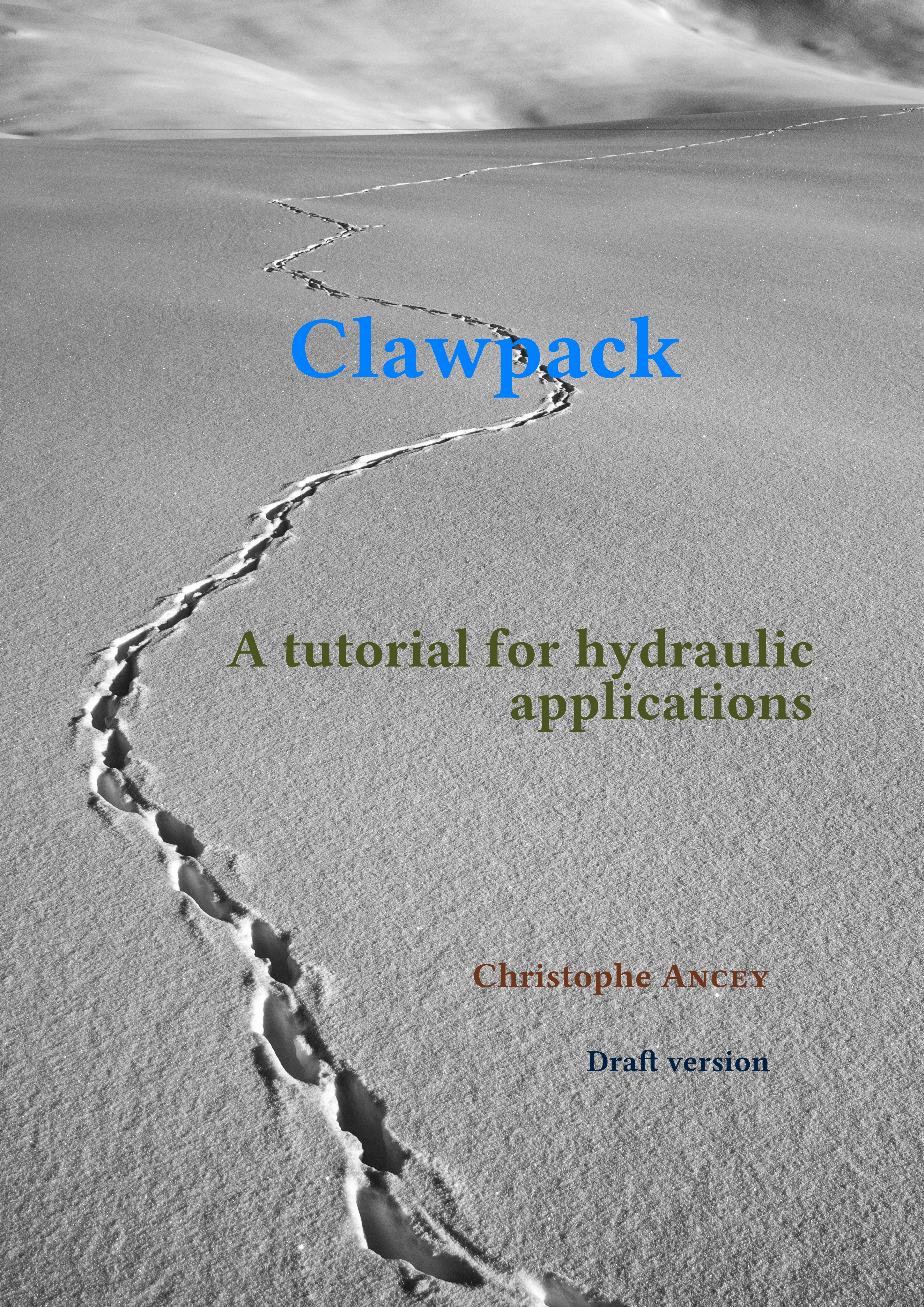


Clawpack

A tutorial for hydraulic applications

Christophe Ancey

Draft version

Christophe Ancey,

EPFL, ENAC/IIC/LHE,

Ecublens, CH-1015 Lausanne, Switzerland

christophe.ancey@epfl.ch, en.ancey.ch, lhe.epfl.ch

Clawpack tutorial / C. Ancey

version 1.1 from 17 June 2020, Lausanne

version 1.2 from 20 April 2021, Lausanne

version 1.3 from 31 July 2023, Lausanne

version 1.4 from 3 January 2024, Lausanne

***Picture credit*. Cover page** : en route vers les monts Tellier (VS). **Chapter 1**: William Turner, *The Falls of the Rhine at Schaffhausen* (Courtauld collection, London). **Chapter 2**: William Turner, *dawn after the wreck*(Courtauld collection, London). **Chapter 3**: William Turner, *Lausanne (cathedral and bridge)* (Tate Britain, London). **Chapter 4**: William Turner, *The fall of anarchy* (Tate Britain, London). **Chapter 5**: William Turner, *The Dogano, San Giorgio, Citella, from the Steps of the Europa hotel* (Tate Britain, London). **Chapter 6**: William Turner, *Sea disaster* (Tate Britain, London). **Chapter 7**: William Turner, *The Parting of Hero and Leander* (National Gallery, London). **Bibliography**: William Turner, *Lausanne (War: The Exile and the Rock Limpet* (Tate Britain, London). **Index**: William Turner, *Lausanne (Two Figures on a Beach with a Boat* (Tate Britain, London).

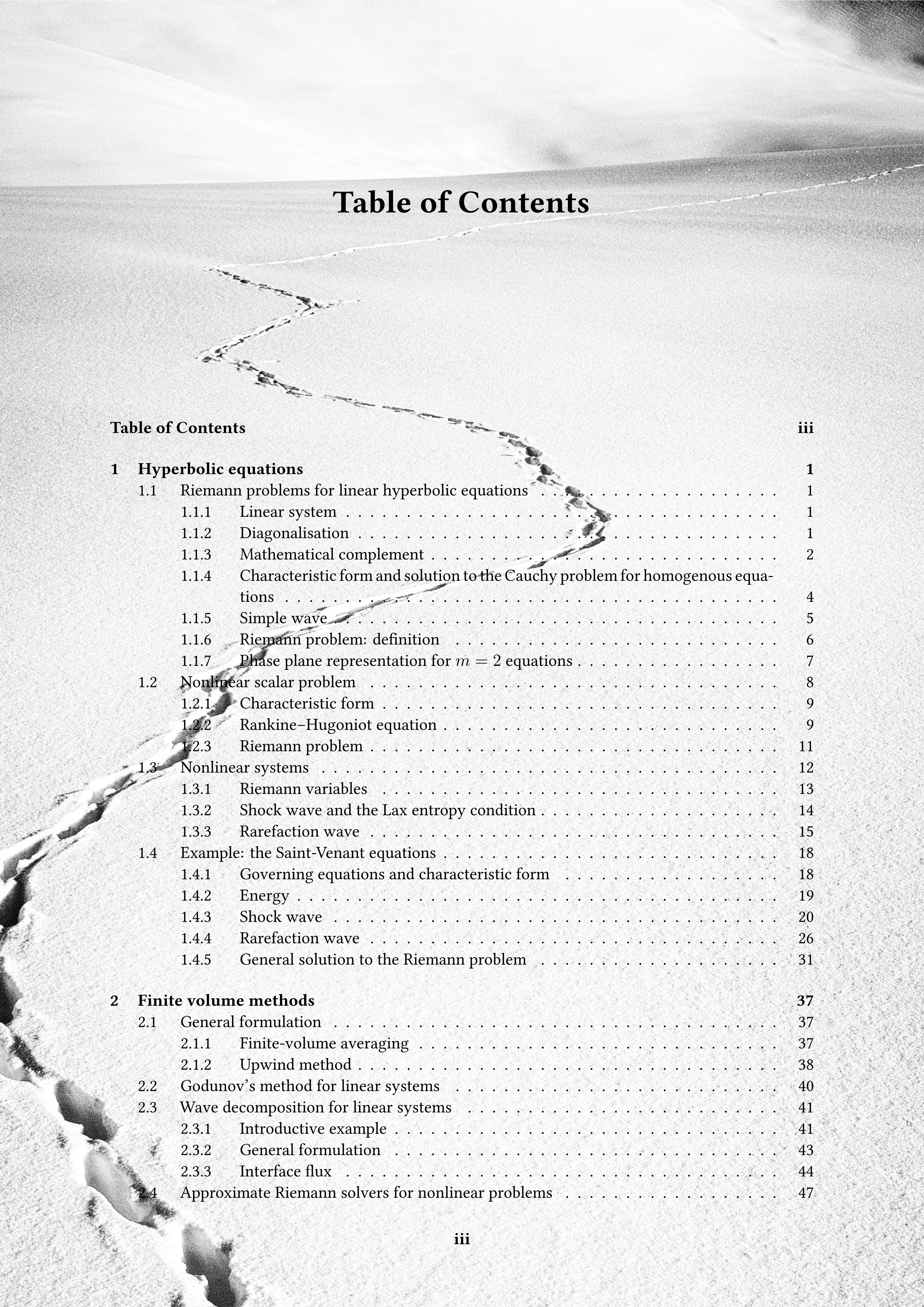


Table of Contents

Foreword

This tutorial is primarily based on the material written by Randall LeVeque and his collaborators.

References

Hyperbolic equation theory is described in a few books, including:

- Numerical Methods for Conservation Laws (LeVeque, 1992)
- Finite Volume Methods for Hyperbolic Problems (LeVeque, 2002)
- Riemann Problems and Jupyter Solutions (Ketcheson *et al.*, 2020)

Some review papers can give an overview of the Clawpack approach (Berger *et al.*, 2011; Ketcheson *et al.*, 2012, 2013; Mandli *et al.*, 2016).

Online material

Clawpack can be downloaded from its official website www.clawpack.org. It can also be downloaded from github: github.com/clawpack.

The new book based on jupyter notebooks by David Ketcheson et al. offers a convenient way of learning clawpack and the theoretical fundamentals through a series of examples. Many of these examples will be used here. The html version of this book is available online.

Jupyter notebook viewers: cocalc.com and nbviewer.jupyter.org.

Jupyter can be installed under different operating systems (see jupyter.org/install). I personally use the Anaconda distribution or code visual studio and their user graphics interface. Registred EPFL students can use jupyter notebooks online by accessing the noto platform.

I place my jupyter notebooks in my github page: github.com/cancey.

Additional ressources

- Shaltop (Peruzzetto *et al.*, 2021)
- Basilisk
- Iber (Bladé *et al.*, 2014; Cea & Bladé, 2015)

Notation

The notation used in this tutorial differs from that used by Randall LeVeque. I use the classic tensorial notation: vectors and tensors are denoted by boldface symbols. I also use the operator $\cdot$ to refer to the contracted product ("produit une fois contracté" in French) and $:$ for the double-contracted product.

With this notation, we introduce the following operations between tensors $\boldsymbol{A}$ and $\boldsymbol{B}$ whose matrix representation is A_{ij} (i row, j column index) and B_{ij}, respectively, $\boldsymbol{v}$ and $\boldsymbol{w}$ two vectors of coordinates v_i and w_i in a given basis:

$$\begin{aligned}
\boldsymbol{A}\cdot\boldsymbol{B} &= \sum_j A_{ij}B_{jk}\\
\boldsymbol{A}:\boldsymbol{B} &= \sum_{i,j} A_{ij}B_{ji}\\
\boldsymbol{A}\cdot\boldsymbol{v} &= \sum_j A_{ij}v_j\\
\boldsymbol{w}\cdot\boldsymbol{v} &= \sum_i w_i v_i
\end{aligned}$$

Sometimes indices are in superscript when there is no possible confusion with exponents of a power function and the subscript position is already in use.

CHAPTER 1 Hyperbolic equations

Let us start with linear hyperbolic systems. Nonlinear equations are more complex, but the solutions to the Riemann problem have a similar structure to that exhibited by linear systems. Furthermore, finite-volume numerical solvers involve approximate (linearised) solutions to this Riemann problem.

1.1 Riemann problems for linear hyperbolic equations

1.1.1 *Linear system*

For one-dimensional problems, a linear hyperbolic equation is defined by an equation of the form

$$\frac{\partial}{\partial t}\boldsymbol{q} + \boldsymbol{A}\cdot\frac{\partial}{\partial x}\boldsymbol{q} = \boldsymbol{S}, \tag{1.1}$$

where $\boldsymbol{q}$ is a vector with m components representing the unknowns q_i, $\boldsymbol{A}$ is an $m \times m$ matrix whose eigenvalues λ_i are assumed to be real and distinct, and $\boldsymbol{S}$ is a vector (of dimension m) called the *source term*, x is the spatial coordinate, and t is time. For the moment, we assume that $\boldsymbol{S} = 0$ (the equation is said to be *homogenous*). The matrix $\boldsymbol{A}$ has m real eigenvalues λ_i, which are associated with m left $\boldsymbol{v}_i$ and m right eigenvectors $\boldsymbol{w}_i$:

$$\boldsymbol{A}\cdot\boldsymbol{w}_i = \lambda_i\boldsymbol{w}_i \text{ and } \boldsymbol{v}_i\cdot\boldsymbol{A} = \lambda_i\boldsymbol{v}_i. \tag{1.2}$$

In the following, the eigenvalues are ranked in ascending order: $\lambda_1 < \lambda_2 \cdots < \lambda_m$.

1.1.2 *Diagonalisation*

Equation (1.1) is a system of coupled partial differential equations. It is easier to solve uncoupled equations than coupled equations, and thus our first task is to see how we can uncouple Eq. (1.1).

If we multiply Eq. (1.1) by $\boldsymbol{v}_i$, we obtain:

$$\boldsymbol{v}_i\cdot\frac{\partial}{\partial t}\boldsymbol{q} + \boldsymbol{v}_i\cdot\boldsymbol{A}\cdot\frac{\partial}{\partial x}\boldsymbol{q} = \boldsymbol{v}_i\cdot\boldsymbol{S}. \tag{1.3}$$

We introduce the *characteristic variable* or *Riemann variable* s_i ($1 \le i \le m$):

$$s_i = \boldsymbol{v}_i\cdot\boldsymbol{q} \text{ and the vector } \boldsymbol{s} = (s_1, \cdots, s_m), \tag{1.4}$$

and the diagonal matrix $\boldsymbol{\Lambda} = \operatorname{diag}(\lambda_1, \cdots \lambda_m)$. With this notation, we transform Eq. (1.3) into a system of m uncoupled equations:

$$\frac{\partial}{\partial t}\boldsymbol{s} + \boldsymbol{\Lambda}\cdot\frac{\partial}{\partial x}\boldsymbol{s} = \boldsymbol{L}\cdot\boldsymbol{S}, \tag{1.5}$$

where $\boldsymbol{L}$ is a matrix whose rows are made of the left eigenvectors: $\boldsymbol{L} = [\boldsymbol{v}_1, \cdots, \boldsymbol{v}_m]^T$.

1.1.3 *Mathematical complement*

Similarly to what we did with $\boldsymbol{L}$, we define the matrix $\boldsymbol{R}$ whose columns are made of the right eigenvectors: $\boldsymbol{R} = [\boldsymbol{w}_1, \cdots, \boldsymbol{w}_m]$. The following relationships hold true

$$\begin{aligned} \boldsymbol{A} \cdot \boldsymbol{R} &= \boldsymbol{R} \cdot \boldsymbol{\Lambda}, && (1.6) \\ \boldsymbol{L} \cdot \boldsymbol{A} &= \boldsymbol{\Lambda} \cdot \boldsymbol{L}. && (1.7) \end{aligned}$$

$\boldsymbol{R}$ is sometimes called the *modal matrix*, while $\boldsymbol{\Lambda}$ is called the *spectral matrix*. We also have:

$$\begin{aligned} \boldsymbol{A} &= \boldsymbol{R} \cdot \boldsymbol{\Lambda} \cdot \boldsymbol{R}^{-1}, && (1.8) \\ \boldsymbol{A} &= \boldsymbol{L}^{-1} \cdot \boldsymbol{\Lambda} \cdot \boldsymbol{L}. && (1.9) \end{aligned}$$

Because when taking the transpose of $\boldsymbol{v}_i \cdot \boldsymbol{A} = \lambda_i \boldsymbol{v}_i$ we have

$$(\boldsymbol{v}_i \cdot \boldsymbol{A})^T = \boldsymbol{A}^T \cdot \boldsymbol{v}_i^T = \lambda_i \boldsymbol{v}_i^T, \tag{1.10}$$

the left eigenvectors $\boldsymbol{v}_i$ of $\boldsymbol{A}$ is also the right eigenvector of $\boldsymbol{A}^T$.

Multiplying Eq. (1.6) by $\boldsymbol{L}$ and Eq. (1.7) by $\boldsymbol{R}$, we get

$$\boldsymbol{L} \cdot \boldsymbol{A} \cdot \boldsymbol{R} = \boldsymbol{L} \cdot \boldsymbol{R} \cdot \boldsymbol{\Lambda} = \boldsymbol{\Lambda} \cdot \boldsymbol{L} \cdot \boldsymbol{R}. \tag{1.11}$$

When two matrices $\boldsymbol{M}$ and $\boldsymbol{D}$ (where $\boldsymbol{D}$ is diagonal) satisfy $\boldsymbol{D} \cdot \boldsymbol{M} = \boldsymbol{M} \cdot \boldsymbol{D}$, then $\boldsymbol{M}$ is diagonal. This means here that $\boldsymbol{M} = \boldsymbol{L} \cdot \boldsymbol{R}$ is diagonal. There is no unique choice for $\boldsymbol{R}$ and $\boldsymbol{L}$ as any multiple of an eigenvector is also an eigenvector. As a consequence, the product $\boldsymbol{L} \cdot \boldsymbol{R}$ is a diagonal matrix whose entries can take any value.

We can always define the right eigenvectors such that:

$$\boldsymbol{R} = \boldsymbol{L}^{-1}. \tag{1.12}$$

A geometrical interpretation of $\boldsymbol{R}$ and $\boldsymbol{L}$ is the following: as we have $\boldsymbol{R} \cdot \boldsymbol{L} = \boldsymbol{L}^{-1} \cdot \boldsymbol{L} = \boldsymbol{1}$ (where $\boldsymbol{1}$ denotes the identity matrix), then the left and right eigenvectors are orthogonal two by two: $\boldsymbol{v}_i \cdot \boldsymbol{w}_i \neq 0$ and $\boldsymbol{v}_i \cdot \boldsymbol{w}_k = 0$ for $k \neq i$.

In practice, we determine the right eigenvectors $\boldsymbol{w}_i$. The left eigenvectors are the right eigenvectors of the transpose of $\boldsymbol{A}$. The resulting matrices $\boldsymbol{R}$ and $\boldsymbol{L}$ satisfy: $\boldsymbol{R} \cdot \boldsymbol{L}^T = \mathrm{diag}(\boldsymbol{w}_k \cdot \boldsymbol{v}_k)_{1 \le k \le m}$. Furthermore, by normalising the right eigenvectors ($\tilde{\boldsymbol{w}}_i = \boldsymbol{w}_i / |\boldsymbol{w}_i|$), we can enforce $\boldsymbol{L} = \boldsymbol{R}^{-1}$, a relationship that turns out to be helpful thereafter.

