(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du1(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du1(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        qhelp(i,j)=q(i,j)+du1(i,j);
                    end
                end
            elseif stage == 2
                du2(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du2(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du2(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        qhelp(i,j)=q(i,j)+(1/4)*du1(i,j)+(1/4)*du2(i,j);
                    end
                end
            elseif stage == 3
                du3(1,1)=-(dt/dx)*(F(2,1)-F(1,1));
                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));
                du3(Nx,1)=-(dt/dx)*(F(Nx+1,1)-F(Nx,1));
                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));
                for i=2:Nx-1
                    du3(i,1)=-(dt/dx)*(F(i+1,1)-F(i,1));
                    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));
                end
                for i=3:Nx-2
                    for j=1:2
                        q(i,j)=q(i,j)+(1/6)*du1(i,j)+(1/6)*du2(i,j)+(4/6)*du3(i,j);
                    end
```

```
                end
            end
        end
    end

    if k == 2
        hold on
    end

    h(:,k) = q(:,1);
    u(:,k) = q(:,2)./q(:,1);

    plot(xmid,q(:,1),'-o');
    xlabel('x');
    ylabel('velocity u');
    axis([xmin,xmax,-0.5,0.5]);
    %title(['state\_interpolation=',num2str(state_inter),',  CFL=',num2str(CFL),',  Nx=',
    num2str(Nx),',  Tmax=',num2str(tmax),',  t=',num2str(round(t,4))]);
    grid on

end

%% Define parameters

Nx = 100; % number of finite volumes (= cells)
dx = ( xmax - xmin ) / Nx;  % cell size

x = xmin:dx:xmax;                          % x-coordinates of the cell boundaries
xmid = (1/2)*( x(1:Nx) + x(2:Nx+1) ); % x-coordinates of the cell centres

h_e = zeros(Nx,1);
u_e = zeros(Nx,1);

c_m = 1.205753246885354;
x_a = - tmax * sqrt(g*2);
x_b = t * ( 2 * sqrt(g*2) - 3 * c_m );
x_c = t * ( 2*(c_m^2)*( sqrt(g*2) - c_m) ) / ( (c_m^2) - (g*1) );

for i=1:Nx
    if xmid(i) <= x_a
        h_e(i) = 2;
        u_e(i) = 0;
    elseif xmid(i) <= x_b
        h_e(i) = ( 4 / (9 * g) ) * (( sqrt(g*2) - (xmid(i)/(2*tmax)) )^2);
        u_e(i) = (2/3) * ((xmid(i)/tmax) + sqrt(g*2));
    elseif xmid(i) <= x_c
        h_e(i) = (c_m^2)/g;
        u_e(i) = 2 * ( sqrt(g*2) - c_m );
    else
        h_e(i) = 1;
        u_e(i) = 0;
    end
end



plot(xmid,h_e,'-k');
xlabel('x');
ylabel('velocity u');
ylabel('free surface s');
axis([xmin,xmax,-0.05,0.5]);
axis([xmin,xmax,0.9,2.1]);
%title(['state\_interpolation=',num2str(state_inter),',  CFL=',num2str(CFL),',  Nx=',num2str(
    Nx),',  Tmax=',num2str(tmax),',  t=',num2str(round(t,4))]);
grid on
hold off
legend({'Flux vector splitting','Midpoint rule','Trapezoidal rule','Roe''s method','HLL solver
    ','HLLE solver',''},'Location','northeast','Orientation','vertical')



  %% Begin plotting
    %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(
        flux_eval),',  CFL=',num2str(CFL),',  Nx=',num2str(Nx),',  Tmax=',num2str(tmax),',  t=',
        num2str(round(t,4))]);
412     %     hold on
413     %     grid on
414     %     drawnow
415     %end
416
417
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)
        +2*b(i))/2);
423     %end
424     %energy(count) = energy(count) / Nx;
425
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
```

```
for k=1:6
    disp(k);
    L1_h = 0;
    L2_h = 0;
    L3_h = 0;
    for i=1:Nx
        L1_h = L1_h + abs(h(i,k)-h_e(i));
        L2_h = L2_h + ((h(i,k)-h_e(i))^2);
        if abs(h(i,k)-h_e(i)) > L3_h
            L3_h = abs(h(i,k)-h_e(i));
        end
    end
    disp(L1_h);
    disp(L2_h);
    disp(L3_h);

    L1_u = 0;
    L2_u = 0;
    L3_u = 0;
    for i=1:Nx
        L1_u = L1_u + abs(u(i,k)-u_e(i));
        L2_u = L2_u + ((u(i,k)-u_e(i))^2);
        if abs(u(i,k)-u_e(i)) > L3_u
            L3_u = abs(u(i,k)-u_e(i));
        end
    end
    disp(L1_u);
    disp(L2_u);
    disp(L3_u);
end
```

B.6 Test case 3: Flux vector splitting

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

```
clear all
format long

%% Select conditions

state_inter = 4;



%% Define parameters
Nx = 100;
xmin = -10;
xmax = 10;
dx = (xmax-xmin)/Nx;
x = xmin:dx:xmax;
xmid = (1/2)*(x(1:Nx)+x(2:Nx+1));

Ny = 100;
ymin = -10;
ymax = 10;
dy = (ymax-ymin)/Ny;
y = ymin:dy:ymax;
ymid = (1/2)*(y(1:Ny)+y(2:Ny+1));

CFL = 0.9;
tmax = 15;

g = 1;

break_this_loop = 0;
count = 0;



%% Declare sizes of matrices and vectors to lower running time
F = zeros(Nx+1,Ny,3);
G = zeros(Nx,Ny+1,3);

FL = zeros(Nx+1,Ny,3);
FR = zeros(Nx+1,Ny,3);
GL = zeros(Nx,Ny+1,3);
GR = zeros(Nx,Ny+1,3);

r = zeros(3,1);
q_initial = zeros(3,1);

du1 = zeros(Nx,Ny,3);
du2 = zeros(Nx,Ny,3);
du3 = zeros(Nx,Ny,3);
u = zeros(Nx,Ny,3);
b = zeros(Nx,Ny,1);
b_num = zeros(Nx,Ny,2);



%% Begin initial condition initialization
for i=1:Nx
    for j=1:Ny
        b(i,j) = (1/5)*exp(-(xmid(i)^2)-(ymid(j)^2));
    end
end

for i=2:Nx-1
    b_num(i,1,1)=((b(i+1,1)-b(i-1,1))/(2*dx));
    b_num(i,1,2)=((-3*b(i,1)+4*b(i,2)-b(i,3))/(2*dy));
    b_num(i,Ny,1)=((b(i+1,Ny)-b(i-1,Ny))/(2*dx));
    b_num(i,Ny,2)=((3*b(i,Ny)-4*b(i,Ny-1)+b(i,Ny-2))/(2*dy));
end
for j=2:Ny-1
```