Example Let us consider the 3×3 matrix

$$\boldsymbol{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 2 & 8 & 2 \end{pmatrix}.$$

The eigenvalues are $\lambda_1 = -3$, $\lambda_2 = -1$, $\lambda_3 = 12$ associated with the right eigenvectors

$$\boldsymbol{w}_1 = \begin{pmatrix} -3 \\ 0 \\ 2 \end{pmatrix}, \ \boldsymbol{w}_2 = \begin{pmatrix} -1 \\ -1 \\ 2 \end{pmatrix}, \ \boldsymbol{w}_3 = \begin{pmatrix} 11 \\ 26 \\ 23 \end{pmatrix},$$

and the left eigenvectors are determined by seeking the right eigenvectors of the transpose of $\boldsymbol{A}$

$$\boldsymbol{v}_1 = \begin{pmatrix} 5 \\ -3 \\ 1 \end{pmatrix}, \ \boldsymbol{v}_2 = \begin{pmatrix} 4 \\ -7 \\ 6 \end{pmatrix}, \ \boldsymbol{v}_3 = \begin{pmatrix} 2 \\ 4 \\ 3 \end{pmatrix}.$$

It can be checked that $\boldsymbol{A}\cdot\boldsymbol{R}-\boldsymbol{R}\cdot\boldsymbol{\Lambda}$ is the null matrix. Note that the order with which the product is made is important. For instance, $\boldsymbol{A}\cdot\boldsymbol{R}-\boldsymbol{\Lambda}\cdot\boldsymbol{R}$ provides a matrix whose diagonal entries are zero, but the off-diagonal entries are nonzero

$$\boldsymbol{A}\cdot\boldsymbol{R}-\boldsymbol{\Lambda}\cdot\boldsymbol{R}=\begin{pmatrix} 0 & 2 & 143 \\ 0 & 0 & 390 \\ -26 & -30 & 0 \end{pmatrix}$$

and similarly $\boldsymbol{L}\cdot\boldsymbol{A}-\boldsymbol{\Lambda}\cdot\boldsymbol{L}$ provides the null matrix.

We can also check that $\boldsymbol{L}\cdot\boldsymbol{R}$ is a diagonal matrix:

$$\boldsymbol{L}\cdot\boldsymbol{R}=\begin{pmatrix} -13 & 0 & 0 \\ 0 & 15 & 0 \\ 0 & 0 & 195 \end{pmatrix}.$$

```python
#import the numpy libray and its linear algebra sublibrary
import numpy as np
from numpy import linalg as la
eps=1e-8
```

```python
#examples
A=np.array([[1,2,3],[4,5,6],[2,8,2]])
```

```python
#computing the right and left eigenvalues and vectors
# @ refers to the simple contracted product and .T provides the transpose
# R: matrix whose columns are right eigenvectors.
# L: matrix whose rows are left eigenvectors.
# LAM: diagonal matrix whose entries are the eigenvalues
lamb1, R = la.eig(A)
lamb2, L = la.eig(A.T)
L=L.T
print("R = \n",R)
print("L = \n",L)
LAM = np.diag(lamb1)
print("[lambda] = \n",LAM)
```

```
R =
 [[-3.02079270e-01 -8.32050294e-01 -4.08248290e-01]
 [-7.14005547e-01  2.07143878e-16 -4.08248290e-01]
 [-6.31620292e-01  5.54700196e-01  8.16496581e-01]]
L =
 [[-0.37139068 -0.74278135 -0.55708601]
 [-0.84515425  0.50709255 -0.16903085]
 [ 0.39801488 -0.69652603  0.59702231]]
[lambda] =
 [[12.  0.  0.]
 [ 0. -1.  0.]
 [ 0.  0. -3.]]
```

```python
#check that w is made by the right eigenvectors. We compute C = A.R-R.LAM.
   We replace small values by 0.
# The diagonal of C provides A.w_i-lambda_i w_i
# We check that this diagonal is zero
C=A@R-R@LAM
C[abs(C)<eps]=0
print(C)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

```
#We do the same with thee left eigenvectors and check that C=L.A- LAM.L is
   zero.
C=L@A -LAM@L
C[abs(C)<eps]=0
print(C)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

```
#Check that the product L.R gives a diagonal matrix
lr=L@R
lr[lr<eps]=0
print(lr)
```

```
[[0.99440626 0.         0.        ]
 [0.         0.6094494  0.        ]
 [0.         0.         0.60933335]]
```

```
#left and right eigenvectors are perpendicular when not associated with the
    same eigenvalue
w1 = R[:,0] # First column is the first right eigenvector
w2 = R[:,1]
w3 = R[:,2]

v1 = L[0,:] # First row is the first left eigenvector
v2 = L[1,:]
v3 = L[2,:]

# v_1.A-lambda_1v_1 = 0 ?
result=v1@A-lamb[0]*v1
result[result<eps]=0

print("v_1.A-lambda_1*v_1 = ",result)
# v_1 and w_1 normal?
vw11=v1@w1;
vw12=v1@w3;
print("v_1.w_1 = ",vw11)
print("v_1.w_2 = ",vw12)
```

```
v_1.A-lambda_1*v_1 =  [0. 0. 0.]
v_1.w_1 =  0.9944062617398504
v_1.w_2 =  0.0
```

```
[[0.99440626 0.         0.        ]
 [0.         0.6094494  0.        ]
 [0.         0.         0.60933335]]
```

1.1.4 Characteristic form and solution to the Cauchy problem for homogenous equations

When we have seen that when we want to solve a hyperbolic problem in the form (1.1), the strategy is to uncouple the original equations by making a change of variable $\boldsymbol{q} \rightarrow \boldsymbol{s} = \boldsymbol{L} \cdot \boldsymbol{q}$. The original equation (1.1) becomes

$$\frac{\partial}{\partial t}\boldsymbol{s} + \boldsymbol{\Lambda}\cdot\frac{\partial}{\partial x}\boldsymbol{s} = \boldsymbol{L}\cdot\boldsymbol{S}. \tag{1.13}$$

Each uncoupled equation of the system (1.13) can be put into a characteristic form

$$\frac{\partial s_i}{\partial t} + \lambda_i\frac{\partial s_i}{\partial x} = \boldsymbol{v}_i^T\cdot\boldsymbol{S} \Leftrightarrow \frac{\mathrm{d}s_i}{\mathrm{d}t} = \boldsymbol{v}_i\cdot\boldsymbol{S} \text{ along the straight line } \frac{\mathrm{d}x}{\mathrm{d}t} = \lambda_i. \tag{1.14}$$

For a homogenous problem, this means that s_i is constant along the line $x = \lambda_i t + x_0$. If we know the initial value $\boldsymbol{q}_0 = \boldsymbol{q}(x, t = 0)$, then we can deduce the initial condition for $\boldsymbol{s}$: $\boldsymbol{s}_0 = \boldsymbol{s}(x, t = 0) = \boldsymbol{L}\cdot\boldsymbol{q}_0$. For a homogenous equation, the solution to Eq. (1.14) is

$$s_i(x,t) = s_{i,0}(x-\lambda_i t), \tag{1.15}$$

and thus the solution to the initial-value (Cauchy) problem is

$$\begin{aligned}
\boldsymbol{q}(x,t) &= \boldsymbol{L}^{-1}\cdot\boldsymbol{s} = \boldsymbol{R}\cdot\boldsymbol{s} && (1.16)\\
&= \sum_{i=1}^{m} s_i(x,t)\boldsymbol{w}_i, && (1.17)\\
&= \sum_{i=1}^{m} s_{0,i}(x-\lambda_i t)\boldsymbol{w}_i, && (1.18)\\
&= \sum_{i=1}^{m} (\boldsymbol{v}_i\,\boldsymbol{\cdot q_0}\,(x-\lambda_i t))\boldsymbol{w}_i. && (1.19)
\end{aligned}$$

The solution $\boldsymbol{q}$ is a combination of the right eigenvectors. In other words, the initial conditions propagate along the directions $\boldsymbol{w}_i$.

This propagation is a consequence of the travelling-wave structure. Indeed, the linear hyperbolic system (1.1) is invariant to the travelling wave group. If we seek a solution in the form $\boldsymbol{s}(x,t) = \boldsymbol{s}(\xi)$ where $\xi = x - at$ and a is the wave velocity, then Eq. (1.1) leads to:

$$-a\frac{\mathrm{d}}{\mathrm{d}\xi}\boldsymbol{s} + \boldsymbol{A}\cdot\frac{\mathrm{d}}{\mathrm{d}\xi}\boldsymbol{s} = 0. \tag{1.20}$$

This shows that $\boldsymbol{s}'$ is an eigenvector of $\boldsymbol{A}$ and a must be one of the eigenvalues, say λ_i. Substituting the Cauchy solution Eq. (1.16) into Eq. (1.20) shows that this condition is met. For strictly hyperbolic systems (i.e., when all eigenvalues are real and distinct), the right eigenvectors form a basis, and the decomposition (1.16) is unique.

The solution to the Cauchy problem is the superposition of m waves, each is advected independently at the velocity λ_i along the direction $\boldsymbol{w}_i$, with no change in shape when the system is homogenous.

1.1.5 Simple wave

When the initial conditions are constant for all but one value k

$$s_{i,0}(x) = s_i \text{ for } i \neq k \text{ and } s_{k,0}(x) = s_{k,0}(x-\lambda_k t), \tag{1.21}$$

then the solution

$$\boldsymbol{q}(x,t) = \boldsymbol{q}_0(x-\lambda_k t) = s_{k,0}(x-\lambda_k t)\boldsymbol{w}_k + \sum_{i\neq k} s_i\boldsymbol{w}_i \tag{1.22}$$

is called a *simple wave*. Propagation concerns the direction k alone.

1.1.6 Riemann problem: definition

A Riemann problem is an initial-value problem for which the initial value is piecewise constant with a single jump discontinuity at some point, by default at $x = 0$:

$$\boldsymbol{q} = \left\{ \begin{array}{l} \boldsymbol{q}_l \text{ if } x < 0, \\ \boldsymbol{q}_r \text{ if } x > 0, \end{array} \right. \tag{1.23}$$

where $\boldsymbol{q}_l$ and $\boldsymbol{q}_r$ are called the left and right states, respectively. Because the right eigenvectors form a basis, we can decompose $\boldsymbol{q}_l$ and $\boldsymbol{q}_r$ in this basis

$$\boldsymbol{q}_l = \sum_{i=1}^{m} \alpha_i^l \boldsymbol{w}_i \text{ and } \boldsymbol{q}_r = \sum_{i=1}^{m} \alpha_i^r \boldsymbol{w}_i, \tag{1.24}$$

where α_i^l and α_i^r are two constants. Each Riemann variable $s_i = \boldsymbol{v} \cdot \boldsymbol{q}$ satisfies the initial condition

$$s_{i,0} = \left\{ \begin{array}{l} s_i^l = \boldsymbol{v}_i \cdot \boldsymbol{q}_l = \alpha_i^l \text{ if } x < 0, \\ s_i^r = \boldsymbol{v}_i \cdot \boldsymbol{q}_r = \alpha_r^l \text{ if } x > 0. \end{array} \right. \tag{1.25}$$

The solution to the individual Riemann problem shows that the initial discontinuity propagates with speed λ_i along the direction $\boldsymbol{w}_i$:

$$s_i(x,t) = \left\{ \begin{array}{l} \alpha_i^l \text{ if } x < \lambda_i t, \\ \alpha_i^r \text{ if } x > \lambda_i t. \end{array} \right. \tag{1.26}$$

Using Eq. (1.16), we deduce that the solution to the Riemann problem is

$$\boldsymbol{q}(x,t) = \boldsymbol{R} \cdot \boldsymbol{s} = \sum_{i=1}^{m} s_i(x,t) \boldsymbol{w}_i, \tag{1.27}$$

We can interpret this solution in the following way. Let us consider a point P at (x,t). We refer to I as the maximum index i for which $x > \lambda_i t$. As illustrated by the example of Fig. 1.1, we can decompose the solution into two parts, either reflecting the left or right initial conditions

$$\boldsymbol{q} = \sum_{i=1}^{I} \alpha_i^r \boldsymbol{w}_i + \sum_{i=I+1}^{m} \alpha_i^l \boldsymbol{w}_i. \tag{1.28}$$

When crossing the ith characteristic, there is a jump from α_i^l to α_i^r while the other coefficients remain constant. As illustrated in Fig. 1.2(a), the plane is split into different wedges separated by characteristic lines of slope λ_i. Across the ith characteristic, the solution $\boldsymbol{q}$ experiences a jump:

$$\Delta \boldsymbol{q} = (\alpha_i^r - \alpha_i^l) \boldsymbol{w}_i, \tag{1.29}$$

which can be written as

$$\Delta \boldsymbol{q} = \alpha_i \boldsymbol{w}_i \text{ with } \alpha_i = \alpha_i^r - \alpha_i^l. \tag{1.30}$$

For linear hyperbolic systems, a strategy of solving the Riemann problem is to decompose the initial jump $\Delta \boldsymbol{q} = \boldsymbol{q}_r - \boldsymbol{q}_l$ in the right eigenvector basis

$$\Delta \boldsymbol{q} = \sum_{i=1}^{m} \alpha_i \boldsymbol{w}_i, \tag{1.31}$$

which requires determining the coefficient α_i

$$\Delta \boldsymbol{q} = \boldsymbol{R} \cdot \boldsymbol{\alpha} \Rightarrow \boldsymbol{\alpha} = \boldsymbol{R}^{-1} \cdot \Delta \boldsymbol{q} = \boldsymbol{L} \cdot \Delta \boldsymbol{q}. \tag{1.32}$$

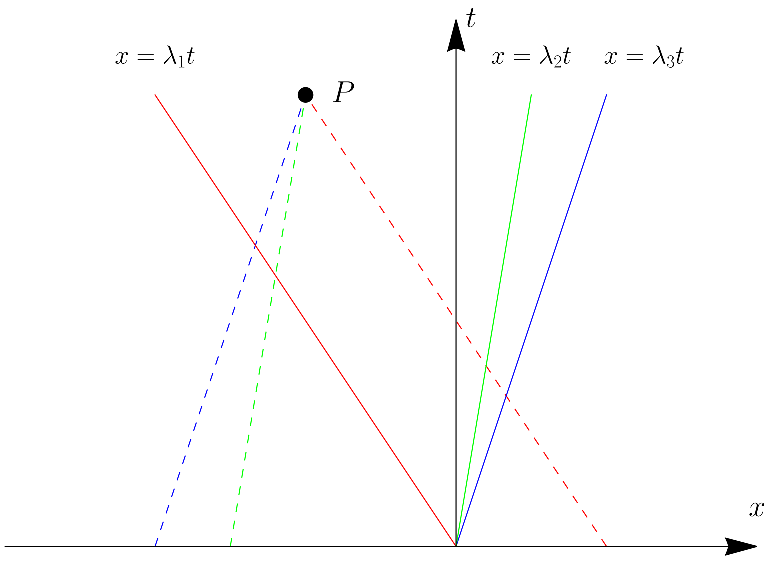


Figure 1.1 Characteristic lines emanating from the origin point (solid lines) and joining P (dashed lines). In this figure, we have $I = 1$: P is on the right of the first characteristic curve $x = \lambda_1 t$ (and thus the P coordinates satisfy $x > \lambda_1 t$), and on the left of the two others. Here we have $\boldsymbol{q} = \alpha_1^r \boldsymbol{w}_1 + \alpha_2^l \boldsymbol{w}_2 + \alpha_3^l \boldsymbol{w}_3$.

As this decomposition is central to Clawpack, we give further information about this decomposition. Let us introduce the wave

$$\boldsymbol{W}_i = \alpha_i \boldsymbol{w}_i. \tag{1.33}$$

The solution to the Riemann problem can thus be written

$$\boldsymbol{q} = \sum_{i=1}^{m} s_i \boldsymbol{w}_i \text{ with } s_i = \boldsymbol{v}_i \cdot \boldsymbol{q}_0, \tag{1.34}$$

$$\boldsymbol{q} = \boldsymbol{q}_l + \sum_{i=1}^{I} \boldsymbol{W}_i, \tag{1.35}$$

$$\boldsymbol{q} = \boldsymbol{q}_r - \sum_{i=I+1}^{m} \boldsymbol{W}_i, \tag{1.36}$$

$$\boldsymbol{q} = \boldsymbol{q}_l + \sum_{i=1}^{m} H(x - \lambda_i t) \boldsymbol{W}_i, \tag{1.37}$$

where H is the Heaviside function. Equation (1.35) can also be written

$$\boldsymbol{q} = \boldsymbol{q}_l + \sum_{\lambda_i < x/t} \boldsymbol{W}_i, \tag{1.38}$$

which can be interpreted as follows (see the example in Fig. 1.1): at time t and position x, the state $\boldsymbol{q}$ is the left initial state to which contributions from the right initial state are added if this point is on the right of the characteristic $x = \lambda_i t$ (that is, when $x > \lambda_i t$).

1.1.7 Phase plane representation for $m = 2$ equations

For a linear system of two hyperbolic equations, the solution consists of two discontinuities $x = \lambda_1 t$ and $x = \lambda_2 t$, and within the wedge formed by these two discontinuities, there is an intermediate (constant) state

$$\boldsymbol{q}_* = \alpha_1^r \boldsymbol{w}_1 + \alpha_2^l \boldsymbol{w}_2. \tag{1.39}$$

As shown by Fig. 1.2, the intermediate state can be expressed in different ways:

$$\begin{aligned}\boldsymbol{q}_* =& \alpha_1^r \boldsymbol{w}_1 + \alpha_2^l \boldsymbol{w}_2,\\ =& \boldsymbol{q}_l + (\alpha_1^r - \alpha_1^l)\boldsymbol{w}_1 = \boldsymbol{q}_l + \boldsymbol{W}_1,\\ =& \boldsymbol{q}_r - (\alpha_2^r - \alpha_2^l)\boldsymbol{w}_2 = \boldsymbol{q}_r - \boldsymbol{W}_2.\end{aligned}$$

The intermediate $\boldsymbol{q}_*$ is related to the left state $\boldsymbol{q}_l$ by the shock wave $\boldsymbol{W}_1 = (\alpha_1^r - \alpha_1^l)\boldsymbol{w}_1$, and to the right state $\boldsymbol{q}_r$ by the shock wave $-\boldsymbol{W}_2 = -(\alpha_2^r - \alpha_2^l)\boldsymbol{w}_2$.

We can also plot the solution in the phase plane (q_1, q_2): starting from the left state $\boldsymbol{q}_l$, we follow the direction $\boldsymbol{w}_1$ to reach the intermediate state $\boldsymbol{q}_*$, and finally the direction $\boldsymbol{w}_2$ to reach the right state $\boldsymbol{q}_r$, shown by Fig. 1.2(b). As information is propagated along the characteristic curves at the velocities $\lambda_1 < \lambda_2$, the intermediate state is necessarily connected to the left state by the 1-shock wave (velocity λ_1 and direction $\boldsymbol{w}_1$), and it is connected to the right state by the 2-shock wave (velocity λ_2 and direction $\boldsymbol{w}_2$).

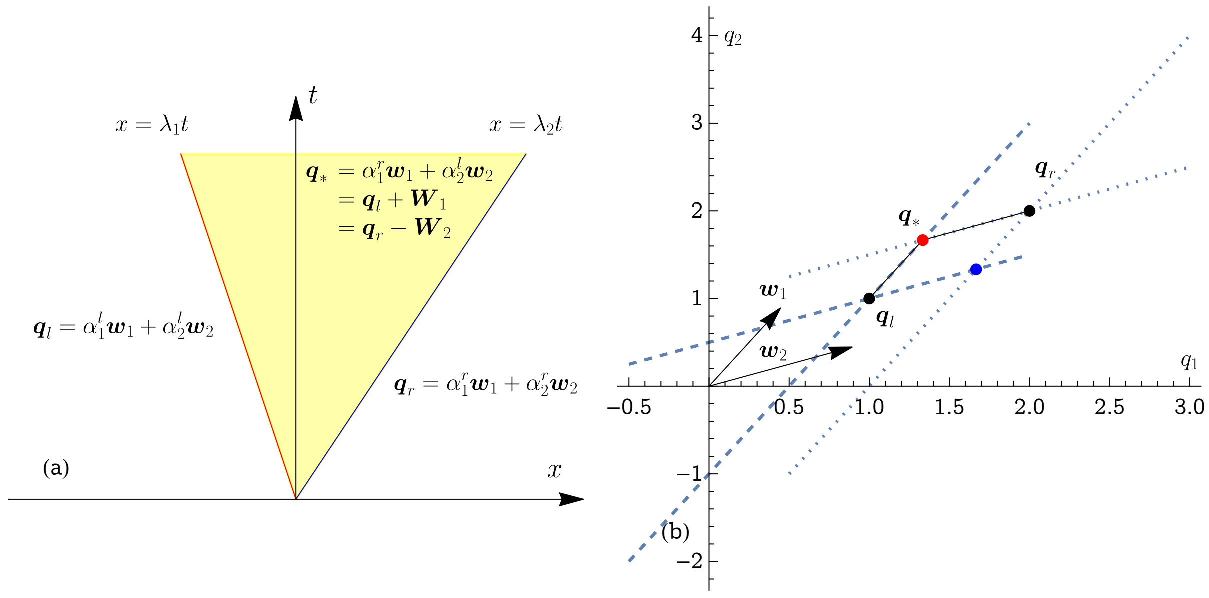

$x = \lambda_1 t$
t
$x = \lambda_2 t$
$\boldsymbol{q}_* = \alpha_1^r \boldsymbol{w}_1 + \alpha_2^l \boldsymbol{w}_2$
$= \boldsymbol{q}_l + \boldsymbol{W}_1$
$= \boldsymbol{q}_r - \boldsymbol{W}_2$
$\boldsymbol{q}_l = \alpha_1^l \boldsymbol{w}_1 + \alpha_2^l \boldsymbol{w}_2$
$\boldsymbol{q}_r = \alpha_1^r \boldsymbol{w}_1 + \alpha_2^r \boldsymbol{w}_2$
(a)
x
q_2
$\boldsymbol{q}_r$
$\boldsymbol{q}_*$
$\boldsymbol{w}_1$
$\boldsymbol{q}_l$
$\boldsymbol{w}_2$
q_1
(b)
4
3
2
1
−1
−2
−0.5
0.5
1.0
1.5
2.0
2.5
3.0

Figure 1.2 (a) General solution to a two-dimensional Riemann problem in the (x, t) plane. The intermediate state $\boldsymbol{q}_*$ is separated from the left and right states by the shocks $\boldsymbol{W}_i = \alpha_i \boldsymbol{w}_i$ with $i = 1$ or 2. (b) Phase plane representation.

1.2 Nonlinear scalar problem

Let us consider the following nonlinear hyperbolic equation called the *nonlinear advection equation*, which can be cast in two different forms called the conservative (left equation) and non-conservative (right equation) forms:

$$\frac{\partial}{\partial t}q + \frac{\partial}{\partial x}f(q) = S(q, x, t) \Leftrightarrow \frac{\partial}{\partial t}q + c(q)\frac{\partial}{\partial x}q = S(q, x, t), \tag{1.40}$$

where f is the flux function (a function of q, and possibly of x and t), q is the unknown, S is the source term, and $c = f'(q)$ is the celerity. We assume that the celerity is an increasing function of q, which implies that the flux function is convex ($f'' > 0$). Nonconvex functions are possible, but they lead to difficulties that we will not address here. The equation is said to be *homogeneous* when the source term s is zero

$$\frac{\partial}{\partial t}q + \frac{\partial}{\partial x}f(q) = 0. \tag{1.41}$$

1.2.1 Characteristic form

Equation (1.40) can be put into the characteristic form

$$\frac{\mathrm{d}}{\mathrm{d}t}q = S(q,\, x,\, t) \text{ along the curve } \frac{\mathrm{d}x}{\mathrm{d}t} = c(q). \tag{1.42}$$

When the source term is zero ($S = 0$), then q is constant along the characteristic curve, which is therefore a straight line of slope c, whose value is fixed by the initial or boundary condition.

1.2.2 Rankine–Hugoniot equation

As the celerity $c(q)$ is function of q, the characteristic curves are not parallel like in the linear case, and may intersect. As multivalued functions are not possible (this would otherwise break the assumptions of smoothness and uniqueness of the solution), then a shock takes place and connects two continuous branches of the solution. By taking a control volume around the shock position $x = \sigma(t)$, we can deduce that its velocity $\dot\sigma$ is given by the Rankine–Hugoniot equation:

$$\dot\sigma = \frac{[\![f(q)]\!]}{[\![q]\!]}, \tag{1.43}$$

where the double brackets denote the flux jump across the shock wave

$$[\![f(q)]\!] = \lim_{x\to\sigma,\, x>\sigma} f(q) - \lim_{x\to\sigma,\, x<\sigma} f(q).$$

Proof. Let us integrate Eq. (1.40) over the interval $[x_l, x_r]$ with a discontinuity at $x = \sigma(t)$ (with $x_l < \sigma < x_r$):

$$\begin{aligned}\int_{x_l}^{x_r} \frac{\partial}{\partial t}q(x,t)\mathrm{d}x &= -\int_{x_l}^{x_r} \frac{\partial}{\partial x}f[q(x,\, t)]\mathrm{d}x\\ &= f(q(x_l,\, t)) - f(q(x_r,\, t)).\end{aligned}$$

We would like to swap spatial integration and time differentiation while paying attention to the discontinuity at $x = \sigma(t)$. By breaking down the interval $[x_l,\, x_r]$ into two parts, we can write

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{x_l}^{x_r} q(x,\, t)\mathrm{d}x = \frac{\mathrm{d}}{\mathrm{d}t}\left(\int_{x_l}^{\sigma} q(x,\, t)\mathrm{d}x + \int_{\sigma}^{x_r} q(x,\, t)\mathrm{d}x\right),$$

and by differentiating with respect to t and using Leibniz rule, we deduce:

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_{x_l}^{x_r} q(x,\, t)\mathrm{d}x = \int_{x_l}^{\sigma} \frac{\partial}{\partial t}q(x,\, t)\mathrm{d}x + \int_{\sigma}^{x_r} \frac{\partial}{\partial t}q(x,\, t)\mathrm{d}x + \dot\sigma q(x_r,t) - \dot\sigma q(x_l,t).$$

The integral version of Eq. (1.40) is thus

$$-\frac{\mathrm{d}}{\mathrm{d}t}\int_{x_l}^{x_r} q(x,\, t)\mathrm{d}x + \dot\sigma q(x_r,t) - \dot\sigma q(x_l,t) = f(q(x_r,\, t)) - f(q(x_l,\, t)).$$

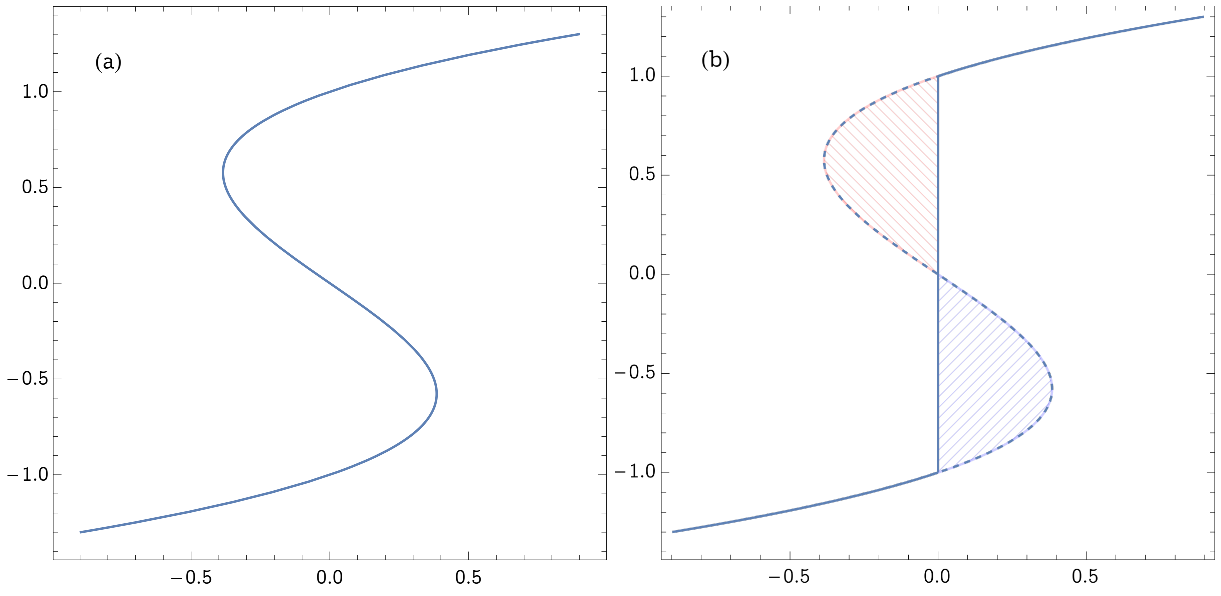


Figure 1.3 (a) multivalued function. (b) The multivalued part is replaced by a discontinuity. The areas of the two lobes are identical.

Taking the limit $x_r \to \sigma$ and $x_l \to \sigma$, we eventually find the desired expression:

$$\dot{\sigma}[\![q]\!] = [\![f(q)]\!].$$

□

One problem with the Rankine–Hugoniot equation (1.43) is that it can provide solutions that are mathematically correct, but physically unrealistic because they violate the principle of energy dissipation (shocks dissipate energy) (Smoller, 1982; Holden & Risebro, 2015). One method for determining whether a solution is physically admissible is to consider a regularised version of the governing equation (1.41)

$$\frac{\partial}{\partial t}q + \frac{\partial}{\partial x}f(q) = \mu\frac{\partial^2}{\partial x^2}q, \tag{1.44}$$

where μ is a constant viscosity ($\mu > 0$). We would like to solve the regularised equation (1.40) in the limit of small viscosities $\mu \to 0$. We are seeking a travelling-wave solution in the form

$$q(x,t) = U(\xi) \text{ with } \xi = \frac{x-at}{\mu} \tag{1.45}$$

where $a > 0$ is the wave velocity, and U satisfies the boundary conditions

$$\begin{cases} U \to q_l \text{ when } \xi \to -\infty, \\ U \to q_r \text{ when } \xi \to +\infty. \end{cases} \tag{1.46}$$

Substituting the travelling wave (1.45) into Eq. (1.44) and making the change of variable

$$\frac{\partial\cdot}{\partial t} = \frac{\partial\cdot}{\partial \xi}\frac{\partial \xi}{\partial t} = -\frac{a}{\mu}\frac{\mathrm{d}\cdot}{\mathrm{d}\xi} \text{ and } \frac{\partial\cdot}{\partial x} = \frac{\partial\cdot}{\partial \xi}\frac{\partial \xi}{\partial x} = \frac{1}{\mu}\frac{\mathrm{d}\cdot}{\mathrm{d}\xi},$$

we obtain the following equation

$$\ddot{U}(\xi) = -a\dot{U} + \frac{\mathrm{d}}{\mathrm{d}\xi}f(U),$$

whose integration provides

$$\dot{U}(\xi) = -aU + f(U) + b,$$

where b is a constant of integration. By looking at the boundary conditions (1.46), we find that this constant satisfies the following condition (1.46)

$$b = aq_r - f(q_r) = aq_l - f(q_l), \tag{1.47}$$

which is possible only if a satisfied

$$a = \frac{f(q_r) - f(q_l)}{q_r - q_l}. \tag{1.48}$$

It can be observed that the wave speed a coincides with the shock speed $\dot{\sigma}$ predicted by Eq. (1.43): $a = \dot{\sigma}$. We thus have to solve the equation

$$\dot{U} = V(U) = -\dot{\sigma}(U - q_l) + f(U) - f(q_l) = -\dot{\sigma}(U - q_r) + f(U) - f(q_r).$$

We note that q_l and q_r are equilibrium points of $V(U)$: $V(q_l) = V(q_r) = 0$. To determine the behaviour of the solution near equilibrium point, we linearise V. We assume that $q_l > q_r$ and we start with V near q_l:

$$\dot{U} = V(U) = -\dot{\sigma}(U - q_l) - f(q_l) + f(q_l) + (U - q_l)f'(q_l) + \cdots,$$

and introducing $\eta = U - q_l$, we have to first order in η

$$\frac{\mathrm{d}\eta}{\mathrm{d}\xi} = (f'(q_l) - \dot{\sigma})\eta,$$

whose solution is $U = q_l + u_0 \exp[(f'(q_l) - \dot{\sigma})\xi]$ where u_0 is a constant of integration. When $\xi \to -\infty$, the boundary condition (1.46) imposes that $f'(q_l) - \dot{\sigma} > 0$ and thus

$$f'(q_l) > \dot{\sigma}.$$

Reiterating the same procedure for the right state q_r, we obtain that $f'(q_r) < \dot{\sigma}$. We then obtain a relation that is called the *Lax entropy condition*

$$f'(q_r) < \dot{\sigma} < f'(q_l). \tag{1.49}$$

This condition is consistent with the flux convexity ($f'' > 0$ implies that f' is an increasing function). This would have not been the case if we had initially assumed that $q_r > q_l$.

1.2.3 Riemann problem

Let us consider the Riemann problem for a homogeneous hyperbolic equation:

$$\frac{\partial}{\partial t}q + \frac{\partial}{\partial x}f(q) = 0, \tag{1.50}$$

subject to the initial condition

$$q(x,\ 0) = q_0(x) = \begin{cases} q_l & \text{if } x < 0, \\ q_r & \text{if } x > 0, \end{cases}$$

where q_l and q_r are constant. When the flux function $f(q)$ is convex, two solutions are possible depending on these constants:

- rarefaction waves,
- shock waves.

Let us start with rarefaction waves. The governing equation (1.50) is invariant to the transformation $x \to \alpha x$ and $t \to \alpha t$. A solution can thus be sought in the form $q(\xi)$ with $\xi = x/t$. Substituting this form into Eq. (1.50):

$$\left(f'(q(\xi)) - \xi\right) q' = 0.$$

Apart from the trivial solution $q = 0$ (note that it does not satisfy the initial condition), the solution is

$$q(x,\, t) = f'^{(-1)}(\xi)$$

where $f'^{(-1)}$ is the inverse of f'. The whole solution is the piecewise continuous function:

$$q(x,\, t) = \begin{cases} q_l & \text{if } \dfrac{x}{t} \le f'(q_l), \\ f'^{(-1)}(\xi) & \text{if } f'(q_l) \le \dfrac{x}{t} \le f'(q_r) \\ q_r & \text{if } \dfrac{x}{t} \ge f'(q_r). \end{cases}$$

Let us now consider a shock wave. It position is $x = \sigma(t) = \dot{\sigma} t$. The Rankine–Hugoniot equation (1.43) gives $[\![f(q)]\!] = \dot{\sigma}[\![q]\!]$. The whole solution is:

$$q(x,\, t) = \begin{cases} q_l & \text{if } x < \dot{\sigma} t, \\ q_r & \text{if } x > \dot{\sigma} t. \end{cases}$$

The shock velocity $\dot{\sigma}$ is given by:

$$\dot{\sigma} = \frac{[\![f(q)]\!]}{[\![q]\!]} = \frac{f(q_l) - f(q_r)}{q_l - q_r}.$$

Let us summarise the two possible solutions (see Fig. 1.4): Recall that when $f'' > 0$, the celerity $c(q) = f'(q)$ is an increasing function of q, which is also the slope of the characteristic curves (straight lines):

- $q_r < q_l$. The two families of characteristic curves cross at any time $t > 0$. The shock wave moves at speed $c(q_r) < \dot{\sigma} < c(q_l)$. This condition is called the *Lax condition*, which defines whether a shock is physically admissible.
- $q_r > q_l$, $c(q_r) > c(q_l)$. At time $t = 0$, the two families of characteristic curves fan out. Equation $\xi = f'(U(\xi))$ is an implicit solution over the interval $c(q_r) > \xi > c(q_l)$.

1.3 Nonlinear systems

Let us now consider the nonlinear case for one-dimensional problems in its conservative and non-conservative forms

$$\frac{\partial}{\partial t}\boldsymbol{q} + \frac{\partial}{\partial x}\boldsymbol{f}(\boldsymbol{q}) = \boldsymbol{S} \Leftrightarrow \frac{\partial}{\partial t}\boldsymbol{q} + \boldsymbol{A}(\boldsymbol{q}) \cdot \frac{\partial}{\partial x}\boldsymbol{q} = \boldsymbol{S}, \tag{1.51}$$

where $\boldsymbol{q}$ is a vector with m components representing the unknowns, $\boldsymbol{f}$ is the flux function, $\boldsymbol{A} = \nabla \boldsymbol{f}$ is its Jacobian (the gradient involves the derivatives with respect to the $\boldsymbol{q}$ components). We assume that $\boldsymbol{A}$ is $m \times m$ matrix whose eigenvalues λ_i are assumed to be real and distinct over a certain domain—like for the linear case.

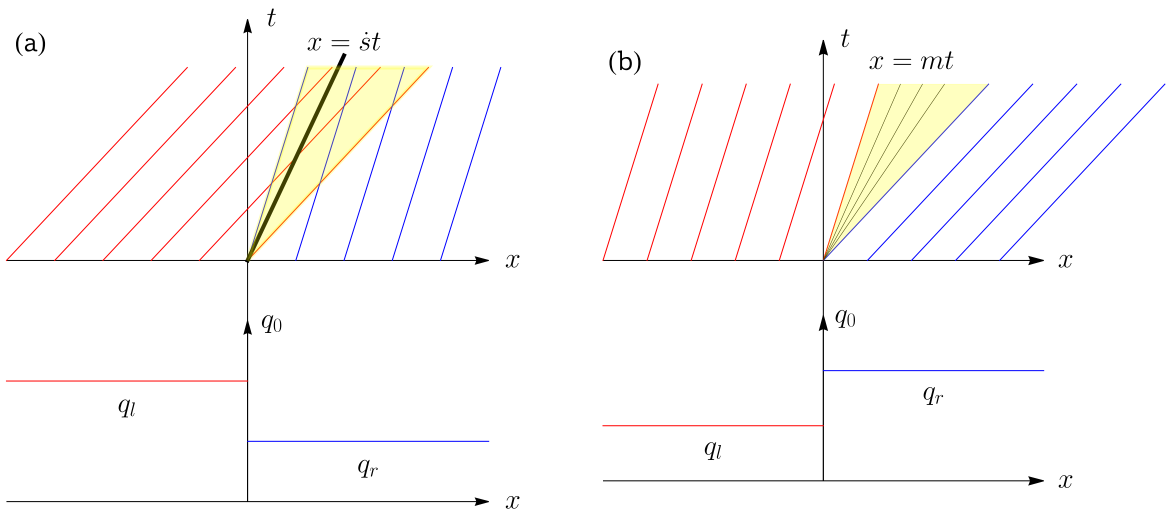


Figure 1.4 (a) When $q_r > q_l$, the solution is a shock wave $x = \dot{\sigma}t$ with $\dot{\sigma} = [\![f(q)]\!]/[\![q]\!]$ separating the two constant states q_l and q_r. The coloured area shows the region in which characteristic lines intersect. (b) When $q_r < q_l$, the solution is a rarefaction wave. The characteristic curves are the straight lines $x = mt$ with $c(q_r) > m > c(q_l)$. The coloured area shows the region in which the characteristic curves fan out.