```
    b_num(1,j,1)=((-3*b(1,j)+4*b(2,j)-b(3,j))/(2*dx));
    b_num(1,j,2)=((b(1,j+1)-b(1,j-1))/(2*dy));
    b_num(Nx,j,1)=((3*b(Nx,j)-4*b(Nx-1,j)+b(Nx-2,j))/(2*dx));
    b_num(Nx,j,2)=((b(Nx,j+1)-b(Nx,j-1))/(2*dy));
end
b_num(1,1,1)=((-3*b(1,1)+4*b(2,1)-b(3,1))/(2*dx));
b_num(1,1,2)=((-3*b(1,1)+4*b(1,2)-b(1,3))/(2*dy));
b_num(1,Ny,1)=((-3*b(1,Ny)+4*b(2,Ny)-b(3,Ny))/(2*dx));
b_num(1,Ny,2)=((3*b(1,Ny)-4*b(1,Ny-1)+b(1,Ny-2))/(2*dy));
snapnow;
b_num(Nx,1,1)=((3*b(Nx,1)-4*b(Nx-1,1)+b(Nx-2,1))/(2*dx));
b_num(Nx,1,2)=((-3*b(Nx,1)+4*b(Nx,2)-b(Nx,3))/(2*dy));
b_num(Nx,Ny,1)=((3*b(Nx,Ny)-4*b(Nx-1,Ny)+b(Nx-2,Ny))/(2*dx));
b_num(Nx,Ny,2)=((3*b(Nx,Ny)-4*b(Nx,Ny-1)+b(Nx,Ny-2))/(2*dy));
for i=2:Nx-1
  for j=2:Ny-1
        b_num(i,j,1)=((b(i+1,j)-b(i-1,j))/(2*dx));
        b_num(i,j,2)=((b(i,j+1)-b(i,j-1))/(2*dy));
  end
end

for i=1:Nx
    for j=1:Ny
        %if (xmid(i)^2)+(ymid(j)^2) < 10
        %    u(i,j,1) = 1 - b(i,j);
        %else
        %    u(i,j,1) = 1 - b(i,j);
        %end
        u(i,j,1) = 1 - b(i,j);
        u(i,j,2) = 2*u(i,j,1);
        u(i,j,3) = 0;
    end
end

q_initial(1) = 1;
q_initial(2) = 2;
q_initial(3) = 0;



%% Begin time stepping

t=0;

while t<tmax

    count = count + 1;

  % Adjust time
    max_wave_speed = 0;
    for i = 1:Nx
        for j = 1:Ny
            if abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1)));
            end
            if abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1)));
            end
            if abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1)));
            end
            if abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1)));
            end
        end
    end
    dt = CFL * (min(dx,dy) / (2 * max_wave_speed) );

  t = t+dt;

    if t - tmax > 0
        t = t-dt;
        dt = tmax - t;
```

```
        t = t+dt;
        break_this_loop = 1;
    end



    %% Calculate timestep

uhelp=u;

    for istage=1:3

        %% Begin computation net cell fluxes
        for i=1:Nx+1
            for j=1:Ny
                if i==1
                    FL(1,j,1)=q_initial(1);
                    FL(1,j,2)=q_initial(2);
                    FL(1,j,3)=q_initial(3);
                elseif i==2
                    FL(2,j,1)=(1/2)*(uhelp(1,j,1)+uhelp(2,j,1));
                    FL(2,j,2)=(1/2)*(uhelp(1,j,2)+uhelp(2,j,2));
                    FL(2,j,3)=(1/2)*(uhelp(1,j,3)+uhelp(2,j,3));
                elseif i==Nx+1
                    FL(Nx+1,j,1)=(3/2)*uhelp(Nx,j,1)-(1/2)*uhelp(Nx-1,j,1);
                    FL(Nx+1,j,2)=(3/2)*uhelp(Nx,j,2)-(1/2)*uhelp(Nx-1,j,2);
                    FL(Nx+1,j,3)=(3/2)*uhelp(Nx,j,3)-(1/2)*uhelp(Nx-1,j,3);
                else
                    for k=1:3
                        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) );
                    end
                end
            end
        end

        for i=1:Nx
            for j=1:Ny+1
                if j==1
                    GL(i,1,1)=q_initial(1);
                    GL(i,1,2)=q_initial(2);
                    GL(i,1,3)=q_initial(3);
                elseif j==2
                    GL(i,2,1)=(1/2)*(uhelp(i,1,1)+uhelp(i,2,1));
                    GL(i,2,2)=(1/2)*(uhelp(i,1,2)+uhelp(i,2,2));
                    GL(i,2,3)=(1/2)*(uhelp(i,1,3)+uhelp(i,2,3));
                elseif j==Ny+1
                    GL(i,Ny+1,1)=(3/2)*uhelp(i,Ny,1)-(1/2)*uhelp(i,Ny-1,1);
                    GL(i,Ny+1,2)=(3/2)*uhelp(i,Ny,2)-(1/2)*uhelp(i,Ny-1,2);
                    GL(i,Ny+1,3)=(3/2)*uhelp(i,Ny,3)-(1/2)*uhelp(i,Ny-1,3);
                else
                    for k=1:3
                        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) );
                    end
                end
            end
        end

        for i=1:Nx+1
            for j=1:Ny
                if i==1
                    FR(1,j,1)=(3/2)*uhelp(1,j,1)-(1/2)*uhelp(2,j,1);
                    FR(1,j,2)=(3/2)*uhelp(1,j,2)-(1/2)*uhelp(2,j,2);
                    FR(1,j,3)=(3/2)*uhelp(1,j,3)-(1/2)*uhelp(2,j,3);
                elseif i==Nx
                    FR(Nx,j,1)=(1/2)*(uhelp(Nx-1,j,1)+uhelp(Nx,j,1));
                    FR(Nx,j,2)=(1/2)*(uhelp(Nx-1,j,2)+uhelp(Nx,j,2));
                    FR(Nx,j,3)=(1/2)*(uhelp(Nx-1,j,3)+uhelp(Nx,j,3));
                elseif i==Nx+1
                    FR(Nx+1,j,1)=q_initial(1);
                    FR(Nx+1,j,2)=q_initial(2);
                    FR(Nx+1,j,3)=q_initial(3);
```

```
            else
                for k=1:3
                    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) );
                end
            end
        end
    end

    for i=1:Nx
        for j=1:Ny+1
            if j==1
                GR(i,1,1)=(3/2)*uhelp(i,1,1)-(1/2)*uhelp(i,2,1);
                GR(i,1,2)=(3/2)*uhelp(i,1,2)-(1/2)*uhelp(i,2,2);
                GR(i,1,3)=(3/2)*uhelp(i,1,3)-(1/2)*uhelp(i,2,3);
            elseif j==Ny
                GR(i,Ny,1)=(1/2)*(uhelp(i,Ny-1,1)+uhelp(i,Ny,1));
                GR(i,Ny,2)=(1/2)*(uhelp(i,Ny-1,2)+uhelp(i,Ny,2));
                GR(i,Ny,3)=(1/2)*(uhelp(i,Ny-1,3)+uhelp(i,Ny,3));
            elseif j==Ny+1
                GR(i,Ny+1,1)=q_initial(1);
                GR(i,Ny+1,2)=q_initial(2);
                GR(i,Ny+1,3)=q_initial(3);
            else
                for k=1:3
                    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) );
                end
            end
        end
    end



    %% Flux method: Flux vector splitting
    for i=1:Nx+1
        for j=1:Ny
            lambda1L = (FL(i,j,2)/FL(i,j,1)) - sqrt(g*FL(i,j,1)/2);
            lambda2L = (FL(i,j,2)/FL(i,j,1));
            lambda3L = (FL(i,j,2)/FL(i,j,1)) + sqrt(g*FL(i,j,1)/2);
            diagL = diag([max(lambda1L,0) ; max(lambda2L,0) ; max(lambda3L,0)]);
            RL = [1 0 1; lambda1L 0 lambda3L; FL(i,j,3)/FL(i,j,1) 1 FL(i,j,3)/FL(i,j,1) ];

            lambda1R = (FR(i,j,2)/(FR(i,j,1)+eps)) - sqrt(g*FR(i,j,1)/2);
            lambda2R = (FR(i,j,2)/(FR(i,j,1)+eps));
            lambda3R = (FR(i,j,2)/(FR(i,j,1)+eps)) + sqrt(g*FR(i,j,1)/2);
            diagR = diag([min(lambda1R,0) ; min(lambda2R,0) ; min(lambda3R,0)]);
            RR = [1 0 1; lambda1R 0 lambda3R; FR(i,j,3)/FR(i,j,1) 1 FR(i,j,3)/FR(i,j,1) ];

            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)]);
        end
    end

    for i=1:Nx
        for j=1:Ny+1
            lambda1L = (GL(i,j,3)/GL(i,j,1)) - sqrt(g*GL(i,j,1)/2);
            lambda2L = (GL(i,j,3)/GL(i,j,1));
            lambda3L = (GL(i,j,3)/GL(i,j,1)) + sqrt(g*GL(i,j,1)/2);
            diagL = diag([max(lambda1L,0) ; max(lambda2L,0) ; max(lambda3L,0)]);
            RL = [ 1, 0, 1; GL(i,j,2)/GL(i,j,1), 1, GL(i,j,2)/GL(i,j,1); lambda1L, 0, lambda3L ];

            lambda1R = (GR(i,j,3)/GR(i,j,1)) - sqrt(g*GR(i,j,1)/2);
            lambda2R = (GR(i,j,3)/GR(i,j,1));
            lambda3R = (GR(i,j,3)/GR(i,j,1)) + sqrt(g*GR(i,j,1)/2);
            diagR = diag([min(lambda1R,0) ; min(lambda2R,0) ; min(lambda3R,0)]);
            RR = [ 1, 0, 1; GR(i,j,2)/GR(i,j,1), 1, GR(i,j,2)/GR(i,j,1); lambda1R, 0, lambda3R ];

            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)]);
        end
```

```
        end



        %% Perform time-integration method

        if istage == 1
            for i=1:Nx
                for j=1:Ny
                    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));
                    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);
                    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);
                    for k=1:3
                        uhelp(i,j,k)=u(i,j,k)+du1(i,j,k);
                    end
                end
            end
        elseif istage == 2
            for i=1:Nx
                for j=1:Ny
                    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));
                    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);
                    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);
                    for k=1:3
                        uhelp(i,j,k)=u(i,j,k)+(1/4)*du1(i,j,k)+(1/4)*du2(i,j,k);
                    end
                end
            end
        elseif istage == 3
            for i=1:Nx
                for j=1:Ny
                    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));
                    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);
                    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);
                    for k=1:3
                        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);
                    end
                end
            end
        end
    end



%% Begin plotting
    if mod(count,10) == 0
        set(gcf,'Position',[100 100 600 500])
        hplot = u(:,:,1) + b(:,:);

        %surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
        surf(xmid,ymid,transpose(u(:,:,2)./u(:,:,1)),'EdgeColor','none','LineStyle','none');
        %hold on
        %surf(xmid,ymid,transpose(b(:,:)));
        %view(2)
        %hold off
        xlabel('x');
        ylabel('y');
        zlabel('free surface s');
        xlim([xmin xmax]);
        ylim([ymin ymax]);
        zlim([0.75 1.25]);
        zlim([1.75 2.25]);
        h = colorbar;
        h.Limits = [0.75 1.25];
        title(['Nx=',num2str(Nx),',  Ny=',num2str(Ny),',  Tmax=',num2str(tmax),',  t=',num2str
    (round(t,4))]);
        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(atand(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(atand(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(:,:);
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)
```