1.3.1 Riemann variables

The computational strategy closely follows the one taken for the linear case. It relies on the concept of differential invariants. Let us illustrate this concept for $m = 2$. The unknown vector $\boldsymbol{q}$ has components $(q_1,\ q_2)$. We seek a new variable $\boldsymbol{s} = \{s_1, s_2\}$ such that:

$$\boldsymbol{v}_1 \cdot \mathrm{d}\boldsymbol{q} = \mu_1 \mathrm{d}s_1,$$

$$\boldsymbol{v}_2 \cdot \mathrm{d}\boldsymbol{q} = \mu_2 \mathrm{d}s_2,$$

where μ_i are the integrating factors such that $\mathrm{d}s_i$ are exact differentials. By expanding the differential $\mathrm{d}s_1$, we get:

$$\mu_1 \mathrm{d}s_1 = \mu_1 \left(\frac{\partial s_1}{\partial q_1}\mathrm{d}q_1 + \frac{\partial s_1}{\partial q_2}\mathrm{d}q_2 \right) = v_{11}\mathrm{d}q_1 + v_{12}\mathrm{d}q_2.$$

Upon identification with the former equation, we deduce:

$$\frac{\partial s_1}{\partial q_1} = \frac{v_{11}}{\mu_1},$$

and

$$\frac{\partial s_1}{\partial q_2} = \frac{v_{12}}{\mu_1}.$$

We deduce the governing equations for s_1 and μ_1. By dividing the two equations above, we obtain:

$$\frac{\partial s_1}{\partial q_1} = \frac{v_{11}}{v_{12}}\frac{\partial s_1}{\partial q_2}, \tag{1.52}$$

while the integrating factor is obtained by applying the Schwarz theorem

$$\frac{\partial}{\partial q_1}\frac{v_{12}}{\mu_1} = \frac{\partial}{\partial q_2}\frac{v_{11}}{\mu_1}.$$

We have seen above that the left and right eigenvectors are orthogonal two by two, which means here that $\boldsymbol{v}_1 \cdot \boldsymbol{w}_2 = 0$ (that is, $v_{12} = w_{21}$ and $-v_{11} = w_{22}$). We can then transform Eq. (1.52) into

$$w_{21}\frac{\partial s_1}{\partial q_1} + w_{22}\frac{\partial s_1}{\partial q_2} = 0 \Rightarrow \boldsymbol{w}_2 \cdot \nabla s_1 = 0. \tag{1.53}$$

Caveat. Later, we will introduce Riemann invariants r_k (see § 1.3.3). These invariants will be defined as $\boldsymbol{w}_k \cdot \nabla r_k = 0$. Although they are closely related to the Riemann variables s_k, they differ from each other: we have $s_1 = r_2$ and $s_2 = r_1$.

Equation (1.52) can be cast in the form

$$\frac{\mathrm{d}q_1}{v_{12}} = \frac{\mathrm{d}q_2}{v_{11}} = \frac{\mathrm{d}s_1}{0},$$

whose integration provides s_1. Equation (1.51) leads to:

$$\boldsymbol{v}_1 \cdot \left.\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t}\right|_{x=X_1(t)} + \boldsymbol{v}_1 \cdot \boldsymbol{S} = 0,$$

where the characteristic curve $x = X_1(t)$ satisfies $\mathrm{d}X_1/\mathrm{d}t = \lambda_1$. It is called the 1-characteristic. We have:

$$\mu_1 \left.\frac{\mathrm{d}s_1}{\mathrm{d}t}\right|_{x=X_1(t)} = \boldsymbol{v}_1 \cdot \boldsymbol{S}.$$

Similarly for s_2:

$$\mu_2 \left.\frac{\mathrm{d}s_2}{\mathrm{d}t}\right|_{x=X_2(t)} = \boldsymbol{v}_2 \cdot \boldsymbol{S}.$$

The compact form of Eq. (1.51) after the change of variable is:

$$\left.\frac{\mathrm{d}\boldsymbol{s}}{\mathrm{d}t}\right|_{\boldsymbol{s}=\boldsymbol{X}(t)} = \boldsymbol{L} \cdot \boldsymbol{S} \text{ along } \frac{\mathrm{d}\boldsymbol{X}}{\mathrm{d}t} = \boldsymbol{\Lambda}, \tag{1.54}$$

where $\boldsymbol{L} = [\boldsymbol{v}_1,\, \boldsymbol{v}_2]^T$, $\boldsymbol{\Lambda} = \{\lambda_1, \lambda_2\}$ and $\boldsymbol{s} = \{s_1,\, s_2\}$.

1.3.2 Shock wave and the Lax entropy condition

The Rankine–Hugoniot equation holds for systems of nonlinear hyperbolic equations (1.51). The shock velocity is

$$\dot{\sigma} = \frac{[\![\boldsymbol{f}(\boldsymbol{q})]\!]}{[\![\boldsymbol{q}]\!]}. \tag{1.55}$$

This provides a system of m equations. By eliminating $\dot{\sigma}$, we obtain $m-1$ equations, and thus there are at least $m-1$ curves in the phase plane, called the *Hugoniot locus*. These equations may, however, define more than $m-1$ one-parameter family of curves, as will be illustrated below with the Saint-Venant equations (see § 1.4).

We have seen in §,1.2.3 that a shock wave occurs when the characteristic curves intersect. In other words, for a right-going shock, the characteristic speed (which is also the eigenvalue) on the left is higher than the one on the right. It can also be shown by considering energy balance or by regularising the homogenous equation that only some characteristic speeds lead to physically admissible solutions. Overall, the constraint on the characteristic speed leads to the *Lax entropy condition*: a shock exists between two states $\boldsymbol{q}_l$ and $\boldsymbol{q}_r$ if there is one index k for which the shock velocity $\dot{\sigma}$ satisfies:

$$\lambda_k(\boldsymbol{q}_l) > \dot{\sigma} > \lambda_k(\boldsymbol{q}_r), \tag{1.56}$$

whereas for the other indices, the characteristic curves do not cross the shock curve:

$$\lambda_i(\boldsymbol{q}_l) < \dot{\sigma} \text{ and } \lambda_i(\boldsymbol{q}_r) < \dot{\sigma} \text{ for } i < k, \tag{1.57}$$

and

$$\lambda_i(\boldsymbol{q}_l) > \dot{\sigma} \text{ and } \lambda_i(\boldsymbol{q}_r) > \dot{\sigma} \text{ for } i > k, \tag{1.58}$$

When working on the Saint-Venant equations (see § 1.4), we will see how to derive these relations by generalising what has been done with one-dimensional equations in §,1.2.3.

If we take the limit of $[\![\boldsymbol{q}]\!] \to 0$ around a state $\boldsymbol{q}_*$, then

$$\lim_{[\![\boldsymbol{q}]\!]\to 0} [\![\boldsymbol{f}(\boldsymbol{q})]\!] = \nabla \boldsymbol{f}(\boldsymbol{q}_*) \cdot [\![\boldsymbol{q}]\!].$$

Making use of the Rankine–Hugoniot equation (1.55), we find that

$$\nabla \boldsymbol{f}(\boldsymbol{q}_*) \cdot [\![\boldsymbol{q}]\!] = \dot{\sigma}[\![\boldsymbol{q}]\!],$$

which shows that shock waves in the close vicinity of $\boldsymbol{q}_*$ are right eigenvectors of the Jacobian $\nabla \boldsymbol{f}(\boldsymbol{q}_*)$. In the next section, we will find that rarefaction waves are tangent to the right eigenvector field. As a consequence, in the close vicinity of $\boldsymbol{q}_*$, rarefaction and shock waves are tangent to the same integral curves. This remarkable property will be used by approximate Riemann solvers, which substitute the Jacobian $\nabla \boldsymbol{f}(\boldsymbol{q}_*)$ by a constant matrix (see § 2.4.2). Figure 1.5 shows an example with the shallow water equations.

1.3.3 Rarefaction wave

Definition of simple waves

Before defining rarefaction waves for nonlinear hyperbolic systems, we need a more general concept that extends the concept of *simple wave* seen in § 1.1.5. A simple wave is a special solution to the homogeneous hyperbolic system

$$\frac{\partial}{\partial t}\boldsymbol{q} + \boldsymbol{A}(\boldsymbol{q}) \cdot \frac{\partial}{\partial x}\boldsymbol{q} = \boldsymbol{0} \tag{1.59}$$

where the solution $\boldsymbol{q}(x,t)$ is sought in the form

$$\boldsymbol{q}(x,t) = \boldsymbol{q}(\xi(x,t)),$$

where $\xi(x,t)$ is a function of x and t. If we substitute $\boldsymbol{q}(x,t)$ with $\boldsymbol{q}(\xi)$ into Eq. (1.59), we then obtain

$$\frac{\partial \xi}{\partial t}\boldsymbol{q}' + \frac{\partial \xi}{\partial x}\boldsymbol{A}(\boldsymbol{q}) \cdot \boldsymbol{q}' = \boldsymbol{0}.$$

It is possible to solve this equation if we further assume that a simple wave is the integral curve of one of the right eigenvectors $\boldsymbol{w}_k$, or in other words, we assume that $\boldsymbol{q}'$ and $\boldsymbol{w}_k$ are collinear:

$$\frac{\mathrm{d}}{\mathrm{d}\xi}\boldsymbol{q} = \alpha \boldsymbol{w}_k, \tag{1.60}$$

where α is a proportionality factor (possibly a function of ξ). When $\boldsymbol{q}$ is an integral curve of $\boldsymbol{w}_k$, the function $\xi(x,t)$ satisfies the equation

$$\frac{\partial \xi}{\partial t} + \lambda_k(\boldsymbol{q})\frac{\partial \xi}{\partial x} = 0. \tag{1.61}$$

Note that this equation is a nonlinear hyperbolic equation and thus may develop discontinuities even though $\xi(x,t)$ is initially continuous.

Note that the characteristic form of Eq. (1.61) is

$$\frac{\mathrm{d}\xi}{\mathrm{d}t} = 0 \text{ along the curve } \frac{\mathrm{d}x}{\mathrm{d}t} = \lambda_k(\boldsymbol{q}),$$

which implies that ξ is constant along the characteristic curve $x'(t) = \lambda_k(\boldsymbol{q}(\xi))$, but since the characteristic speed $\lambda_k(\boldsymbol{q}(\xi))$ is also constant, then the characteristic curves are straight lines. Along each of these characteristic curves, $\boldsymbol{q}(\xi)$ is constant. A simple wave can be interpreted as a wave that carries some constant information at a constant velocity.

If we assume that $\lambda_k(\boldsymbol{q}(x,0))$ is initially monotonically increasing in x at time $t=0$, the characteristic curves will not cross each other at later times, and thus a continuous solution will exist for all time. In the opposite case, if $\lambda_k(\boldsymbol{q}(x,0))$ is locally decreasing in x at time $t=0$, the wave undergoes compression, and thus an initially smooth simple wave may compress to the point of forming a shock in the region where the characteristic speed λ_k decreases. To determine whether the characteristic speed λ_k is a monotonically increasing function, we can compute

$$\frac{\mathrm{d}}{\mathrm{d}\xi}\lambda_k(\boldsymbol{q}(\xi)) = \nabla\lambda_k \cdot \boldsymbol{q}'. \tag{1.62}$$

Since $\boldsymbol{q}'$ is a right eigenvector of $\boldsymbol{A}$, $\boldsymbol{q}'$ is collinear with the right eigenvector $\boldsymbol{w}_k$, and thus the monotonic character is provided by the sign of $\nabla\lambda_k \cdot \boldsymbol{w}_k$. When

$$\nabla\lambda_k \cdot \boldsymbol{w}_k \neq 0 \tag{1.63}$$

the characteristic speed is said to be a *genuinely nonlinear field.* When

$$\nabla\lambda_k \cdot \boldsymbol{w}_k = 0 \tag{1.64}$$

for all $\boldsymbol{q}$, the field is said to be *linearly degenerate.*

To sum up, we can define simple waves as curves $\boldsymbol{q}(\xi(x,t))$ where $\boldsymbol{q}$ is an integral curve of the right eigenvector $\boldsymbol{w}_k$. The function ξ must satisfy the nonlinear advection equation (1.61). Characteristic curves in the $x-t$ plane are thus straight lines of slope $\lambda_k(\boldsymbol{q}(\xi))$. A geometric interpretation is that the curve $\boldsymbol{q}(\xi)$ is tangent to the vector field $\boldsymbol{w}_k$. Note that if we seek a function $r(\boldsymbol{q})$ that remains constant along this integral curve, then we recover the definition (1.52) of the Riemann invariant:

$$\frac{\mathrm{d}}{\mathrm{d}\xi}r(\boldsymbol{q}) = 0 \Rightarrow \nabla r \cdot \boldsymbol{q}' = 0,$$

and since $\boldsymbol{q}' = \alpha\boldsymbol{w}_k$, then the invariance condition is:

$$\nabla r \cdot \boldsymbol{w}_k = 0. \tag{1.65}$$

As indicated in § 1.3.1, this property gives an operational way of defining the k-Riemann invariant: it is a function whose gradient is normal to the right eigenvector $\boldsymbol{w}_k$ at each point $\boldsymbol{q}$. In general, we can find m–1 distinct k-Riemann invariants for each family k: the solution to $\boldsymbol{w}_k \cdot \nabla r_k = 0$ gives m equations, but eliminating ξ reduces by one the number of independent solutions (see the example of the Saint-Venant equations in the next section).

Definition of rarefaction waves

A (centred) rarefaction wave is a particular case of simple wave for which $\xi = x/t$. The interest of this special solution is obvious when considering that the governing equation (1.59) is invariant to the stretching group $x \to \alpha x$ and $t \to \alpha t$. The function ξ is thus the similarity variable $\xi = x/t$. We are seeking a similarity solution $\boldsymbol{q} = \boldsymbol{q}(\xi)$. Substituting this form into Eq. (1.59) gives

$$-\xi \boldsymbol{q}' + \boldsymbol{A} \cdot \boldsymbol{q}' = 0,$$

which shows that $\boldsymbol{q}'$ is a right eigenvector of $\boldsymbol{A}$, imposing $\xi = \lambda_k$ and $\boldsymbol{q}'$ collinear with the right eigenvector $\boldsymbol{w}_k$.

The condition $\xi = \lambda_k(\boldsymbol{q}(\xi))$ gives us the possibility to derive a formal definition of the integral curve $\boldsymbol{q}(\xi)$. By differentiating this condition with respect to ξ, we obtain

$$1 = \nabla \lambda_k \cdot \boldsymbol{q}'(\xi)$$

where the gradient $\nabla \lambda_k$ is obtained by differentiating the characteristic speed λ_k with respect to $\boldsymbol{q}$. Since $\boldsymbol{q}'(\xi)$ is collinear with the right eigenvector $\boldsymbol{w}_k$, there is a scalar α such that $\boldsymbol{q}' = \alpha \boldsymbol{w}_k$. Substituting this form in the equation above, we obtain

$$1 = \alpha \nabla \lambda_k \cdot \boldsymbol{w}_k,$$

and we eventually deduce

$$\alpha = \frac{1}{\nabla \lambda_k \cdot \boldsymbol{w}_k},$$

and the integral curve thus satisfies the differential equation

$$\boldsymbol{q}'(\xi) = \frac{\boldsymbol{w}_k}{\nabla \lambda_k \cdot \boldsymbol{w}_k}, \tag{1.66}$$

whose solution is denoted by $\tilde{\boldsymbol{q}}$.

The characteristics are straight lines $x = mt$, where m is a real factor satisfying $\xi_l < m < \xi_r$ with $\xi_l \lambda_k(\boldsymbol{q}_l)$ and $\xi_r = \lambda_k(\boldsymbol{q}_r)$. Equation (1.66) holds for $\xi_l < \xi < \xi_r$ and satisfies the boundary conditions

$$\boldsymbol{q}(\xi_l) = \boldsymbol{q}_l \text{ and } \boldsymbol{q}(\xi_r) = \boldsymbol{q}_r.$$

The rarefaction wave's structure is then

$$\boldsymbol{q}(x,t) = \begin{cases} \boldsymbol{q}_l & \text{if } x/t \le \xi_1, \\ \tilde{\boldsymbol{q}}(x/t) & \text{if } \xi_1 \le x/t \le \xi_2, \\ \boldsymbol{q}_r & \text{if } x/t \ge \xi_2, \end{cases} \tag{1.67}$$

where

$$\xi_1 = \lambda_k(\boldsymbol{q}_l) \text{ and } \xi_2 = \lambda_k(\boldsymbol{q}_r).$$

Since a k-rarefaction wave is a simple wave, the k-Riemann invariant given by (1.65) is constant along this wave.

1.4 Example: the Saint-Venant equations

1.4.1 Governing equations and characteristic form

For water waves over horizontal frictionless beds, the governing equations (called Saint-Venant or shallow water equations) are given by

$$\frac{\partial h}{\partial t}+\frac{\partial h\bar{u}}{\partial x} = 0 \tag{1.68}$$

$$\frac{\partial h\bar{u}}{\partial t}+\frac{\partial h\bar{u}^2}{\partial x}+gh\frac{\partial h}{\partial x} = 0, \tag{1.69}$$

which can be cast in tensorial form (1.59) with:

$$\boldsymbol{q}=\begin{pmatrix} h \\ q \end{pmatrix} \text{ and } \boldsymbol{A}=\boldsymbol{f}'=\begin{pmatrix} 0 & 1 \\ -q^2/h^2+gh & 2q/h \end{pmatrix}, \tag{1.70}$$

where h and $q=h\bar{u}$ denote the flow depth and momentum, g is gravity acceleration, and $\bar{u}$ is depth-averaged velocity. By integrating $\boldsymbol{f}'=\boldsymbol{A}$, we can determine the flux function $\boldsymbol{f}$:

$$\boldsymbol{f}=\begin{pmatrix} q \\ \dfrac{q^2}{h}+\dfrac{1}{2}gh^2 \end{pmatrix}. \tag{1.71}$$

The eigenvalues of the Jacobian matrix $\boldsymbol{A}$ are

$$\lambda_1=\bar{u}-c \text{ and } \lambda_2=\bar{u}+c, \tag{1.72}$$

where $c=\sqrt{gh}$ and the right eigenvectors are

$$\boldsymbol{w}_1=\begin{pmatrix} 1 \\ \bar{u}-c \end{pmatrix} \text{ and } \boldsymbol{w}_2=\begin{pmatrix} 1 \\ \bar{u}+c \end{pmatrix}. \tag{1.73}$$

If we define the 1-Riemann variable s_1 as $\nabla s_1\cdot\boldsymbol{w}_2=0$ (see § 1.3.1), then s_1 is the solution to

$$\frac{\partial s_1}{\partial h}+(\bar{u}+c)\frac{\partial s_1}{\partial q}=0 \Leftrightarrow \frac{\mathrm{d}h}{1}=\frac{\mathrm{d}q}{\bar{u}+c}=\frac{\mathrm{d}s_1}{0}.$$

Integrating the first pair of equations gives

$$\mathrm{d}q=(\bar{u}+c)\mathrm{d}h \Rightarrow \bar{u}\mathrm{d}h+h\mathrm{d}\bar{u}=(\bar{u}+c)\mathrm{d}h.$$

After simplification and integration, we obtain

$$\mathrm{d}\bar{u}=\sqrt{\frac{g}{h}}\mathrm{d}h \Rightarrow \bar{u}=2\sqrt{gh}+a$$

where a is a constant of integration. As s_1 is an arbitrary function of a, we select the simplest form and thus set $a=s_1=\bar{u}-2\sqrt{gh}$. The first Riemann variable s_1 is defined as

$$s_1=\bar{u}-2\sqrt{gh}. \tag{1.74}$$

Similarly for the 2-variable s_2, we find

$$s_2=\bar{u}+2\sqrt{gh} \tag{1.75}$$

The Saint-Venant equations (1.68)–(1.69) are thus equivalent to

$$\left\{\begin{array}{l} \dfrac{\mathrm{d}s_1}{\mathrm{d}t}=0 \text{ along } \dfrac{\mathrm{d}x}{\mathrm{d}t}=\lambda_1=\bar{u}-c, \\ \dfrac{\mathrm{d}s_2}{\mathrm{d}t}=0 \text{ along } \dfrac{\mathrm{d}x}{\mathrm{d}t}=\lambda_2=\bar{u}+c. \end{array}\right. \tag{1.76}$$