```
                j2 = j;
            end
        end
        j3 = 1;
        j4 = 1;
        for j = round(Ny/2):Ny
            if u(Nx,j,1) < u(Nx,j3,1)
                j3 = j;
            end
        end
        for j = round(Ny/2):Ny
            if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
                j4 = j;
            end
        end
        disp('current time is')
        disp(t)
        disp('max')
        disp(atand(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j1,1))
        disp(u(round(11*Nx/20),j2,1))
        disp('min')
        disp(atand(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j3,1))
        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.

```
clear all
format long

%% Select conditions

state_inter = 4;



%% Define parameters
Nx = 100;
xmin = -10;
xmax = 10;
dx = (xmax-xmin)/Nx;
x = xmin:dx:xmax;
xmid = (1/2)*(x(1:Nx)+x(2:Nx+1));

Ny = 100;
ymin = -10;
ymax = 10;
dy = (ymax-ymin)/Ny;
y = ymin:dy:ymax;
ymid = (1/2)*(y(1:Ny)+y(2:Ny+1));

CFL = 0.9;
tmax = 15;

g = 1;

break_this_loop = 0;
count = 0;



%% Declare sizes of matrices and vectors to lower running time
F = zeros(Nx+1,Ny,3);
G = zeros(Nx,Ny+1,3);

FL = zeros(Nx+1,Ny,3);
FR = zeros(Nx+1,Ny,3);
GL = zeros(Nx,Ny+1,3);
GR = zeros(Nx,Ny+1,3);

r = zeros(3,1);
q_initial = zeros(3,1);

du1 = zeros(Nx,Ny,3);
du2 = zeros(Nx,Ny,3);
du3 = zeros(Nx,Ny,3);
u = zeros(Nx,Ny,3);
b = zeros(Nx,Ny,1);
b_num = zeros(Nx,Ny,2);



%% Begin initial condition initialization
for i=1:Nx
    for j=1:Ny
        b(i,j) = (1/5)*exp(-(xmid(i)^2)-(ymid(j)^2));
    end
end

for i=2:Nx-1
    b_num(i,1,1)=((b(i+1,1)-b(i-1,1))/(2*dx));
    b_num(i,1,2)=((-3*b(i,1)+4*b(i,2)-b(i,3))/(2*dy));
    b_num(i,Ny,1)=((b(i+1,Ny)-b(i-1,Ny))/(2*dx));
    b_num(i,Ny,2)=((3*b(i,Ny)-4*b(i,Ny-1)+b(i,Ny-2))/(2*dy));
end
for j=2:Ny-1
```

```
    b_num(1,j,1)=((-3*b(1,j)+4*b(2,j)-b(3,j))/(2*dx));
    b_num(1,j,2)=((b(1,j+1)-b(1,j-1))/(2*dy));
    b_num(Nx,j,1)=((3*b(Nx,j)-4*b(Nx-1,j)+b(Nx-2,j))/(2*dx));
    b_num(Nx,j,2)=((b(Nx,j+1)-b(Nx,j-1))/(2*dy));
end
b_num(1,1,1)=((-3*b(1,1)+4*b(2,1)-b(3,1))/(2*dx));
b_num(1,1,2)=((-3*b(1,1)+4*b(1,2)-b(1,3))/(2*dy));
b_num(1,Ny,1)=((-3*b(1,Ny)+4*b(2,Ny)-b(3,Ny))/(2*dx));
b_num(1,Ny,2)=((3*b(1,Ny)-4*b(1,Ny-1)+b(1,Ny-2))/(2*dy));
snapnow;
b_num(Nx,1,1)=((3*b(Nx,1)-4*b(Nx-1,1)+b(Nx-2,1))/(2*dx));
b_num(Nx,1,2)=((-3*b(Nx,1)+4*b(Nx,2)-b(Nx,3))/(2*dy));
b_num(Nx,Ny,1)=((3*b(Nx,Ny)-4*b(Nx-1,Ny)+b(Nx-2,Ny))/(2*dx));
b_num(Nx,Ny,2)=((3*b(Nx,Ny)-4*b(Nx,Ny-1)+b(Nx,Ny-2))/(2*dy));
for i=2:Nx-1
  for j=2:Ny-1
        b_num(i,j,1)=((b(i+1,j)-b(i-1,j))/(2*dx));
        b_num(i,j,2)=((b(i,j+1)-b(i,j-1))/(2*dy));
  end
end

for i=1:Nx
    for j=1:Ny
        %if (xmid(i)^2)+(ymid(j)^2) < 10
        %    u(i,j,1) = 1 - b(i,j);
        %else
        %    u(i,j,1) = 1 - b(i,j);
        %end
        u(i,j,1) = 1 - b(i,j);
        u(i,j,2) = 2*u(i,j,1);
        u(i,j,3) = 0;
    end
end

q_initial(1) = 1;
q_initial(2) = 2;
q_initial(3) = 0;



%% Begin time stepping

t=0;

while t<tmax

    count = count + 1;

  % Adjust time
    max_wave_speed = 0;
    for i = 1:Nx
        for j = 1:Ny
            if abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1)));
            end
            if abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1)));
            end
            if abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1)));
            end
            if abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1)));
            end
        end
    end
    dt = CFL * (min(dx,dy) / (2 * max_wave_speed) );

  t = t+dt;

    if t - tmax > 0
        t = t-dt;
        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);
```

```
                else
                    for k=1:3
                        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) );
                    end
                end
            end
        end

        for i=1:Nx
            for j=1:Ny+1
                if j==1
                    GR(i,1,1)=(3/2)*uhelp(i,1,1)-(1/2)*uhelp(i,2,1);
                    GR(i,1,2)=(3/2)*uhelp(i,1,2)-(1/2)*uhelp(i,2,2);
                    GR(i,1,3)=(3/2)*uhelp(i,1,3)-(1/2)*uhelp(i,2,3);
                elseif j==Ny
                    GR(i,Ny,1)=(1/2)*(uhelp(i,Ny-1,1)+uhelp(i,Ny,1));
                    GR(i,Ny,2)=(1/2)*(uhelp(i,Ny-1,2)+uhelp(i,Ny,2));
                    GR(i,Ny,3)=(1/2)*(uhelp(i,Ny-1,3)+uhelp(i,Ny,3));
                elseif j==Ny+1
                    GR(i,Ny+1,1)=q_initial(1);
                    GR(i,Ny+1,2)=q_initial(2);
                    GR(i,Ny+1,3)=q_initial(3);
                else
                    for k=1:3
                        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) );
                    end
                end
            end
        end



        %% Flux method: Flux vector splitting
        for i=1:Nx+1
            for j=1:Ny
                qm = FL(i,j,:);
                lambda1 = (qm(2)/qm(1)) - sqrt(g*qm(1));
                lambda2 = (qm(2)/qm(1));
                lambda3 = (qm(2)/qm(1)) + sqrt(g*qm(1));
                diag1 = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);
                R1 = [1 0 1; lambda1 0 lambda3; qm(3)/qm(1) 1 qm(3)/qm(1) ];

                qm = FR(i,j,:);
                lambda1 = (qm(2)/qm(1)) - sqrt(g*qm(1));
                lambda2 = (qm(2)/qm(1));
                lambda3 = (qm(2)/qm(1)) + sqrt(g*qm(1));
                diag2 = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);
                R2 = [1 0 1; lambda1 0 lambda3; qm(3)/qm(1) 1 qm(3)/qm(1) ];

                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)]);
            end
        end

        for i=1:Nx
            for j=1:Ny+1
                qm = GL(i,j,:);
                lambda1 = (qm(3)/qm(1)) - sqrt(g*qm(1));
                lambda2 = (qm(3)/qm(1));
                lambda3 = (qm(3)/qm(1)) + sqrt(g*qm(1));
                diag1 = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);
                R1 = [1 0 1; qm(2)/qm(1) 1 qm(2)/qm(1); lambda1 0 lambda3; ];

                qm = GR(i,j,:);
                lambda1 = (qm(3)/qm(1)) - sqrt(g*qm(1));
                lambda2 = (qm(3)/qm(1));
                lambda3 = (qm(3)/qm(1)) + sqrt(g*qm(1));
                diag2 = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);
```

```
                R2 = [1 0 1; qm(2)/qm(1) 1 qm(2)/qm(1); lambda1 0 lambda3; ];

                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)]);
            end
        end



        %% Perform time-integration method

        if istage == 1
            for i=1:Nx
                for j=1:Ny
                    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));
                    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);
                    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);
                    for k=1:3
                        uhelp(i,j,k)=u(i,j,k)+du1(i,j,k);
                    end
                end
            end
        elseif istage == 2
            for i=1:Nx
                for j=1:Ny
                    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));
                    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);
                    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);
                    for k=1:3
                        uhelp(i,j,k)=u(i,j,k)+(1/4)*du1(i,j,k)+(1/4)*du2(i,j,k);
                    end
                end
            end
        elseif istage == 3
            for i=1:Nx
                for j=1:Ny
                    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));
                    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);
                    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);
                    for k=1:3
                        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);
                    end
                end
            end
        end
    end



%% Begin plotting
    if mod(count,10) == 0
        set(gcf,'Position',[100 100 600 500])
        hplot = u(:,:,1) + b(:,:);

        surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
        %surf(xmid,ymid,transpose(u(:,:,2)./u(:,:,1)),'EdgeColor','none','LineStyle','none');
        %hold on
        %surf(xmid,ymid,transpose(b(:,:)));
        %view(2)
        %hold off
        xlabel('x');
        ylabel('y');
        zlabel('free surface s');
        xlim([xmin xmax]);
```

```
        ylim([ymin ymax]);
        zlim([0.75 1.25]);
        %zlim([1.75 2.25]);
        h = colorbar;
        h.Limits = [0.75 1.25];
        title(['Nx=',num2str(Nx),',  Ny=',num2str(Ny),',  Tmax=',num2str(tmax),',  t=',num2str
    (round(t,4))]);
        drawnow

        % angle measurement
        j1 = 1;
        j2 = 1;
        for j = round(Ny/2):Ny
            if u(Nx,j,1) > u(Nx,j1,1)
                j1 = j;
            end
        end
        for j = round(Ny/2):Ny
            if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)
                j2 = j;
            end
        end
        j3 = 1;
        j4 = 1;
        for j = round(Ny/2):Ny
            if u(Nx,j,1) < u(Nx,j3,1)
                j3 = j;
            end
        end
        for j = round(Ny/2):Ny
            if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
                j4 = j;
            end
        end
        disp('current time is')
        disp(t)
        disp('max')
        disp(atand(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j1,1))
        disp(u(round(11*Nx/20),j2,1))
        disp('min')
        disp(atand(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j3,1))
        disp(u(round(11*Nx/20),j4,1))

    end
end


%% Begin plotting
        set(gcf,'Position',[100 100 600 500])
        hplot = u(:,:,1) + b(:,:);

        surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
        %hold on
        %surf(xmid,ymid,transpose(b(:,:)));
        %view(2)
        %hold off
        xlabel('x');
        ylabel('y');
        zlabel('free surface s');
        xlim([xmin xmax]);
        ylim([ymin ymax]);
        zlim([0.75 1.25]);
        h = colorbar;
        h.Limits = [0.75 1.25];
        drawnow

        % angle measurement
        j1 = 1;
        j2 = 1;
        for j = round(Ny/2):Ny
```

```
            if u(Nx,j,1) > u(Nx,j1,1)
                j1 = j;
            end
        end
        for j = round(Ny/2):Ny
            if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)
                j2 = j;
            end
        end
        j3 = 1;
        j4 = 1;
        for j = round(Ny/2):Ny
            if u(Nx,j,1) < u(Nx,j3,1)
                j3 = j;
            end
        end
        for j = round(Ny/2):Ny
            if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
                j4 = j;
            end
        end
        disp('current time is')
        disp(t)
        disp('max')
        disp(atand(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j1,1))
        disp(u(round(11*Nx/20),j2,1))
        disp('min')
        disp(atand(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j3,1))
        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.

```
clear all
format long

%% Select conditions

state_inter = 4;



%% Define parameters
Nx = 100;
xmin = -10;
xmax = 10;
dx = (xmax-xmin)/Nx;
x = xmin:dx:xmax;
xmid = (1/2)*(x(1:Nx)+x(2:Nx+1));

Ny = 100;
ymin = -10;
ymax = 10;
dy = (ymax-ymin)/Ny;
y = ymin:dy:ymax;
ymid = (1/2)*(y(1:Ny)+y(2:Ny+1));

CFL = 0.9;
tmax = 15;

g = 1;

break_this_loop = 0;
count = 0;



%% Declare sizes of matrices and vectors to lower running time
F = zeros(Nx+1,Ny,3);
G = zeros(Nx,Ny+1,3);

FL = zeros(Nx+1,Ny,3);
FR = zeros(Nx+1,Ny,3);
GL = zeros(Nx,Ny+1,3);
GR = zeros(Nx,Ny+1,3);

r = zeros(3,1);
q_initial = zeros(3,1);

du1 = zeros(Nx,Ny,3);
du2 = zeros(Nx,Ny,3);
du3 = zeros(Nx,Ny,3);
u = zeros(Nx,Ny,3);
b = zeros(Nx,Ny,1);
b_num = zeros(Nx,Ny,2);



%% Begin initial condition initialization
for i=1:Nx
    for j=1:Ny
        b(i,j) = (1/5)*exp(-(xmid(i)^2)-(ymid(j)^2));
    end
end

for i=2:Nx-1
    b_num(i,1,1)=((b(i+1,1)-b(i-1,1))/(2*dx));
    b_num(i,1,2)=((-3*b(i,1)+4*b(i,2)-b(i,3))/(2*dy));
    b_num(i,Ny,1)=((b(i+1,Ny)-b(i-1,Ny))/(2*dx));
    b_num(i,Ny,2)=((3*b(i,Ny)-4*b(i,Ny-1)+b(i,Ny-2))/(2*dy));
end
for j=2:Ny-1
```

```
    b_num(1,j,1)=((-3*b(1,j)+4*b(2,j)-b(3,j))/(2*dx));
    b_num(1,j,2)=((b(1,j+1)-b(1,j-1))/(2*dy));
    b_num(Nx,j,1)=((3*b(Nx,j)-4*b(Nx-1,j)+b(Nx-2,j))/(2*dx));
    b_num(Nx,j,2)=((b(Nx,j+1)-b(Nx,j-1))/(2*dy));
end
b_num(1,1,1)=((-3*b(1,1)+4*b(2,1)-b(3,1))/(2*dx));
b_num(1,1,2)=((-3*b(1,1)+4*b(1,2)-b(1,3))/(2*dy));
b_num(1,Ny,1)=((-3*b(1,Ny)+4*b(2,Ny)-b(3,Ny))/(2*dx));
b_num(1,Ny,2)=((3*b(1,Ny)-4*b(1,Ny-1)+b(1,Ny-2))/(2*dy));
snapnow;
b_num(Nx,1,1)=((3*b(Nx,1)-4*b(Nx-1,1)+b(Nx-2,1))/(2*dx));
b_num(Nx,1,2)=((-3*b(Nx,1)+4*b(Nx,2)-b(Nx,3))/(2*dy));
b_num(Nx,Ny,1)=((3*b(Nx,Ny)-4*b(Nx-1,Ny)+b(Nx-2,Ny))/(2*dx));
b_num(Nx,Ny,2)=((3*b(Nx,Ny)-4*b(Nx,Ny-1)+b(Nx,Ny-2))/(2*dy));
for i=2:Nx-1
  for j=2:Ny-1
        b_num(i,j,1)=((b(i+1,j)-b(i-1,j))/(2*dx));
        b_num(i,j,2)=((b(i,j+1)-b(i,j-1))/(2*dy));
  end
end

for i=1:Nx
    for j=1:Ny
        %if (xmid(i)^2)+(ymid(j)^2) < 10
        %    u(i,j,1) = 1 - b(i,j);
        %else
        %    u(i,j,1) = 1 - b(i,j);
        %end
        u(i,j,1) = 1 - b(i,j);
        u(i,j,2) = 2*u(i,j,1);
        u(i,j,3) = 0;
    end
end

q_initial(1) = 1;
q_initial(2) = 2;
q_initial(3) = 0;



%% Begin time stepping

t=0;

while t<tmax

    count = count + 1;

  % Adjust time
    max_wave_speed = 0;
    for i = 1:Nx
        for j = 1:Ny
            if abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1)));
            end
            if abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1)));
            end
            if abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1)));
            end
            if abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1)));
            end
        end
    end
    dt = CFL * (min(dx,dy) / (2 * max_wave_speed) );

  t = t+dt;

    if t - tmax > 0
        t = t-dt;
        dt = tmax - t;
```

```
        t = t+dt;
        break_this_loop = 1;
    end



    %% Calculate timestep

uhelp=u;

    for istage=1:3

        %% Begin computation net cell fluxes
        for i=1:Nx+1
            for j=1:Ny
                if i==1
                    FL(1,j,1)=q_initial(1);
                    FL(1,j,2)=q_initial(2);
                    FL(1,j,3)=q_initial(3);
                elseif i==2
                    FL(2,j,1)=(1/2)*(uhelp(1,j,1)+uhelp(2,j,1));
                    FL(2,j,2)=(1/2)*(uhelp(1,j,2)+uhelp(2,j,2));
                    FL(2,j,3)=(1/2)*(uhelp(1,j,3)+uhelp(2,j,3));
                elseif i==Nx+1
                    FL(Nx+1,j,1)=(3/2)*uhelp(Nx,j,1)-(1/2)*uhelp(Nx-1,j,1);
                    FL(Nx+1,j,2)=(3/2)*uhelp(Nx,j,2)-(1/2)*uhelp(Nx-1,j,2);
                    FL(Nx+1,j,3)=(3/2)*uhelp(Nx,j,3)-(1/2)*uhelp(Nx-1,j,3);
                else
                    for k=1:3
                        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) );
                    end
                end
            end
        end

        for i=1:Nx
            for j=1:Ny+1
                if j==1
                    GL(i,1,1)=q_initial(1);
                    GL(i,1,2)=q_initial(2);
                    GL(i,1,3)=q_initial(3);
                elseif j==2
                    GL(i,2,1)=(1/2)*(uhelp(i,1,1)+uhelp(i,2,1));
                    GL(i,2,2)=(1/2)*(uhelp(i,1,2)+uhelp(i,2,2));
                    GL(i,2,3)=(1/2)*(uhelp(i,1,3)+uhelp(i,2,3));
                elseif j==Ny+1
                    GL(i,Ny+1,1)=(3/2)*uhelp(i,Ny,1)-(1/2)*uhelp(i,Ny-1,1);
                    GL(i,Ny+1,2)=(3/2)*uhelp(i,Ny,2)-(1/2)*uhelp(i,Ny-1,2);
                    GL(i,Ny+1,3)=(3/2)*uhelp(i,Ny,3)-(1/2)*uhelp(i,Ny-1,3);
                else
                    for k=1:3
                        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) );
                    end
                end
            end
        end