1.4.2 Energy

By multiplying the momentum balance equation (1.69) by the depth-averaged velocity $\bar{u}$, we can derive an equation for the energy (per unit mass):

$$\bar{u}\frac{\partial h\bar{u}}{\partial t} + \bar{u}\frac{\partial}{\partial x}\left(h\bar{u}^2 + g\frac{h^2}{2}\right) = 0. \tag{1.77}$$

Using Leibniz product role, we have

$$\bar{u}\frac{\partial h\bar{u}}{\partial t} = \bar{u}\frac{\partial h\bar{u}^2}{\partial t} - h\bar{u}\frac{\partial \bar{u}}{\partial t}.$$

To determine the last term on the right side of the momentum balance equation (1.69) in non-conservation form

$$\frac{\partial \bar{u}}{\partial t} + \bar{u}\frac{\partial \bar{u}}{\partial x} + g\frac{\partial h}{\partial x} = 0.$$

Equation (1.77) becomes

$$\frac{\partial h\bar{u}^2}{\partial t} + \bar{u}\frac{\partial}{\partial x}\left(h\bar{u}^2 + g\frac{h^2}{2}\right) + h\bar{u}^2\frac{\partial \bar{u}}{\partial x} + gh\bar{u}\frac{\partial h}{\partial x} = 0. \tag{1.78}$$

The idea is to put the equation in conservative form. By rearranging the terms in Eq. (1.78),we obtain

$$\frac{\partial h\bar{u}^2}{\partial t} + \frac{\partial}{\partial x}\left(h\bar{u}^3\right) + 2gh\bar{u}\frac{\partial h}{\partial x} = 0. \tag{1.79}$$

We can express the flow-depth gradient as follows

$$2gh\bar{u}\frac{\partial h}{\partial x} = 2g\left(\frac{\partial h^2\bar{u}}{\partial x} - h\frac{\partial h\bar{u}}{\partial x}\right).$$

From the mass balance equation (1.68), we get

$$h\frac{\partial h\bar{u}}{\partial x} = -\frac{1}{2}\frac{\partial h^2}{\partial t}.$$

Substituting this equation into Eq. (1.79), we eventually arrive at the energy balance equation

$$\frac{\partial}{\partial t}\left(\frac{1}{2}h\bar{u}^2 + \frac{1}{2}gh^2\right) + \frac{\partial}{\partial x}\left(\frac{1}{2}h\bar{u}^3 + gh^2\bar{u}\right) = 0. \tag{1.80}$$

This equation can be cast in the generic form

$$\frac{\partial E}{\partial t} + \frac{\partial F}{\partial x} = 0.$$

where the total energy E comprises kinetic and hydrostatic contributions:

$$E = \frac{1}{2}h\bar{u}^2 + \frac{1}{2}gh^2 \text{ and } F = \frac{1}{2}h\bar{u}^3 + gh^2\bar{u}.$$

This equation is associated with a Rankine–Hugoniot that describes the energy jump across a hydraulic jump, which should be

$$[\![F]\!] - \dot{\sigma}[\![E]\!] = 0. \tag{1.81}$$

There is a consensus to state that energy is not conserved across a discontinuity, but should decrease because of internal energy dissipation (Rayleigh, 1914; Stoker, 1957; Whitham, 1974; Antuono, 2010; Richard & Gavrilyuk, 2013; Kalisch *et al.*, 2017; Paulsen & Kalisch, 2020). Let us define the energy jump

$$\Delta E = [\![F]\!] - \dot{\sigma}[\![E]\!]. \tag{1.82}$$

Energy should be dissipated across a jump, which implies the following inequality

$$\Delta E < 0, \tag{1.83}$$

which can be regarded as the equivalence of the entropy condition in gas dynamics. The difference $[\![E]\!]$ must be calculated by considering that the flow goes from left to right. The definition of total energy E in Eq. (1.82) can be simplified a great deal. We need to introduce the relative velocity

$$v_r = u_r - \dot\sigma \text{ and } v_l = u_l - \dot\sigma,$$

and the mass flux across the discontinuity $x = \sigma(t)$

$$\dot m = h_l v_l = h_r v_r \tag{1.84}$$

which is the Rankine–Hugoniot equation (1.89) expressed in the frame attached to the discontinuity (see below). From the Rankine–Hugoniot equation (1.89) associated with momentum balance, we obtain

$$h_l v_l (v_r - v_l) = -\frac{1}{2} g(h_r^2 - h_l^2), \tag{1.85}$$

and thus by solving Eqs (1.84)–(1.85), we deduce the relationship between relative velocities and flow depths

$$v_l^2 = \frac{g}{2}\frac{h_r}{h_l}(h_l + h_r) \text{ and } v_r^2 = \frac{g}{2}\frac{h_l}{h_r}(h_l + h_r). \tag{1.86}$$

With these notations, we can express the total energy jump across the discontinuity (1.82)

$$\begin{aligned}
\Delta E &= [\![F]\!] - \dot\sigma [\![E]\!], \\
&= \left[\!\!\left[\frac{1}{2} h \bar u^2 v\right]\!\!\right] + \left[\!\!\left[g h^2 \bar u - \frac{\dot\sigma}{2} g h^2 \right]\!\!\right], \\
&= \left[\!\!\left[\frac{1}{2} \dot m \bar u^2\right]\!\!\right] + \left[\!\!\left[\frac{1}{2} g h^2 (\bar u + v)\right]\!\!\right], \\
&= \left[\!\!\left[\frac{1}{2} \dot m (v + \dot\sigma)^2\right]\!\!\right] + \left[\!\!\left[\frac{1}{2} g h^2 (2v + \dot\sigma)\right]\!\!\right], \\
&= \dot m \left[\!\!\left[\frac{1}{2} v^2 + g h\right]\!\!\right],
\end{aligned}$$

which finds an easy interpretation: ΔE is the jump in the quantity $\Psi = \dot m(\frac{1}{2}v^2 + gh)$ representing the product of the mass flux $\dot m$ and head $\frac{1}{2}v^2 + gh$ (per unit mass). By using the relations (1.85), we eventually find

$$\Delta E = -\dot m g \frac{(h_r - h_l)^3}{4 h_l h_r}. \tag{1.87}$$

The inequality (1.83) implies that

$$\Delta E < 0 \Rightarrow \begin{cases} \text{if } \dot m > 0, \text{ then } h_r > h_l, \\ \text{if } \dot m < 0, \text{ then } h_r < h_l. \end{cases} \tag{1.88}$$

If this inequality is not satisfied, then the shock is not physically admissible.

1.4.3 Shock wave

From the Rankine–Hugoniot equation (1.55), we can determine the shock velocity and its features:

$$\dot\sigma \left[\!\!\left[\begin{array}{c} h \\ q \end{array} \right]\!\!\right] = \left[\!\!\left[\begin{array}{c} q \\ \dfrac{q^2}{h} + \dfrac{1}{2} g h^2 \end{array} \right]\!\!\right]. \tag{1.89}$$

Let us determine the Hugoniot locus, that is, the locus of points that are connected to a given point (h_*, q_*) in the phase plane and for which the Rankine–Hugoniot equation (1.89) is satisfied. The mass balance equation of Eq. (1.89) leads to

$$\dot{\sigma} = \frac{q - q_*}{h - h_*}. \tag{1.90}$$

Substituting this relationship into the momentum balance of Eq. (1.89) leads to the equation

$$q - q_* = \frac{h - h_*}{q - q_*}\left(\frac{q^2}{h} + \frac{1}{2}gh^2 - \frac{q_*^2}{h_*} - \frac{1}{2}gh_*^2\right).$$

The solution to this quadratic equation is

$$q(h|h_*, q_*) = \frac{h}{h_*}q_* \pm (h - h_*)\sqrt{\frac{1}{2}g\frac{h}{h_*}(h + h_*)}.$$

This equation defines two curves:

- The 1-shock curve

$$q(h|h_*, q_*) = \frac{h}{h_*}q_* - (h - h_*)\sqrt{\frac{1}{2}g\frac{h}{h_*}(h + h_*)}. \tag{1.91}$$

- The 2-shock curve

$$q(h|h_*, q_*) = \frac{h}{h_*}q_* + (h - h_*)\sqrt{\frac{1}{2}g\frac{h}{h_*}(h + h_*)}. \tag{1.92}$$

From Eq. (1.90), we deduce that the shock moves at the velocity

$$\dot{\sigma} = u_* \pm \sqrt{gh\frac{h_* + h}{2h_*}}. \tag{1.93}$$

Figure 1.5 shows the two shock curves issuing from point P with coordinate $(h_*, q_*) = (2, 1)$. One remarkable property is that at point P, the show curves are tangent to the eigenvectors $\boldsymbol{w}_i$ $(i = 1, 2)$ of the Jacobian matrix $\boldsymbol{A}$. Indeed, in the limit $\boldsymbol{q} \to \boldsymbol{q}_*$, the Rankine–Hugoniot equation (1.55) can be linearised

$$\dot{\sigma}[\![\boldsymbol{q}]\!] = \boldsymbol{f}'(\boldsymbol{q}_*) \cdot [\![\boldsymbol{q}]\!] \tag{1.94}$$

$$= \boldsymbol{A}(\boldsymbol{q}_*) \cdot [\![\boldsymbol{q}]\!], \tag{1.95}$$

which shows that in this limit, $[\![\boldsymbol{q}]\!]$ is an eigenvector of $\boldsymbol{A}$, and thus is collinear to either $\boldsymbol{w}_1$ or $\boldsymbol{w}_2$. As a consequence, rarefaction curves are also tangent to the shock curves at point P. This property is used in numerical algorithms (see Chap. 2).

Physical admissibility based on energy balance

Not all points on the shock curves are physically admissible. We have seen that a shock is said to be *physically admissible* when it dissipates energy. If it creates energy, it is not admissible. This condition has been called the *entropy condition.* One way of looking at the physical relevance of the solution is to compute the energy change across the shock. The energy jump is given by Eq. (1.87):

$$\Delta E = \dot{m}\left[\!\!\left[\frac{1}{2}\bar{v}^2 + gh\right]\!\!\right] = -\dot{m}g\frac{(h_r - h_l)^3}{4h_r h_r},$$

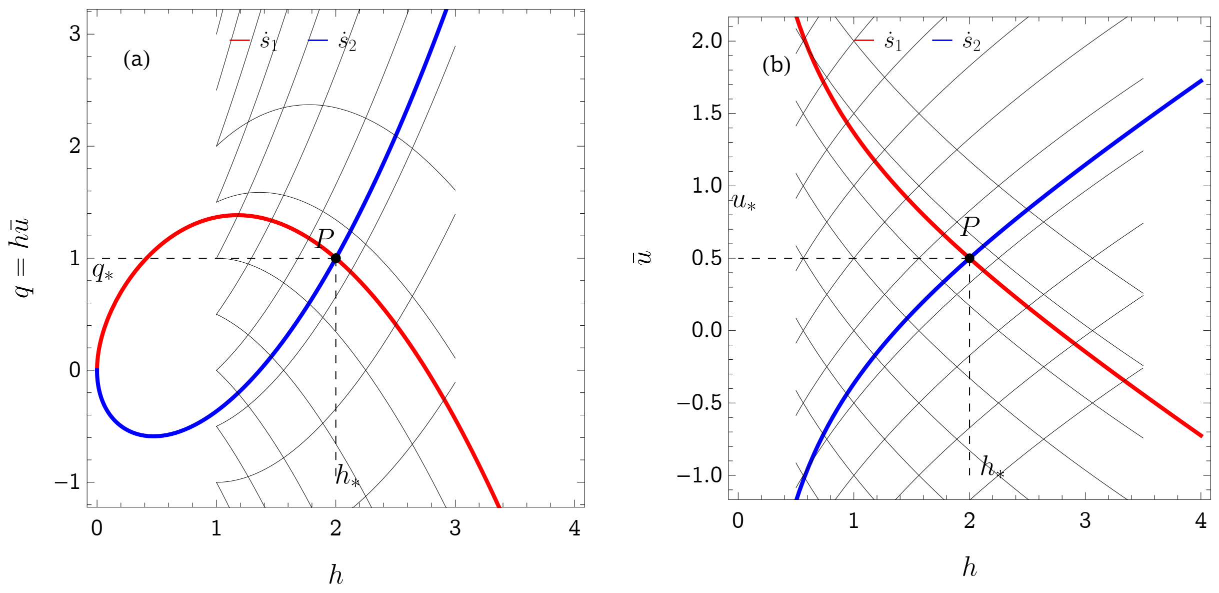


Figure 1.5 Hugoniot locus of all points connected to point P with coordinate $(h_*, q_*) = (2, 1)$ through the 1-shock curve (1.91) or 2-shock curve (1.92). The red curve shows the 1-shock wave (1.91), while the blue curve shows the 2-shock wave (1.92). The thin lines show the contour plot of the Riemann invariants r_1 and r_2 given by Eqs. (1.74) and (1.75), respectively. (a) Curves in the $(h, q = h\bar{u})$ plane. (b) Curves in the $(h, \bar{u})$ plane.

where $v = u - \dot{\sigma}$ is the relative velocity and $\dot{m} = h_l v_l = h_r u_r$ is the mass flux across the discontinuity $x = \sigma(t)$. Shocks are physically admissible if

$$\Delta E < 0.$$

The sign of ΔE depends on the mass flux $\dot{m}$ and the difference $h_r - h_l$. To determine the sign of $\dot{m}$ depending on the left and right states, we need to do some algebraic manipulations. Let us note that the shock speed (1.90) can be cast in the following form

$$\begin{aligned}\dot{\sigma} &= \frac{h_r u_r - h_l u_l}{h_r - h_l},\\ &= \frac{h_r(u_r - u_l) + (h_r - h_l)u_l}{h_r - h_l},\\ &= \frac{[\![u]\!]}{[\![h]\!]} h_r + u_l,\end{aligned}$$

and thus, we deduce

$$v_l = u_l - \dot{\sigma} = -\frac{[\![u]\!]}{[\![h]\!]} h_r.$$

Similarly for the relative velocity v_r, we have

$$\begin{aligned}\dot{\sigma} &= \frac{u_r(h_r - h_l) + (u_r - u_l)h_l}{h_r - h_l},\\ &= \frac{[\![u]\!]}{[\![h]\!]} h_l + u_r,\end{aligned}$$

and thus, we deduce

$$v_r = u_r - \dot{\sigma} = -\frac{[\![u]\!]}{[\![h]\!]} h_l.$$

As shown by Fig. 1.6, we consider a point P which can be the left or right state. For each possibility, we seek the Hugoniot loci of states that are physically admissible. To that end, we determine the sign of ΔE by looking at the sign of $\dot{m} = h_l v_l$ (or $\dot{m} = h_r v_r$) and that of $h_r - h_l$. For each considered point, we can split the plane into four quadrants, and evaluate the sign of $\dot{m}$ and $h_r - h_l$. The solid lines in Fig. 1.6 show the parts of the 1- and 2-shock curves for which $\Delta E < 0$, whereas the dashed lines show the non-physical parts ($\Delta E > 0$).

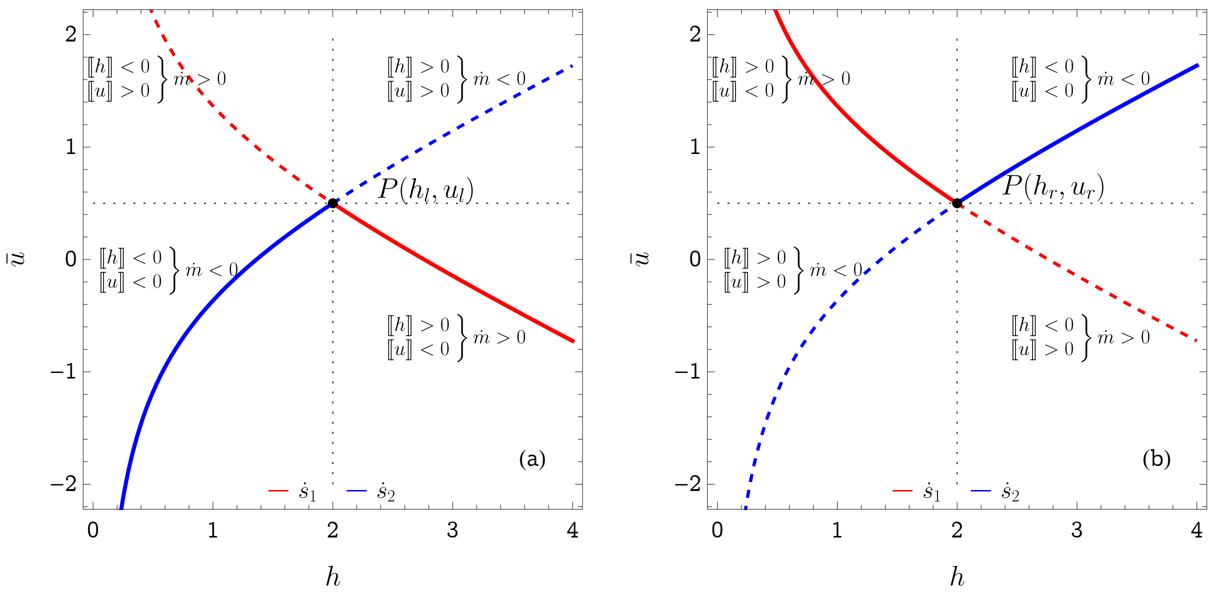


Figure 1.6 Determination of the admissible states. (a) When the left state (h_l, u_l) is fixed, we look for which part of the curve the relationship $\Delta E < 0$ is satisfied. (b) When the right state (h_r, u_r) is fixed, we look for which part of the curve the relationship $\Delta E < 0$ is satisfied. The red curve shows the 1-shock wave (1.91), while the blue curve shows the 2-shock wave (1.92).