        for i=1:Nx+1
            for j=1:Ny
                if i==1
                    FR(1,j,1)=(3/2)*uhelp(1,j,1)-(1/2)*uhelp(2,j,1);
                    FR(1,j,2)=(3/2)*uhelp(1,j,2)-(1/2)*uhelp(2,j,2);
                    FR(1,j,3)=(3/2)*uhelp(1,j,3)-(1/2)*uhelp(2,j,3);
                elseif i==Nx
                    FR(Nx,j,1)=(1/2)*(uhelp(Nx-1,j,1)+uhelp(Nx,j,1));
                    FR(Nx,j,2)=(1/2)*(uhelp(Nx-1,j,2)+uhelp(Nx,j,2));
                    FR(Nx,j,3)=(1/2)*(uhelp(Nx-1,j,3)+uhelp(Nx,j,3));
                elseif i==Nx+1
                    FR(Nx+1,j,1)=q_initial(1);
                    FR(Nx+1,j,2)=q_initial(2);
                    FR(Nx+1,j,3)=q_initial(3);
```

```
            else
                for k=1:3
                    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) );
                end
            end
        end
    end

    for i=1:Nx
        for j=1:Ny+1
            if j==1
                GR(i,1,1)=(3/2)*uhelp(i,1,1)-(1/2)*uhelp(i,2,1);
                GR(i,1,2)=(3/2)*uhelp(i,1,2)-(1/2)*uhelp(i,2,2);
                GR(i,1,3)=(3/2)*uhelp(i,1,3)-(1/2)*uhelp(i,2,3);
            elseif j==Ny
                GR(i,Ny,1)=(1/2)*(uhelp(i,Ny-1,1)+uhelp(i,Ny,1));
                GR(i,Ny,2)=(1/2)*(uhelp(i,Ny-1,2)+uhelp(i,Ny,2));
                GR(i,Ny,3)=(1/2)*(uhelp(i,Ny-1,3)+uhelp(i,Ny,3));
            elseif j==Ny+1
                GR(i,Ny+1,1)=q_initial(1);
                GR(i,Ny+1,2)=q_initial(2);
                GR(i,Ny+1,3)=q_initial(3);
            else
                for k=1:3
                    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) );
                end
            end
        end
    end



    %% Flux method: Flux vector splitting
    for i=1:Nx+1
        for j=1:Ny
            qm = (1/2)*( FL(i,j,:) + FR(i,j,:) );
            lambda1 = (qm(2)/qm(1)) - sqrt(g*qm(1));
            lambda2 = (qm(2)/qm(1));
            lambda3 = (qm(2)/qm(1)) + sqrt(g*qm(1));
            diagm = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);
            Rm = [1 0 1; lambda1 0 lambda3; qm(3)/qm(1) 1 qm(3)/qm(1) ];

            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)]);
        end
    end

    for i=1:Nx
        for j=1:Ny+1
            qm = (1/2)*( GL(i,j,:) + GR(i,j,:) );
            lambda1 = (qm(3)/qm(1)) - sqrt(g*qm(1));
            lambda2 = (qm(3)/qm(1));
            lambda3 = (qm(3)/qm(1)) + sqrt(g*qm(1));
            diagm = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);
            Rm = [1 0 1; qm(2)/qm(1) 1 qm(2)/qm(1); lambda1 0 lambda3; ];

            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)]);
        end
    end



    %% Perform time-integration method

    if istage == 1
        for i=1:Nx
```

```
                for j=1:Ny
                    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));
                    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);
                    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);
                    for k=1:3
                        uhelp(i,j,k)=u(i,j,k)+du1(i,j,k);
                    end
                end
            end
        elseif istage == 2
            for i=1:Nx
                for j=1:Ny
                    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));
                    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);
                    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);
                    for k=1:3
                        uhelp(i,j,k)=u(i,j,k)+(1/4)*du1(i,j,k)+(1/4)*du2(i,j,k);
                    end
                end
            end
        elseif istage == 3
            for i=1:Nx
                for j=1:Ny
                    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));
                    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);
                    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);
                    for k=1:3
                        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);
                    end
                end
            end
        end
    end



%% Begin plotting
    if mod(count,10) == 0
        set(gcf,'Position',[100 100 600 500])
        hplot = u(:,:,1) + b(:,:);

        surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
        %surf(xmid,ymid,transpose(u(:,:,2)./u(:,:,1)),'EdgeColor','none','LineStyle','none');
        %hold on
        %surf(xmid,ymid,transpose(b(:,:)));
        %view(2)
        %hold off
        xlabel('x');
        ylabel('y');
        zlabel('free surface s');
        xlim([xmin xmax]);
        ylim([ymin ymax]);
        zlim([0.75 1.25]);
        %zlim([1.75 2.25]);
        h = colorbar;
        h.Limits = [0.75 1.25];
        title(['Nx=',num2str(Nx),',  Ny=',num2str(Ny),',  Tmax=',num2str(tmax),',  t=',num2str
    (round(t,4))]);
        drawnow

        % angle measurement
        j1 = 1;
        j2 = 1;
        for j = round(Ny/2):Ny
            if u(Nx,j,1) > u(Nx,j1,1)
                j1 = j;
            end
```

```
        end
        for j = round(Ny/2):Ny
            if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)
                j2 = j;
            end
        end
        j3 = 1;
        j4 = 1;
        for j = round(Ny/2):Ny
            if u(Nx,j,1) < u(Nx,j3,1)
                j3 = j;
            end
        end
        for j = round(Ny/2):Ny
            if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
                j4 = j;
            end
        end
        disp('current time is')
        disp(t)
        disp('max')
        disp(atand(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j1,1))
        disp(u(round(11*Nx/20),j2,1))
        disp('min')
        disp(atand(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j3,1))
        disp(u(round(11*Nx/20),j4,1))

    end
end



%% Begin plotting
        set(gcf,'Position',[100 100 600 500])
        hplot = u(:,:,1) + b(:,:);

        surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
        %hold on
        %surf(xmid,ymid,transpose(b(:,:)));
        %view(2)
        %hold off
        xlabel('x');
        ylabel('y');
        zlabel('free surface s');
        xlim([xmin xmax]);
        ylim([ymin ymax]);
        zlim([0.75 1.25]);
        h = colorbar;
        h.Limits = [0.75 1.25];
        title(['Nx=',num2str(Nx),',  Ny=',num2str(Ny),',  Tmax=',num2str(tmax),',  t=',num2str
    (round(t,4))]);
        drawnow

        % angle measurement
        j1 = 1;
        j2 = 1;
        for j = round(Ny/2):Ny
            if u(Nx,j,1) > u(Nx,j1,1)
                j1 = j;
            end
        end
        for j = round(Ny/2):Ny
            if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)
                j2 = j;
            end
        end
        j3 = 1;
        j4 = 1;
        for j = round(Ny/2):Ny
            if u(Nx,j,1) < u(Nx,j3,1)
                j3 = j;
```