Physical admissibility based on regularisation

Like for one-dimensional problems seen in § 1.2.2, we consider the regularised version of the Saint-Venant equation by a adding a viscous term

$$\frac{\partial}{\partial t}\boldsymbol{q} + \frac{\partial}{\partial x}\boldsymbol{f}(\boldsymbol{q}) = \mu \frac{\partial^2}{\partial x^2}\boldsymbol{q} \tag{1.96}$$

where μ is the viscosity ($\mu > 0$ and we will take $\mu \to 0$) and

$$\boldsymbol{q} = \begin{pmatrix} h \\ q \end{pmatrix} \text{ and } \boldsymbol{f} = \begin{pmatrix} q \\ \dfrac{q^2}{h} + \dfrac{1}{2}gh^2 \end{pmatrix}.$$

We are seeking a travelling-wave solution the regularised Saint-Venant equations (1.96)

$$\boldsymbol{q}(x,t) = \boldsymbol{U}(\xi) \text{ with } \xi = \frac{x - \dot{\sigma}t}{\mu} \tag{1.97}$$

where $\dot{\sigma}$ is the wave velocity given by the Rankine–Hugnoniot equation (1.89), and $\boldsymbol{U}$ satisfies the boundary conditions

$$\left\{ \begin{array}{l} \boldsymbol{U} \to \boldsymbol{q}_l \text{ when } \xi \to -\infty, \\ \boldsymbol{U} \to \boldsymbol{q}_r \text{ when } \xi \to +\infty. \end{array} \right. \tag{1.98}$$

Substituting the form (1.97) into the regularised Saint-Venant equations (1.96) gives

$$-\dot{\sigma}\frac{\mathrm{d}}{\mathrm{d}\xi}\boldsymbol{U}+\frac{\mathrm{d}}{\mathrm{d}\xi}\boldsymbol{f}(\boldsymbol{U})=\frac{\mathrm{d}^2}{\mathrm{d}\xi^2}\boldsymbol{U},$$

which, after integration and determining the constant of integration from the boundary conditions (1.98), yields

$$\frac{\mathrm{d}}{\mathrm{d}\xi}\boldsymbol{U}=\boldsymbol{V}(\boldsymbol{U}) = -\dot{\sigma}(\boldsymbol{U}-\boldsymbol{q}_l)+\boldsymbol{f}(\boldsymbol{U})-f(\boldsymbol{q}_l) \tag{1.99}$$

$$= -\dot{\sigma}(\boldsymbol{U}-\boldsymbol{q}_r)+\boldsymbol{f}(\boldsymbol{U})-f(\boldsymbol{q}_r). \tag{1.100}$$

We note that $\boldsymbol{q}_l$ and $\boldsymbol{q}_r$ are equilibrium points of $\boldsymbol{V}(\boldsymbol{U})$. To determine the behaviour near these points, we linearise $\boldsymbol{V}(\boldsymbol{U})$. For instance, near $\boldsymbol{q}_l$, we have to first order

$$\boldsymbol{V}(\boldsymbol{U})=(-\dot{\sigma}\mathbf{1}+\boldsymbol{A})\cdot(\boldsymbol{U}-\boldsymbol{q}_l)=\boldsymbol{B}\cdot(\boldsymbol{U}-\boldsymbol{q}_l)$$

where the matrix $\boldsymbol{B}$ (evaluated at $\boldsymbol{q}_l$)

$$\boldsymbol{B}=\boldsymbol{A}-\dot{\sigma}\mathbf{1}=\left[\begin{array}{cc}-\dot{\sigma} & 1\\ gh-\frac{q^2}{h^2} & 2\frac{q}{h}-\dot{\sigma}\end{array}\right]$$

has the eigenvalues $-\dot{\sigma}+\bar{u}_l\pm\sqrt{gh_l}$ or, written differently, $-\dot{\sigma}+\lambda_i(\boldsymbol{q}_l)$. Similarly, near $\boldsymbol{q}_r$, we have to first order

$$\boldsymbol{V}(\boldsymbol{U})=(-\dot{\sigma}\mathbf{1}+\boldsymbol{A})\cdot(\boldsymbol{U}-\boldsymbol{q}_r)=\boldsymbol{B}\cdot(\boldsymbol{U}-\boldsymbol{q}_r)$$

where the matrix $\boldsymbol{B}$ is now evaluated at $\boldsymbol{q}_r$ and has the eigenvalues $-\dot{\sigma}+\lambda_i(\boldsymbol{q}_r)$.

We will show that travelling wave solutions to the regularised Saint-Venant equations exist only if we constrain the eigenvalues of $\boldsymbol{B}$. Here we provide a simple proof based on the expected behaviour of the solutions in the phase plane; the reader is referred to technical books for more formal proofs (Smoller, 1982; Holden & Risebro, 2015). Let us first consider the 1-shock curve connecting the left (L) and right (R) states and associated with the eigenvalue $\eta_1-\dot{\sigma}+\lambda_1$. For $\xi\rightarrow-\infty$, the integral curve of Eq. (1.97) starts from L. When seeking local solutions to Eq. (1.97) near L in the form $\boldsymbol{U}=\boldsymbol{q}_l+\boldsymbol{U}_0e^{\eta_1\xi}$ (with $\boldsymbol{U}_0$ the initial condition which must be collinear to the eigenvector of $\boldsymbol{B}(\boldsymbol{q}_l)$ associated with the eigenvalue η_1), then the condition $\eta_1>0$ at L must be met for the left boundary condition (1.98) to be satisfied. As $\eta_2>\eta_1$, then we deduce that the second eigenvalue is also positive at L: $\eta_2>0$. We then infer the conditions

$$\dot{\sigma}<\lambda_1(\boldsymbol{q}_l)\text{ and }\dot{\sigma}<\lambda_2(\boldsymbol{q}_l) \tag{1.101}$$

As the two eigenvalues η_1 and η_2 are positive, the left state L is a node (source). For the same reason, when looking at the local behaviour near R by seeking solutions in the $\boldsymbol{U}=\boldsymbol{q}_r+\boldsymbol{U}_0e^{\eta_1\xi}$ when $\xi\rightarrow\infty$, then we must have $\eta_1<0$ at R, and thus

$$\dot{\sigma}<\lambda_1(\boldsymbol{q}_l).$$

The right state R must be a saddle point to prevent an integral path issuing from L by (being tangent to the second eigenvector at L) from reaching R. This implies that the second eigenvalue must be positive at R. In the end, we thus have the conditions

$$\dot{\sigma}>\lambda_1(\boldsymbol{q}_r)\text{ and }\dot{\sigma}<\lambda_2(\boldsymbol{q}_r) \tag{1.102}$$

Figure 1.7(a) shows a simplified phase plane where the integral path (also called here a *heteroclinic orbit*) connects the source point L to the saddle point R.

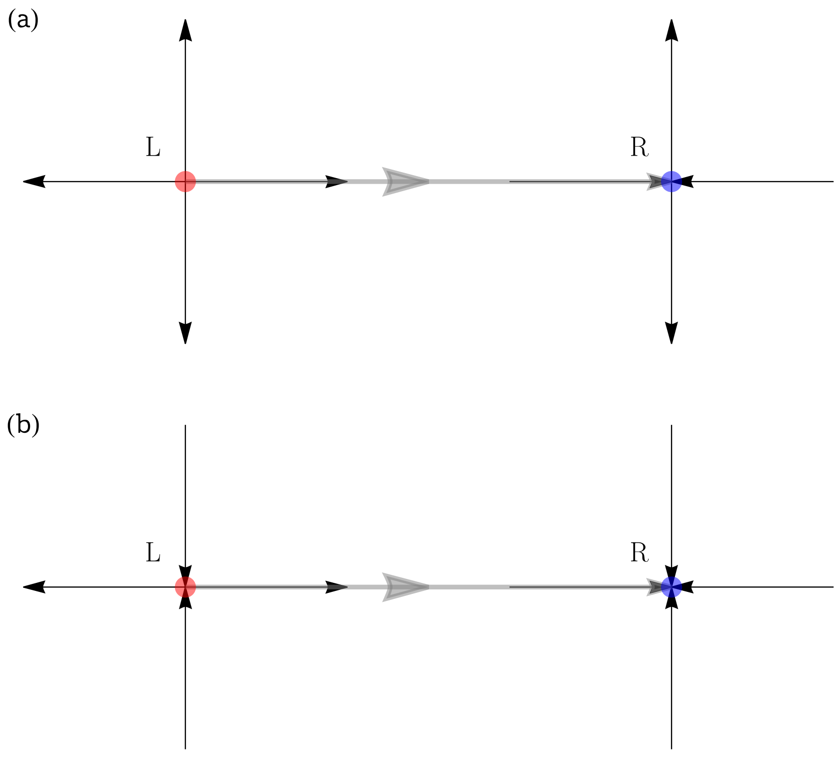


Figure 1.7 Sketch of the phase plane and the behaviour near the left (L) and right (R) states. (a) When considering the 1-shock curve, the left state is a node (source) and the right state is a saddle point. (b) When considering the 2-shock curve, the left state is a saddle point and the right state is a node (sink). The grey line shows the heteroclinic orbit connecting the left and right states.

Recombining Eqs. (1.101) and (1.102), we deduce that a travelling wave (mimicking the 1-shock wave) exists only when the Lax condition is satisfied

$$\lambda_1(\boldsymbol{q}_r) < \dot{\sigma} < \lambda_1(\boldsymbol{q}_l), \tag{1.103}$$

$$\dot{\sigma} < \lambda_2(\boldsymbol{q}_l) \text{ and } \dot{\sigma} < \lambda_2(\boldsymbol{q}_r). \tag{1.104}$$

We could reiterate the reasoning for the 2-shock curve. In that case, we obtain that the left state L must be saddle point whereas the right state R must be a node (sink). A travelling wave (mimicking the 2-shock wave) exists only when the Lax condition is satisfied:

$$\lambda_2(\boldsymbol{q}_r) < \dot{\sigma} < \lambda_2(\boldsymbol{q}_l), \tag{1.105}$$

$$\dot{\sigma} > \lambda_1(\boldsymbol{q}_l) \text{ and } \dot{\sigma} > \lambda_1(\boldsymbol{q}_r). \tag{1.106}$$

Figure 1.7(b) shows the simplified phase plane for the 2-shock curve.

Example

Figure 1.8 shows an example of shocks between two states $\boldsymbol{q}_l = (h_l, q_l) = (2, 1)$ and $\boldsymbol{q}_r = (h_r, q_r) = (2, -0.436)$. The 1-characteristics $x = x_0 + \lambda_1 t$ intersect each other, and thus there is a 1-shock curve separating the two states. The characteristics are parallel straight lines in each wedge where $\boldsymbol{q}$ is constant.

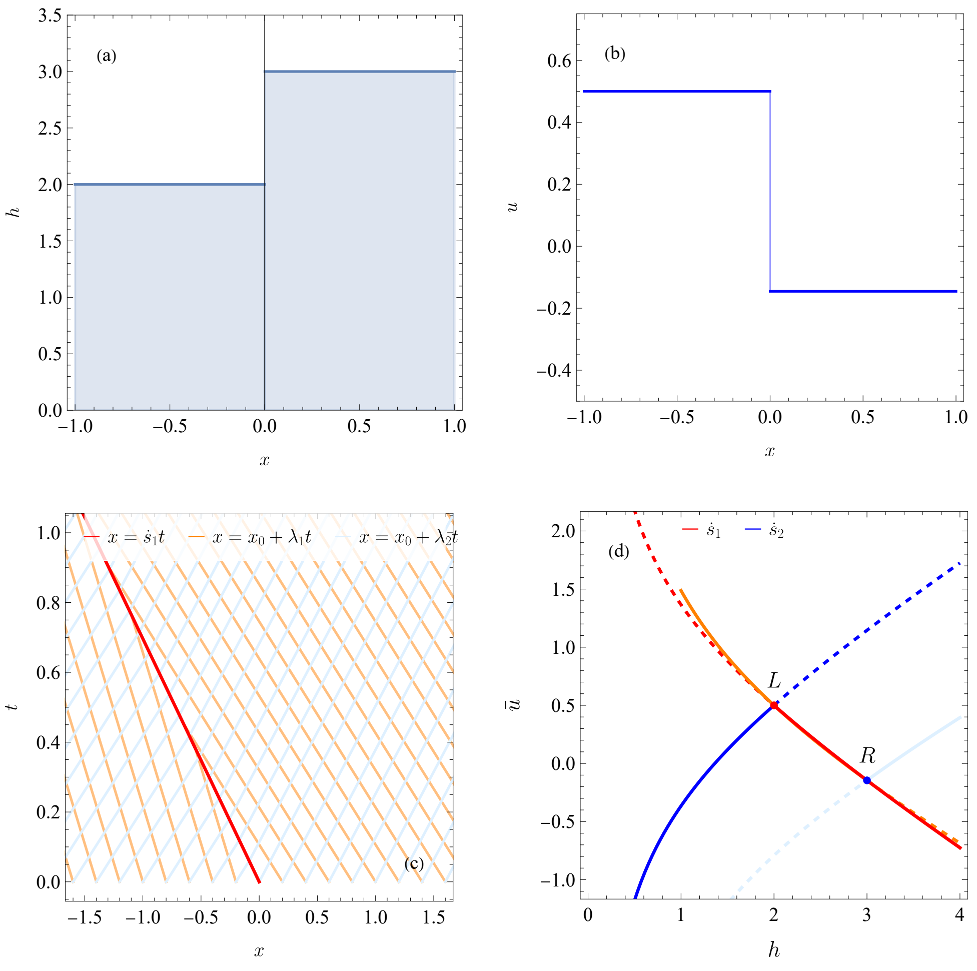


Figure 1.8 Example of Riemann problem with a 1-shock curve as the solution. Left state: $\boldsymbol{q}_l = (h_l, q_l) = (2, 1)$. Right state: $\boldsymbol{q}_r = (h_r, q_r) = (2, -0.436)$. (a) Initial depth profile. (b) Initial velocity profile. (c) Characteristic diagram. (d) Phase plane.

1.4.4 Rarefaction wave

Let us recall that a rarefaction wave is a special case of simple waves, which are defined as integral curves of right eigenvectors. There are thus as many rarefaction waves as right eigenvectors, and these waves are labelled as k-waves when they are associated with the right eigenvector $\boldsymbol{w}_k$. A k-rarefaction wave is the similarity solution $\boldsymbol{q}(\xi)$ (with $\xi = x/t$) to the governing equation (1.59) (see § 1.3.3). The characteristic curves are straight lines $\xi = x/t = m$ where the slope m ranges from $\lambda(\boldsymbol{q}_l)$ to $\lambda(\boldsymbol{q}_r)$. The k-Riemann variable r_k defined by Eq. (1.65) is constant along the k-rarefaction wave.

Riemann invariants

The k-Riemann variable r_k is defined by Eq. (1.65) ($\nabla r_k \cdot \boldsymbol{w}_k = 0$). We will show here that it can also be calculated by using the properties of simple waves.

Let us start with the integral curve of an eigenvector $\boldsymbol{w}_1$. Integrating Eq. (1.60) with the definition of the right eigenvector $\boldsymbol{w}_1$ given by Eq. (1.73) gives

$$\frac{\mathrm{d}}{\mathrm{d}\xi}\begin{pmatrix} h \\ q \end{pmatrix} = \begin{pmatrix} 1 \\ q/h - \sqrt{gh} \end{pmatrix}. \tag{1.107}$$

The first equation tells us that $h' = 1$, and thus we obtain $h = \xi + a$. We assume that $a = 0$ such that ξ and h coincide. The second equation then becomes

$$\frac{\mathrm{d}q}{\mathrm{d}h} = q/h - \sqrt{gh}. \tag{1.108}$$

Let us assume that at $h = h_*$, $q = q_* = h_*\bar{u}_*$. Integrating this equation gives

$$q(h) = h\left(\frac{q_*}{h_*} + 2\sqrt{gh_*} - 2\sqrt{gh}\right) = h\bar{u}_* + 2h\left(\sqrt{gh_*} - \sqrt{gh}\right). \tag{1.109}$$

Expressed in terms of velocity, this equation is equivalent to

$$\bar{u}(h) + 2\sqrt{gh} = \bar{u}_* + 2\sqrt{gh_*},$$

and we then deduce that the quantity

$$r_1 = \bar{u} + 2\sqrt{gh} \tag{1.110}$$

is constant along the 1-rarefaction wave. This is the *1-Riemann invariant.*

Similarly, we can calculate the 2-rarefaction wave associated with $\boldsymbol{w}_2$, and we obtain

$$q(h) = h\left(\frac{q_*}{h_*} + 2\sqrt{gh_*} - 2\sqrt{gh}\right) = h\bar{u}_* + 2h\left(-\sqrt{gh_*} + \sqrt{gh}\right). \tag{1.111}$$

The 2-Riemann invariant is

$$r_2 = \bar{u} - 2\sqrt{gh}. \tag{1.112}$$

Notes:

1. The equations used for characterising the simple waves are not sufficient to determine the structure of the rarefaction wave. Indeed, the k-rarefaction wave is defined from Eq. (1.66) as $\boldsymbol{q}' = \alpha\boldsymbol{w}_k$ where $\alpha = (\nabla\lambda_k \cdot \boldsymbol{w}_k)^{-1}$, but in the calculations just above, we integrated $\boldsymbol{q}' = \alpha\boldsymbol{w}_k$ with $\alpha = 1$. This simplified the calculation of r_k but did not allow us to derive the rarefaction wave's exact structure.

2. The 1-Riemann variable $s_1 = \bar{u} - 2\sqrt{gh}$ used in the characteristic form (1.76) of the Saint-Venant equations is the 2-Riemann invariant $r_2 = \bar{u} - 2\sqrt{gh}$ (as indicated in § 1.3.1). Similarly, the 2-Riemann variable s_2 matches the 1-Riemann invariant.

Rarefaction wave equation

We use the formal definition (1.66) of the rarefaction wave given in § 1.3.3

$$\boldsymbol{q}'(\xi) = \begin{pmatrix} h' \\ q' \end{pmatrix} = \frac{\boldsymbol{w}_1}{\nabla\lambda_1 \cdot \boldsymbol{w}_1} = -\frac{2}{3}\sqrt{\frac{h}{g}}\begin{pmatrix} 1 \\ \bar{u} - c \end{pmatrix} \tag{1.113}$$

since

$$\nabla\lambda_k \cdot \boldsymbol{w}_1 = \begin{pmatrix} -\bar{u}/h - c/2 \\ 1/h \end{pmatrix} \cdot \begin{pmatrix} 1 \\ \bar{u} - c \end{pmatrix} = -\frac{3}{2}\sqrt{\frac{g}{h}} = -\frac{3}{2}\frac{c}{h}.$$

As $h > 0$, the scalar product $\nabla\lambda_k \cdot \boldsymbol{w}_1$ is always negative, and the 1-characteristic is genuinely nonlinear field. The first equation

$$h' = -\frac{2}{3}\sqrt{\frac{h}{g}} \Rightarrow h(\xi) = \frac{(3a - 2\xi)^2}{36g}, \tag{1.114}$$

where a is a constant of integration. Since the boundary condition imposes $h = h_l$ at $\xi = \lambda_l = u_l - \sqrt{gh_l}$, we have

$$3a = 2u_l + 4\sqrt{gh_l},$$

and similarly for the right boundary condition, we have

$$3a = 2u_r + 4\sqrt{gh_r},$$

which is possible only if

$$u_r + 2\sqrt{gh_r} = u_l + 2\sqrt{gh_l}, \tag{1.115}$$

or in other words, if the Riemann invariant r_1 is constant. Using this result in Eq. (1.114), we finally obtain:

$$h(\xi) = \frac{(u_r + 2\sqrt{gh_r} - 2\xi)^2}{9g} = \frac{(u_l + 2\sqrt{gh_l} - 2\xi)^2}{9g} \tag{1.116}$$

The second equation is

$$q' = -\frac{2}{3}(\bar{u} - c)\frac{h}{c}.$$

Formally, we could solve this equation by substituting h with the solution $h(\xi)$ found previously. This leads to intricate calculations. Note first that q can be determined directly without solving any equation by recalling that the 1-Riemann invariant is constant, and thus $q/h + 2\sqrt{gh}$ is a constant. Otherwise, we can integrate the equation above by making a change of variable $q(\xi) \to q(h)$ and making use of Eq. (1.114):

$$q' = \frac{\mathrm{d}q}{\mathrm{d}\xi} = \frac{\mathrm{d}q}{\mathrm{d}h}\frac{\mathrm{d}h}{\mathrm{d}\xi} = -\frac{2}{3}\sqrt{\frac{h}{g}}\frac{\mathrm{d}q}{\mathrm{d}h},$$

and thus

$$\frac{\mathrm{d}q}{\mathrm{d}h} = \frac{q}{h} - \sqrt{gh} \Rightarrow q(h) = h(a - 2\sqrt{gh}),$$

where a is a constant of integration. The boundary conditions impose that

$$a = u_l + 2\sqrt{gh_l} = u_r + 2\sqrt{gh_r}.$$

This leads to the desired result.