```
            end
        end
        for j = round(Ny/2):Ny
            if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
                j4 = j;
            end
        end
        disp('current time is')
        disp(t)
        disp('max')
        disp(atand(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j1,1))
        disp(u(round(11*Nx/20),j2,1))
        disp('min')
        disp(atand(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j3,1))
        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.

```
clear all
format long

%% Select conditions

state_inter = 4;



%% Define parameters
Nx = 100;
xmin = -10;
xmax = 10;
dx = (xmax-xmin)/Nx;
x = xmin:dx:xmax;
xmid = (1/2)*(x(1:Nx)+x(2:Nx+1));

Ny = 100;
ymin = -10;
ymax = 10;
dy = (ymax-ymin)/Ny;
y = ymin:dy:ymax;
ymid = (1/2)*(y(1:Ny)+y(2:Ny+1));

CFL = 0.9;
tmax = 5;

g = 1;

break_this_loop = 0;
count = 0;



%% Declare sizes of matrices and vectors to lower running time
F = zeros(Nx+1,Ny,3);
G = zeros(Nx,Ny+1,3);

FL = zeros(Nx+1,Ny,3);
FR = zeros(Nx+1,Ny,3);
GL = zeros(Nx,Ny+1,3);
GR = zeros(Nx,Ny+1,3);

r = zeros(3,1);
q_initial = zeros(3,1);

du1 = zeros(Nx,Ny,3);
du2 = zeros(Nx,Ny,3);
du3 = zeros(Nx,Ny,3);
u = zeros(Nx,Ny,3);
b = zeros(Nx,Ny,1);
b_num = zeros(Nx,Ny,2);



%% Begin initial condition initialization
for i=1:Nx
    for j=1:Ny
        b(i,j) = (1/5)*exp(-(xmid(i)^2)-(ymid(j)^2));
    end
end

for i=2:Nx-1
    b_num(i,1,1)=((b(i+1,1)-b(i-1,1))/(2*dx));
    b_num(i,1,2)=((-3*b(i,1)+4*b(i,2)-b(i,3))/(2*dy));
    b_num(i,Ny,1)=((b(i+1,Ny)-b(i-1,Ny))/(2*dx));
    b_num(i,Ny,2)=((3*b(i,Ny)-4*b(i,Ny-1)+b(i,Ny-2))/(2*dy));
end
for j=2:Ny-1
```

```
    b_num(1,j,1)=((-3*b(1,j)+4*b(2,j)-b(3,j))/(2*dx));
    b_num(1,j,2)=((b(1,j+1)-b(1,j-1))/(2*dy));
    b_num(Nx,j,1)=((3*b(Nx,j)-4*b(Nx-1,j)+b(Nx-2,j))/(2*dx));
    b_num(Nx,j,2)=((b(Nx,j+1)-b(Nx,j-1))/(2*dy));
end
b_num(1,1,1)=((-3*b(1,1)+4*b(2,1)-b(3,1))/(2*dx));
b_num(1,1,2)=((-3*b(1,1)+4*b(1,2)-b(1,3))/(2*dy));
b_num(1,Ny,1)=((-3*b(1,Ny)+4*b(2,Ny)-b(3,Ny))/(2*dx));
b_num(1,Ny,2)=((3*b(1,Ny)-4*b(1,Ny-1)+b(1,Ny-2))/(2*dy));
snapnow;
b_num(Nx,1,1)=((3*b(Nx,1)-4*b(Nx-1,1)+b(Nx-2,1))/(2*dx));
b_num(Nx,1,2)=((-3*b(Nx,1)+4*b(Nx,2)-b(Nx,3))/(2*dy));
b_num(Nx,Ny,1)=((3*b(Nx,Ny)-4*b(Nx-1,Ny)+b(Nx-2,Ny))/(2*dx));
b_num(Nx,Ny,2)=((3*b(Nx,Ny)-4*b(Nx,Ny-1)+b(Nx,Ny-2))/(2*dy));
for i=2:Nx-1
  for j=2:Ny-1
        b_num(i,j,1)=((b(i+1,j)-b(i-1,j))/(2*dx));
        b_num(i,j,2)=((b(i,j+1)-b(i,j-1))/(2*dy));
  end
end

for i=1:Nx
    for j=1:Ny
        %if (xmid(i)^2)+(ymid(j)^2) < 10
        %    u(i,j,1) = 1 - b(i,j);
        %else
        %    u(i,j,1) = 1 - b(i,j);
        %end
        u(i,j,1) = 1 - b(i,j);
        u(i,j,2) = 2*u(i,j,1);
        u(i,j,3) = 0;
    end
end

q_initial(1) = 1;
q_initial(2) = 2;
q_initial(3) = 0;



%% Begin time stepping

t=0;

while t<tmax

    count = count + 1;

  % Adjust time
    max_wave_speed = 0;
    for i = 1:Nx
        for j = 1:Ny
            if abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,2)/u(i,j,1)) + sqrt(g*u(i,j,1)));
            end
            if abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,2)/u(i,j,1)) - sqrt(g*u(i,j,1)));
            end
            if abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,3)/u(i,j,1)) + sqrt(g*u(i,j,1)));
            end
            if abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1))) > max_wave_speed
                max_wave_speed = abs((u(i,j,3)/u(i,j,1)) - sqrt(g*u(i,j,1)));
            end
        end
    end
    dt = CFL * (min(dx,dy) / (2 * max_wave_speed) );

  t = t+dt;

    if t - tmax > 0
        t = t-dt;
        dt = tmax - t;
```

```
        t = t+dt;
        break_this_loop = 1;
    end



    %% Calculate timestep

uhelp=u;

    for istage=1:3

        %% Begin computation net cell fluxes
        for i=1:Nx+1
            for j=1:Ny
                if i==1
                    FL(1,j,1)=q_initial(1);
                    FL(1,j,2)=q_initial(2);
                    FL(1,j,3)=q_initial(3);
                elseif i==2
                    FL(2,j,1)=(1/2)*(uhelp(1,j,1)+uhelp(2,j,1));
                    FL(2,j,2)=(1/2)*(uhelp(1,j,2)+uhelp(2,j,2));
                    FL(2,j,3)=(1/2)*(uhelp(1,j,3)+uhelp(2,j,3));
                elseif i==Nx+1
                    FL(Nx+1,j,1)=(3/2)*uhelp(Nx,j,1)-(1/2)*uhelp(Nx-1,j,1);
                    FL(Nx+1,j,2)=(3/2)*uhelp(Nx,j,2)-(1/2)*uhelp(Nx-1,j,2);
                    FL(Nx+1,j,3)=(3/2)*uhelp(Nx,j,3)-(1/2)*uhelp(Nx-1,j,3);
                else
                    for k=1:3
                        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) );
                    end
                end
            end
        end

        for i=1:Nx
            for j=1:Ny+1
                if j==1
                    GL(i,1,1)=q_initial(1);
                    GL(i,1,2)=q_initial(2);
                    GL(i,1,3)=q_initial(3);
                elseif j==2
                    GL(i,2,1)=(1/2)*(uhelp(i,1,1)+uhelp(i,2,1));
                    GL(i,2,2)=(1/2)*(uhelp(i,1,2)+uhelp(i,2,2));
                    GL(i,2,3)=(1/2)*(uhelp(i,1,3)+uhelp(i,2,3));
                elseif j==Ny+1
                    GL(i,Ny+1,1)=(3/2)*uhelp(i,Ny,1)-(1/2)*uhelp(i,Ny-1,1);
                    GL(i,Ny+1,2)=(3/2)*uhelp(i,Ny,2)-(1/2)*uhelp(i,Ny-1,2);
                    GL(i,Ny+1,3)=(3/2)*uhelp(i,Ny,3)-(1/2)*uhelp(i,Ny-1,3);
                else
                    for k=1:3
                        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) );
                    end
                end
            end
        end

        for i=1:Nx+1
            for j=1:Ny
                if i==1
                    FR(1,j,1)=(3/2)*uhelp(1,j,1)-(1/2)*uhelp(2,j,1);
                    FR(1,j,2)=(3/2)*uhelp(1,j,2)-(1/2)*uhelp(2,j,2);
                    FR(1,j,3)=(3/2)*uhelp(1,j,3)-(1/2)*uhelp(2,j,3);
                elseif i==Nx
                    FR(Nx,j,1)=(1/2)*(uhelp(Nx-1,j,1)+uhelp(Nx,j,1));
                    FR(Nx,j,2)=(1/2)*(uhelp(Nx-1,j,2)+uhelp(Nx,j,2));
                    FR(Nx,j,3)=(1/2)*(uhelp(Nx-1,j,3)+uhelp(Nx,j,3));
                elseif i==Nx+1
                    FR(Nx+1,j,1)=q_initial(1);
                    FR(Nx+1,j,2)=q_initial(2);
                    FR(Nx+1,j,3)=q_initial(3);
```

```
            else
                for k=1:3
                    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) );
                end
            end
        end
    end

    for i=1:Nx
        for j=1:Ny+1
            if j==1
                GR(i,1,1)=(3/2)*uhelp(i,1,1)-(1/2)*uhelp(i,2,1);
                GR(i,1,2)=(3/2)*uhelp(i,1,2)-(1/2)*uhelp(i,2,2);
                GR(i,1,3)=(3/2)*uhelp(i,1,3)-(1/2)*uhelp(i,2,3);
            elseif j==Ny
                GR(i,Ny,1)=(1/2)*(uhelp(i,Ny-1,1)+uhelp(i,Ny,1));
                GR(i,Ny,2)=(1/2)*(uhelp(i,Ny-1,2)+uhelp(i,Ny,2));
                GR(i,Ny,3)=(1/2)*(uhelp(i,Ny-1,3)+uhelp(i,Ny,3));
            elseif j==Ny+1
                GR(i,Ny+1,1)=q_initial(1);
                GR(i,Ny+1,2)=q_initial(2);
                GR(i,Ny+1,3)=q_initial(3);
            else
                for k=1:3
                    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) );
                end
            end
        end
    end



    %% Flux method: Flux vector splitting
    for i=1:Nx+1
        for j=1:Ny
            h_bar = (1/2)*(FL(i,j,1)+FR(i,j,1));
            c_hat = sqrt(g*h_bar);
            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)));
            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)));

            lambda1 = (u_hat) - c_hat;
            lambda2 = (u_hat);
            lambda3 = (u_hat) + c_hat;
            diagm = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);
            Rm = [1 0 1; lambda1 0 lambda3; v_hat 1 v_hat ];

            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)]);
        end
    end

    for i=1:Nx
        for j=1:Ny+1
            h_bar = (1/2)*(GL(i,j,1)+GR(i,j,1));
            c_hat = sqrt(g*h_bar);
            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)));
            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)));

            lambda1 = (v_hat) - c_hat;
            lambda2 = (v_hat);
            lambda3 = (v_hat) + c_hat;
            diagm = diag([abs(lambda1) ; abs(lambda2) ; abs(lambda3)]);
            Rm = [1 0 1; u_hat 1 u_hat; lambda1 0 lambda3; ];