Graphical representation

Figure 1.9(a) shows the phase portrait of Eq. (1.108) and one integral curve (1.109) (in red). The black curves show the vector field $\boldsymbol{w}_1$, and the red curve shows one 1-rarefaction wave (integral curve of $\boldsymbol{w}_1$) associated with the initial condition $q_*(h_*) = 1$ for $h_* = 1$. The 1-rarefaction wave is the contour line (or level curve) of r_1: we have $r_1 = \bar{u}_* + 2\sqrt{gh_*} = 3$ (when we take $g = 1$ m·s^{-2}).

Figure 1.10(a) shows one integral curve (related to $r_1 = 2$) and the contour lines of the eigenvalues. As the field is genuinely nonlinear, the characteristic speed λ_1 varies monotonically along any integral curve. If we take the decreasing part of the integral path (red curve), we see that the eigenvalue λ_1 decreases when one goes from left to right. This has important consequences to determine which part of the rarefaction waves is physically admissible. Let us assume that we start from a left state $\boldsymbol{q}_l = (h_l, q_l)$ and we seek where a point $\boldsymbol{q}_* = (h_*, q_*)$ connected to the left state lies on the rarefaction wave. As $\lambda_1(\boldsymbol{q}_l) < \lambda_1(\boldsymbol{q}_*)$, the physically admissible part of the integral path is the one along which λ_1 increases.

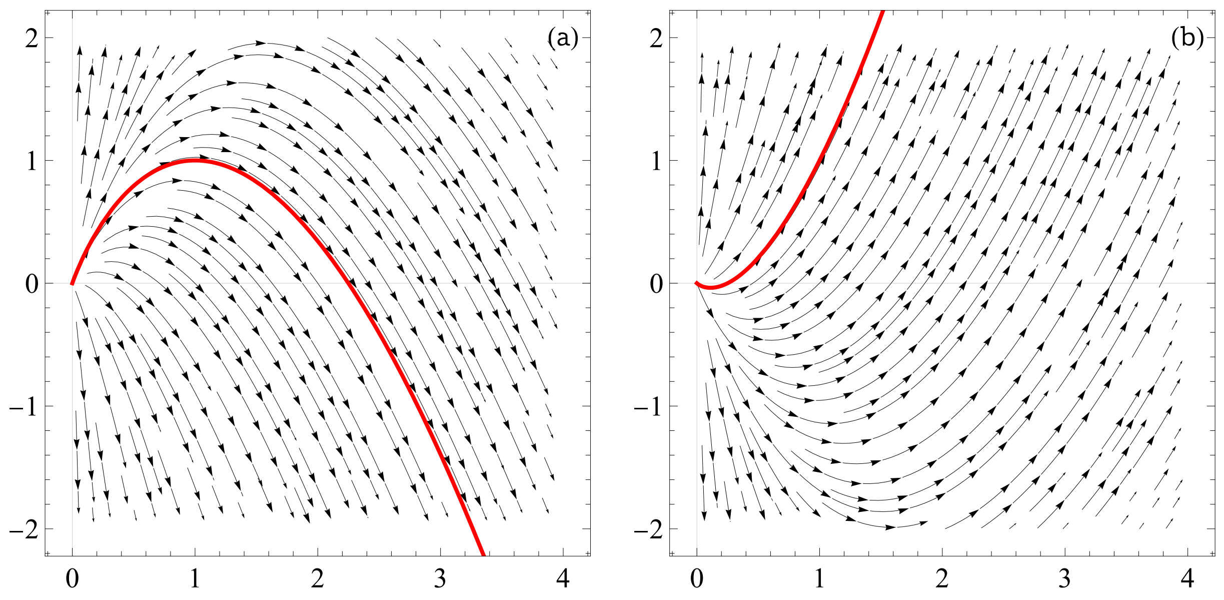


Figure 1.9 Phase portrait (a) associated with $\boldsymbol{w}_1$ ($q'(h) = q/h - \sqrt{gh}$). (b) associated with $\boldsymbol{w}_2$ ($q'(h) = q/h + \sqrt{gh}$). The red curves show the integral curve (a) $q(h) = h\bar{u}_* + 2h\left(\sqrt{gh_*} - \sqrt{gh}\right)$ (1-rarefaction wave) with $h_* = 1$ and $q_* = 1$, and (b) $h\bar{u}_* + 2h\left(\sqrt{gh_*} + \sqrt{gh}\right)$ (2-rarefaction wave). We take $g = 1$ m·s^{-2}.

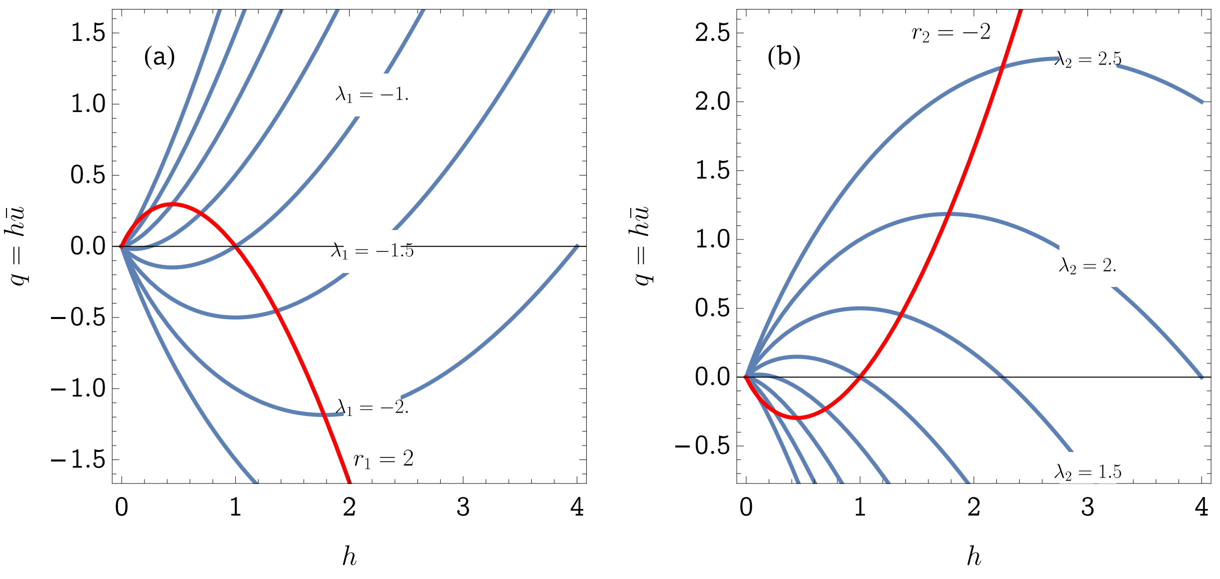


Figure 1.10 (a) $\lambda = -2.5$, 1, 0.5. (b) $\lambda = -1$, 2.5, 0.5. The red curve shows one 1-rarefaction and 2-rarefaction waves (here $r_1 = 2$ and $r_2 = -2$) whereas the blue curves shows the contour lines for the eigenvalues λ_1 and λ_2.

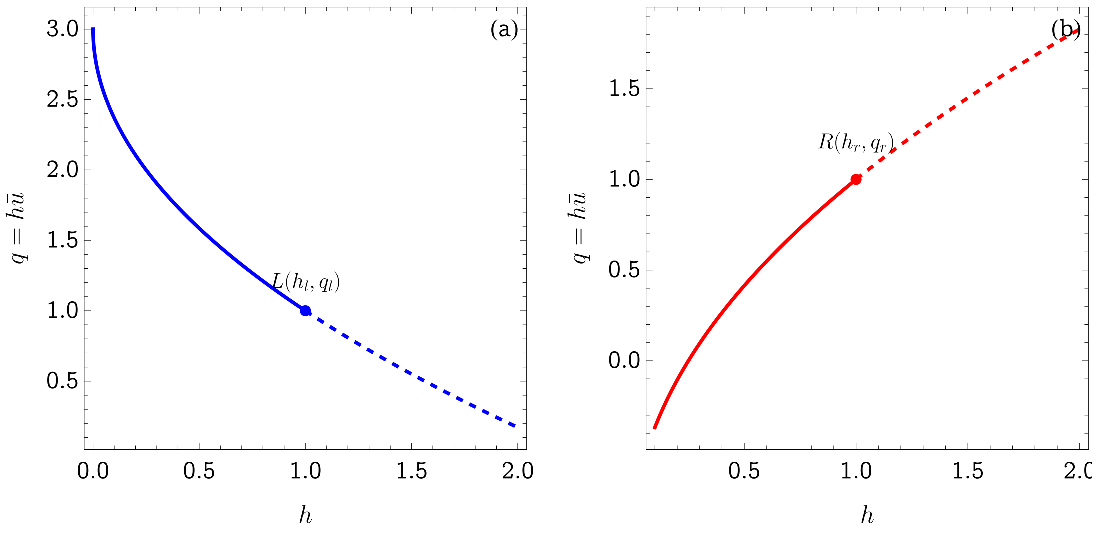


Figure 1.11 (a) 1-rarefaction wave issuing from the left state. (a) 2-rarefaction wave connecting to the right state. The solid lines show the physically admissible states, whereas the dashed lines show the part of the rarefaction wave that is not physically admissible

1.4.5 General solution to the Riemann problem

Governing equations

We are now armed to work out the general solution to the Riemann problem for the Saint-Venant equations

$$\frac{\partial}{\partial t}\boldsymbol{q} + \frac{\partial}{\partial x}\boldsymbol{f}(\boldsymbol{q}) = 0 \Leftrightarrow \frac{\partial}{\partial t}\boldsymbol{q} + \boldsymbol{A}(\boldsymbol{q}) \cdot \frac{\partial}{\partial x}\boldsymbol{q} = 0, \tag{1.117}$$

where the unknown vector and the Jacobian matrix $\boldsymbol{A}$ are

$$\boldsymbol{q} = \left(\begin{array}{c} h \\ q \end{array}\right) \text{ and } \boldsymbol{A} = \boldsymbol{f}' = \left(\begin{array}{cc} 0 & 1 \\ -q^2/h^2 + gh & 2q/h \end{array}\right), \tag{1.118}$$

and the flux function $\boldsymbol{f}$ is:

$$\boldsymbol{f} = \left(\begin{array}{c} q \\ \dfrac{q^2}{h} + \dfrac{1}{2}gh^2 \end{array}\right). \tag{1.119}$$

The initial condition is

$$\boldsymbol{q}(x, 0) = \left\{ \begin{array}{l} \boldsymbol{q}_l \text{ for } x < 0, \\ \boldsymbol{q}_r \text{ for } x > 0. \end{array} \right. \tag{1.120}$$

Unless the left and right states can be connected by a single rarefaction or shock curve, the solution usually involves a combination of two waves, usually a rarefaction and a shock waves, but sometimes, depending on the particular values of $\boldsymbol{q}_l$ and $\boldsymbol{q}_r$, two shocks or two rarefaction waves. One of these waves connects the left state $\boldsymbol{q}_l$ with an intermediate state $\boldsymbol{q}_*$, while the other connects $\boldsymbol{q}_*$ to $\boldsymbol{q}_r$. Note that possibly, the intermediate state corresponds to the dry state $h = 0$. The problem thus lies in the calculation of this intermediate state $\boldsymbol{q}_*$ and the type of connecting waves.

Computational strategy

We have seen that

- Shocks have to satisfy the Lax entropy condition (1.56)–(1.58), which imposes that a 1-shock wave connecting the left and right states has to satisfy

$$\lambda_1(\boldsymbol{q}_r) < \dot{\sigma} < \lambda_1(\boldsymbol{q}_l), \tag{1.121}$$

$$\dot{\sigma} < \lambda_2(\boldsymbol{q}_l) \text{ and } \dot{\sigma} < \lambda_2(\boldsymbol{q}_r). \tag{1.122}$$

  The Lax condition for a 2-shock wave is

$$\lambda_2(\boldsymbol{q}_r) < \dot{\sigma} < \lambda_2(\boldsymbol{q}_l), \tag{1.123}$$

$$\dot{\sigma} > \lambda_1(\boldsymbol{q}_l) \text{ and } \dot{\sigma} > \lambda_1(\boldsymbol{q}_r). \tag{1.124}$$

- If there is an intermediate state $\boldsymbol{q}_*$, the wave from $\boldsymbol{q}_l$ to $\boldsymbol{q}_*$ move more slowly than the wave from $\boldsymbol{q}_*$ to $\boldsymbol{q}_r$. As a consequence, the wave issuing from the left state $\boldsymbol{q}_l$ is necessarily a 1-wave, whereas the wave issuing from the intermediate state $\boldsymbol{q}_*$ is a 2-wave.

Let us now use this information to calculate the features of the 1-shock wave connecting $\boldsymbol{q}_l$ to $\boldsymbol{q}_*$. Equation (1.121) imposes

$$\bar{u}_* - \sqrt{gh_*} < \dot{\sigma} < \bar{u}_l - \sqrt{gh_l},$$

and recalling that the shock velocity can also be expressed [see Eq. (1.93)] in the form

$$\dot{\sigma} = u_* - \sqrt{g h_l \frac{h_* + h_l}{2h_*}} = u_l - \sqrt{g h_* \frac{h_* + h_l}{2h_l}}.$$

we find that the Lax entropy condition implies

$$h_* > h_l \tag{1.125}$$

for the 1-shock wave. By using the same procedure for the 2-shock wave, we deduce that the Lax entropy condition for this wave is

$$h_r > h_* \tag{1.126}$$

These conditions are consistent with the criteria established in § 1.4.4 (see also Fig. 1.11).

We define two functions ϕ_l and ϕ_r:

$$\phi_l(h) = \begin{cases} u_l + 2(\sqrt{g h_l} - \sqrt{g h}) & \text{if } h < h_l \\ u_l - (h - h_l)\sqrt{\dfrac{g}{2}\left(\dfrac{1}{h_l} + \dfrac{1}{h}\right)} & \text{if } h > h_l \end{cases} \tag{1.127}$$

and

$$\phi_r(h) = \begin{cases} u_r - 2(\sqrt{g h_r} - \sqrt{g h}) & \text{if } h < h_r \\ u_r + (h - h_r)\sqrt{\dfrac{g}{2}\left(\dfrac{1}{h_r} + \dfrac{1}{h}\right)} & \text{if } h > h_r \end{cases} \tag{1.128}$$

The intermediate state is found by solving

$$\phi_l(h_*) = \phi_r(h_*). \tag{1.129}$$

Given that we know the depth h, the function ϕ_l yields the value of the velocity u such that the state (h, hu) is connected to the left state $(h_l, h_l u_l)$ by a 1-wave (rarefaction or shock), whereas ϕ_r provides the value of the velocity u such that the state (h, hu) is connected to the right state $(h_r, h_r u_r)$ by a 2-wave (rarefaction or shock).

Example

Let us consider the following example (wet dam break) with $\boldsymbol{q}_l = (4, 0)$ and $\boldsymbol{q}_r = (1, 0)$. Figure 1.12(a) shows the initial depth profile, and Fig. 1.12(b) shows the characteristic lines emanating from the initial condition. The 1-characteristic curves spread out, and thus there is apparently an empty wedge in the left quadrant. This wedge will be occupied by a rarefaction wave (for this reason, we will refer to it as the *rarefaction fan*). The 2-characteristic curves cross each other, which indicates the formation of a shock wave in the right quadrant. From this inspection of the characteristic curves, we deduce that the solution involves a 1-rarefaction wave and 2-shock wave.

We can determine the intermediate state $\boldsymbol{q}_* = (h_*, h_* \bar{u}_*)$ that is connected to the left and right states. We apply the computational strategy outlined above to calculate this state. We find $h_* = 2.207$ m and $\bar{u}_* = 1.028$ m/s. Figure 1.13 confirms that the 1-wave connecting $\boldsymbol{q}_l$ to $\boldsymbol{q}_*$ is a rarefaction wave, whereas the 2-wave is a shock wave.

We can now plot the characteristic curves. Figure 1.14 shows the shock wave (black line)

$$x = \dot{\sigma} t \text{ with } \dot{\sigma} = \frac{h_r \bar{u}_r - h_* \bar{u}_*}{h_r - h_*} = 1,88 \text{ m/s},$$

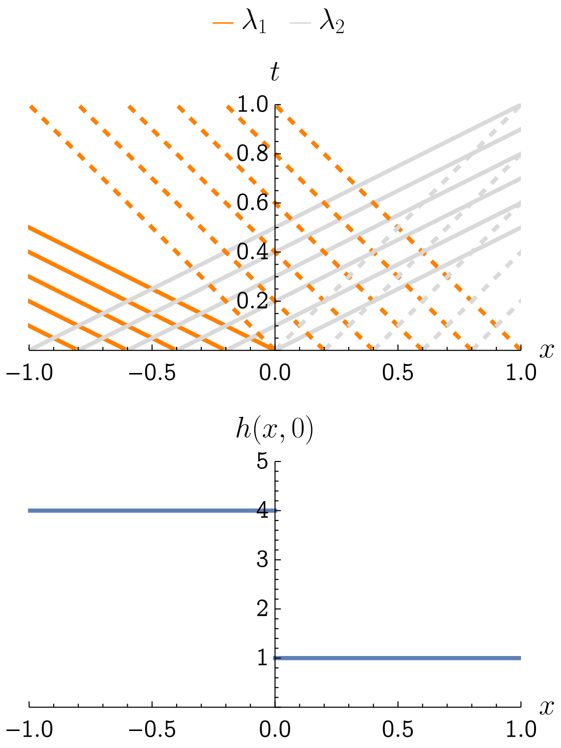


Figure 1.12 We consider the following Riemann problem $\boldsymbol{q}_l = (4,0)$ and $\boldsymbol{q}_r = (1,0)$. (a) Flow depth profile. (b) The solid lines refer to 1- and 2-characteristics related to the left state, while the dashed lines refer to the right state. We use the following colour code: orange for the characteristic curves $x = x_0 + \lambda_1 t$, and grey for the characteristic curves $x = x_0 + \lambda_2 t$.