```

```
                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)]);
            end
        end



        %% Perform time-integration method

        if istage == 1
            for i=1:Nx
                for j=1:Ny
                    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));
                    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);
                    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);
                    for k=1:3
                        uhelp(i,j,k)=u(i,j,k)+du1(i,j,k);
                    end
                end
            end
        elseif istage == 2
            for i=1:Nx
                for j=1:Ny
                    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));
                    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);
                    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);
                    for k=1:3
                        uhelp(i,j,k)=u(i,j,k)+(1/4)*du1(i,j,k)+(1/4)*du2(i,j,k);
                    end
                end
            end
        elseif istage == 3
            for i=1:Nx
                for j=1:Ny
                    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));
                    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);
                    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);
                    for k=1:3
                        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);
                    end
                end
            end
        end
    end



%% Begin plotting
    if mod(count,10) == 0
        set(gcf,'Position',[100 100 600 500])
        hplot = u(:,:,1) + b(:,:);

        surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
        %surf(xmid,ymid,transpose(u(:,:,2)./u(:,:,1)),'EdgeColor','none','LineStyle','none');
        %hold on
        %surf(xmid,ymid,transpose(b(:,:)));
        %view(2)
        %hold off
        xlabel('x');
        ylabel('y');
        zlabel('free surface s');
        xlim([xmin xmax]);
        ylim([ymin ymax]);
        zlim([0.75 1.25]);
        %zlim([1.75 2.25]);
```

```
        h = colorbar;
        h.Limits = [0.75 1.25];
        title(['Nx=',num2str(Nx),',  Ny=',num2str(Ny),',  Tmax=',num2str(tmax),',  t=',num2str
    (round(t,4))]);
        drawnow

        % angle measurement
        j1 = 1;
        j2 = 1;
        for j = round(Ny/2):Ny
            if u(Nx,j,1) > u(Nx,j1,1)
                j1 = j;
            end
        end
        for j = round(Ny/2):Ny
            if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)
                j2 = j;
            end
        end
        j3 = 1;
        j4 = 1;
        for j = round(Ny/2):Ny
            if u(Nx,j,1) < u(Nx,j3,1)
                j3 = j;
            end
        end
        for j = round(Ny/2):Ny
            if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
                j4 = j;
            end
        end
        disp('current time is')
        disp(t)
        disp('max')
        disp(atand(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j1,1))
        disp(u(round(11*Nx/20),j2,1))
        disp('min')
        disp(atand(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j3,1))
        disp(u(round(11*Nx/20),j4,1))

    end
end



%% Begin plotting
        set(gcf,'Position',[100 100 600 500])
        hplot = u(:,:,1) + b(:,:);

        surf(xmid,ymid,transpose(hplot(:,:)),'EdgeColor','none','LineStyle','none');
        %hold on
        %surf(xmid,ymid,transpose(b(:,:)));
        %view(2)
        %hold off
        xlabel('x');
        ylabel('y');
        zlabel('free surface s');
        xlim([xmin xmax]);
        ylim([ymin ymax]);
        zlim([0.75 1.25]);
        h = colorbar;
        h.Limits = [0.75 1.25];
        %title(['Nx=',num2str(Nx),',  Ny=',num2str(Ny),',  Tmax=',num2str(tmax),',  t=',
    num2str(round(t,4))]);
        drawnow

        % angle measurement
        j1 = 1;
        j2 = 1;
        for j = round(Ny/2):Ny
            if u(Nx,j,1) > u(Nx,j1,1)
```

```
                j1 = j;
            end
        end
        for j = round(Ny/2):Ny
            if u(round(11*Nx/20),j,1) > u(round(11*Nx/20),j2,1)
                j2 = j;
            end
        end
        j3 = 1;
        j4 = 1;
        for j = round(Ny/2):Ny
            if u(Nx,j,1) < u(Nx,j3,1)
                j3 = j;
            end
        end
        for j = round(Ny/2):Ny
            if u(round(11*Nx/20),j,1) < u(round(11*Nx/20),j4,1)
                j4 = j;
            end
        end
        disp('current time is')
        disp(t)
        disp('max')
        disp(atand(abs(ymid(j1)-ymid(j2))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j1,1))
        disp(u(round(11*Nx/20),j2,1))
        disp('min')
        disp(atand(abs(ymid(j3)-ymid(j4))/abs(xmid(Nx)-xmid(round(11*Nx/20)))))
        disp(u(Nx,j3,1))
        disp(u(round(11*Nx/20),j4,1))
```