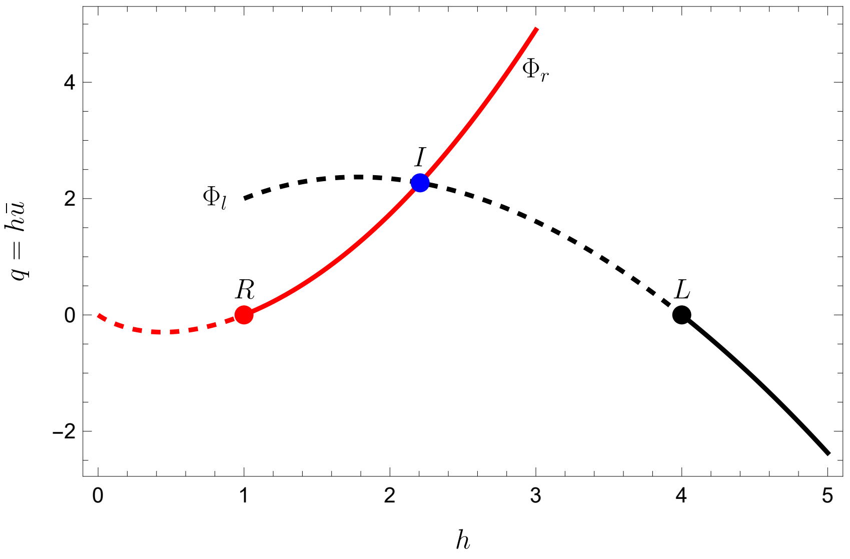


Figure 1.13 Example of solution with $\boldsymbol{q}_l = (4,0)$ and $\boldsymbol{q}_r = (1,0)$. The solid lines show shock waves, while the dashed lines refer to the rarefaction waves.

and the rarefaction fan (orange lines) delineated by the two characteristic curves

$$x = \lambda_{1l} t \text{ and } x = \lambda_{1*} t,$$

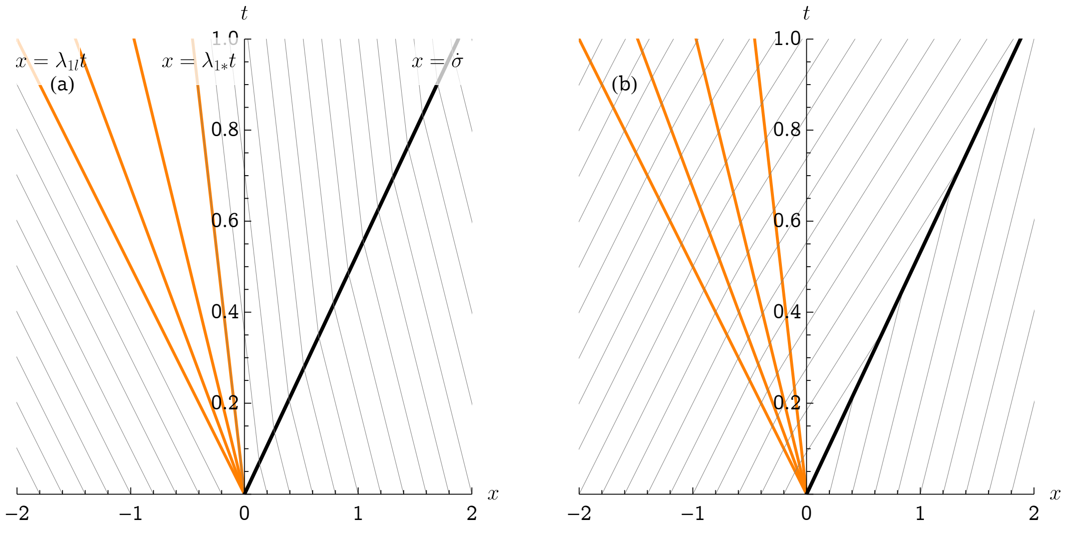


Figure 1.14 Solution to the dambreak problem $\boldsymbol{q}_l=(4,0)$ and $\boldsymbol{q}_r=(1,0)$. (a) 1-characteristic curves. (b) 2-characteristic curves. The orange lines show the rarefaction fan that separates the 1-characteristic curves issuing from the left and right states. The solid black line is the shock curve connecting the intermediate and right states.

where $\lambda_{1l}=\bar u_l-\sqrt{gh_l}=-\sqrt{gh_l}$ and $\lambda_{1*}=\bar u_*-\sqrt{gh_*}$. Figure 1.14(a) shows the 1-characteristic curves. On the left of the rarefaction fan, they take the form

$$x=x_0+\lambda_{1l}t,$$

where x_0 is any position on the x-axis. On the right, they undergo a slight of direction when crossing the shock curve. On the right of the shock curve, the 1-characteristic curves are

$$x=x_0+\lambda_{1r}t,$$

while on the shock curve's right, they have the form

$$x=x_1+\lambda_{1*}t,$$

where $x_1(x_0)$ is set by the continuity with the 1-characteristic curves arriving from the region downstream of the shock curve. Indeed, the 1-characteristic curve crosses the 2-shock curve at time

$$\dot\sigma t=x_0+\lambda_{1r}t\Rightarrow t=\frac{x_0}{\dot\sigma-\lambda_{1r}},$$

and thus $x_1(x_0)=\dot\sigma x_0/(\dot\sigma-\lambda_{1r})$.

Figure 1.14(a) shows the 2-characteristic curves. On the left of the rarefaction wave, these curves take the form

$$x=x_0+\lambda_{2l}t.$$

When these curves move through the rarefaction wave, they are no longer straight lines. Their governing equation is

$$\frac{\mathrm{d}x}{\mathrm{d}t}=\bar u+\sqrt{gh}. \tag{1.130}$$

In the 1-rarefaction fan, the depth and velocity are determined to within a constant of integration. First, the Riemann invariant r_1 is constant in the 1-rarefaction fan, and the constant is fixed by the boundary condition on the left:

$$r_1=\bar u+2\sqrt{gh}=2c_l, \tag{1.131}$$

where $c_l = \sqrt{gh_l}$. Second, the 1-characteristic curves are straight lignes pivoting around the axis origin

$$x = \lambda_1 t = (\bar{u} - \sqrt{gh})t. \tag{1.132}$$

Solving Eqs. (1.131)–(1.132) for $\bar{u}$ and h gives

$$\bar{u} = \frac{2}{3}\left(c_l + \frac{x}{t}\right) \text{ and } h = \frac{1}{9g}\left(2c_l - \frac{x}{t}\right)^2 \tag{1.133}$$

Substituting Eq. (1.133) into Eq. (1.130) and integrating

$$x = 2c_l t + bt^{1/3}, \tag{1.134}$$

where b is a constant that will be determined by imposing the continuity between 2-characteristic curves on both sides of the boundary $x = \lambda_{1l}t$. On the right of the shock wave, the 2-characteristic curves take the form

$$x = x_0 + \lambda_{2r}t.$$

Figure 1.15 shows the flow-depth and velocity profiles at time $t = 0.5$ s. The rarefaction and shock waves are readily identifiable on the flow depth profile.

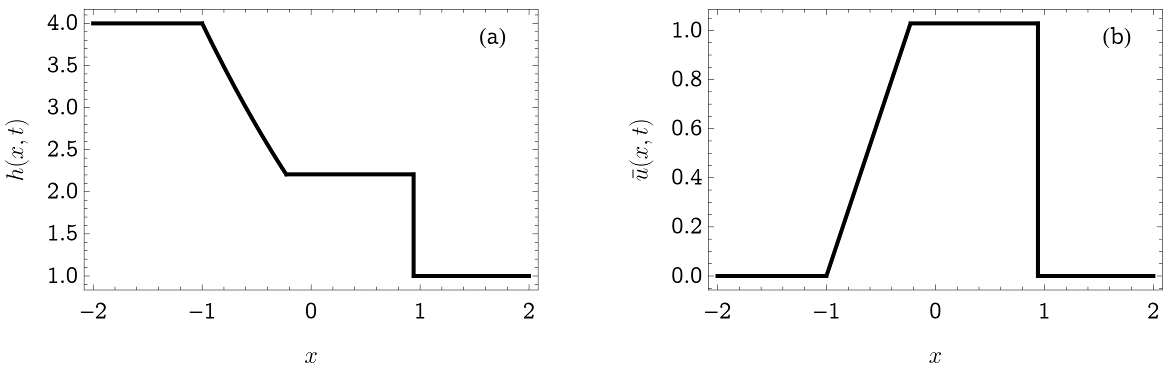


Figure 1.15 Solution to the dambreak problem $\boldsymbol{q}_l = (4, 0)$ and $\boldsymbol{q}_r = (1, 0)$ at time $t = 0.5$ s (we set g to 1 m/s^2). (a) Flow depth profile. (b) Velocity profile.

CHAPTER 2 Finite volume methods

2.1 General formulation

2.1.1 *Finite-volume averaging*

Let us consider a hyperbolic equation in one space dimension and in a conservative form

$$\frac{\partial}{\partial t}\boldsymbol{q} + \frac{\partial}{\partial x}\boldsymbol{f}(\boldsymbol{q}) = 0, \tag{2.1}$$

where $\boldsymbol{q}$ is a vector with m components representing the unknowns and $\boldsymbol{f}$ is the flux function. We consider a uniform grid over an interval $[a, b]$ divided into N identical cells of size $\Delta x = (b - a)/N$. The cell centers are denoted by $x_{i-1/2}$ (see Fig. 2.1):

$$x_i = a + \left(i - \frac{1}{2}\right)\Delta x.$$

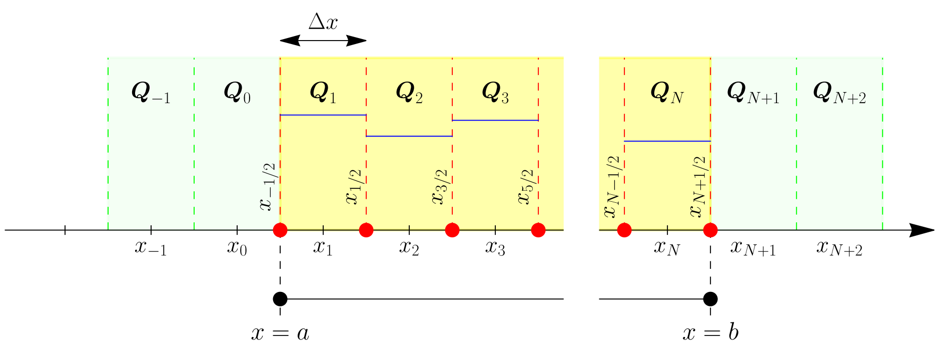


Figure 2.1 Computational grid. The yellow-coloured area represents the computational domain $[a, b]$, while the green-coloured area show the extended domain for handling the boundary conditions. The computational domain can be extended by including *ghost cells* (see § 2.10).

We define the cell $C_i = [x_{i-1/2}, x_{i+1/2})$, centred around the cell x_i and whose bounds (or interfaces) are $x_{i\pm1/2}$ (see Fig. 2.2). Note that:

$$x_{-1/2} = a \text{ and } x_{N+1/2} = b.$$

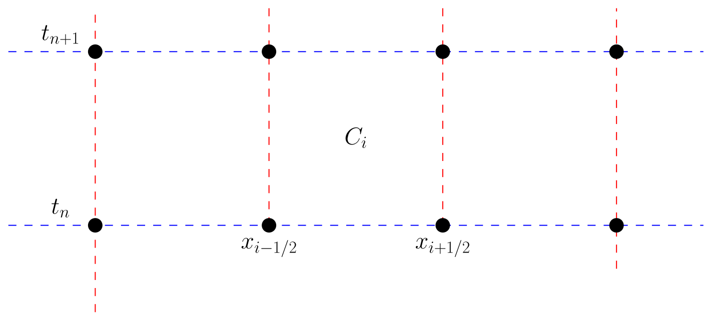


Figure 2.2 Computational domain and grid in the $x-t$ plane.

The time step is denoted by $\Delta t = t_{n+1} - t_n$. The computational domain is $[a,b]$, but we will see that we will need to extend it to handle boundary conditions (see § 2.10).

We integrate Eq. (2.1) over the cell C_i:

$$\int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial \boldsymbol{q}}{\partial t} \mathrm{d}x + [\boldsymbol{f}(\boldsymbol{q})]_{x_{i-1/2}}^{x_{i+1/2}} = 0. \tag{2.2}$$

Integrating this equation again over $(t_n, t_{n+1}]$ gives:

$$\int_{x_{i-1/2}}^{x_{i+1/2}} (\boldsymbol{q}(x,t_{n+1}) - \boldsymbol{q}(x,t_n))\mathrm{d}x + \int_{t_n}^{t_{n+1}} [\boldsymbol{f}(\boldsymbol{q})]_{x_{i-1/2}}^{x_{i+1/2}} \mathrm{d}t = 0. \tag{2.3}$$

We define the cell-averaged value of $\boldsymbol{q}$ at time t_n:

$$\boldsymbol{Q}_i^n = \frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} \boldsymbol{q}(x,t_n)\mathrm{d}x, \tag{2.4}$$

and a time-averaged flux

$$\boldsymbol{F}_{i\pm1/2}^n = \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} \boldsymbol{f}(\boldsymbol{q}(x_{i\pm1/2},t))\mathrm{d}t. \tag{2.5}$$

We can develop an explicit time-marching algorithm by rearranging Eq. (2.3) and introducing the time- and grid-averaged variables

$$\boldsymbol{Q}_i^{n+1} = \boldsymbol{Q}_i^n - \frac{\Delta t}{\Delta x}\left(\boldsymbol{F}_{i+1/2}^n - \boldsymbol{F}_{i-1/2}^n\right). \tag{2.6}$$

2.1.2 Upwind method

Let us give an application example of (2.6) with the upwind method. The idea behind this method is that for hyperbolic problems, information propagates in different directions and at different speeds. If we know how these waves carrying information behave, then we can determine the flux $\boldsymbol{F}_{i\pm1/2}$. We start with the simples case

$$\frac{\partial q}{\partial t} + \bar{u}\frac{\partial q}{\partial x} = 0, \tag{2.7}$$

where $\bar{u} > 0$ is a constant. The time-averaged flux (2.5) is thus

$$F_{i-1/2}^n = \frac{1}{\Delta t} \int_{t_n}^{t_{n+1}} f(q(x_{i\pm1/2},t))\mathrm{d}t = \bar{u}Q_{i-1}^n,$$

since Q^n_{i-1} is constant along the characteristic $x_{i-1} + \bar{u}t$. As a consequence for the linear advection problem, the first-order upwind method built from (2.6) is

$$Q_i^{n+1} = Q_i^n - \frac{\bar{u}\Delta t}{\Delta x}\left(Q_i^n - Q_{i-1}^n\right), \tag{2.8}$$

which looks like the finite-difference discretisation of Eq. (2.7) (one-sided first-order approximation).

This equation can be interpreted in (at least) two different ways, one focusing on space-averaging, the other on time-averaging. First, let us assume that we have a piecewise constant approximation of the solution $q(x, t)$ to (2.7) at time t_n as illustrated by Fig. 2.3. At time $t_{n+1} = t_n + \Delta t$, the curve has been shifted by $\bar{u}\Delta t$. We now calculate the value Q_i^{n+1} by averaging the shifted function over the cell C_i:

$$Q_i^{n+1} = \frac{1}{\Delta x}\left((\Delta x - \bar{u}\Delta t)Q_i^n + \bar{u}\Delta t Q_{i-1}^n,\right) \tag{2.9}$$

which imposes $\Delta x - \bar{u}\Delta t > 0$ for the averaging to make sense. This contraint on Δt

$$0 \leq \frac{\bar{u}\Delta t}{\Delta x} \leq 1$$

is called the *Courant-Friedrichs-Lewy condition* (CFL). It can be checked that after rearrangement, Eq. (2.9) leads to the upwind method (2.8).

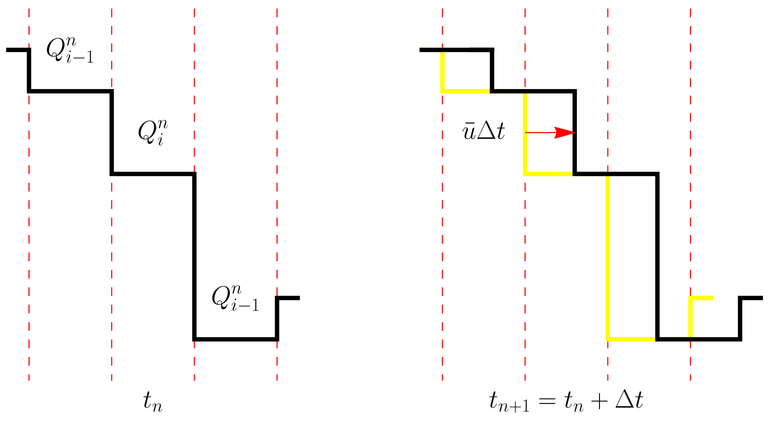


Figure 2.3 Advection of a piecewise continuous function. At time $t = t_n$, the function is piecewise constant, with value Q_i^n on cell C_i. At time t_{n+1}, this function has been shifted by $\bar{u}\Delta t$. The value Q_i^{n+1} is obtained by space-averaging the shifted function on cell C_i. The problem can be interpreted as the propagation of discontinuities originating from the cell boundaries $x_{i-1/2}$. These discontinuities form the wave $W_{i-1/2} = Q_i - Q_{i-1}$.

Another interpretation given by LeVeque (2002) is the wave-propagation viewpoint. If we define the jump

$$W_{i-1/2} = Q_i^n - Q_{i-1}^n,$$

then Eq. (2.8) can be rewritten

$$Q_i^{n+1} = Q_i^n - \frac{\bar{u}\Delta t}{\Delta x} W_{i-1/2}. \tag{2.10}$$

The interpretation is the following: the wave $W_{i-1/2}$ moves at velocity $\bar{u}$. When it moves through C_i, it changes the value of Q_i^n by $-W_{i-1/2}$ at each point it passes (see Fig. (2.3)). Over the time step Δt, the portion $\bar{u}\Delta t/\Delta x$ has been affected by the wave, and thus the cell average Q_i^{n+1} is given by Eq. (2.10). This interpretation will be used in the wave-propagation form of Godunov's method below.

Note that the upwind method takes the form (2.8) when $\bar{u} > 0$. When $\bar{u} < 0$, the upwind method leads to

$$Q_i^{n+1} = Q_i^n - \frac{\bar{u}\Delta t}{\Delta x}\left(Q_i^n - Q_{i-1}^n\right), \tag{2.11}$$

and the wave-propagation interpretation is

$$Q_i^{n+1} = Q_i^n - \frac{\bar{u}\Delta t}{\Delta x} W_{i+1/2}, \tag{2.12}$$

where $W_{i+1/2} = Q_{i+1}^n - Q_i^n$. It is possible to generalise the upwind method by posing

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\left(\bar{u}^+ W_{i-1/2} + \bar{u}^- W_{i+1/2}\right), \tag{2.13}$$

where

$$\bar{u}^+ = \max(0, \bar{u}) \text{ and } \bar{u}^- = \min(0, \bar{u}).$$

The flux approximation is

$$F_{i-1/2}^n = \frac{\bar{u}\Delta t}{\Delta x}\left(\bar{u}^- Q_i^n + \bar{u}^+ Q_{i-1}^n\right).$$

2.2 Godunov's method for linear systems

Godunov's method has been a major achievement in the field of hyperbolic equations, which has opened up the way to modern finite-volume techniques (Godunov, 1959; Toro, 2001; LeVeque, 2002; Toro & Garcia-Navarro, 2007; Guinot, 2010). It consists of three steps: reconstructing, evolving, and averaging.

1. *Reconstruction.* We assume that we can approximate the solution $\boldsymbol{q}(x,t)$ by a piecewise constant function $\tilde{\boldsymbol{q}}_i^n(x, t_n) = \boldsymbol{Q}_i^n$ for $x \in C_i = (x_{i-1/2}, x_{i+1/2})$. Note that to second order, we have

$$\begin{aligned}
\boldsymbol{Q}_i^n &= \frac{1}{\Delta x}\int_{x_{i-1/2}}^{x_{i+1/2}} \tilde{\boldsymbol{q}}(x, t_n)\mathrm{d}x,\\
&= \frac{1}{\Delta x}\int_{x_i-\Delta x/2}^{x_i+\Delta x/2}\left(\tilde{\boldsymbol{q}}(x_i, t_n) + (x - x_i)\frac{\partial}{\partial x}\boldsymbol{q}(x_i, t_n) + \frac{(x-x_i)^2}{2}\frac{\partial^2}{\partial x^2}\boldsymbol{q}(x_i, t_n)\right)\mathrm{d}x,\\
&= \boldsymbol{q}(x_i, t_n) + \frac{\Delta x^2}{24}\frac{\partial^2}{\partial x^2}\boldsymbol{q}(x_i, t_n).
\end{aligned}$$

   Godunov's method is a first-order accurate scheme. We can use higher-order reconstructions of the approximate the function $\boldsymbol{q}(x,t)$ (e.g., a piecewise linear function with a nonzero slope in each grid cell). See § 2.8.

2. *Evolution.* Using Eq. (2.6) or another method, we look at how the solution $\tilde{\boldsymbol{q}}_i^{n+1}$ has evolved from the previous state at time t_n. This step amounts to solving Riemann problems at each cell boundary $x_{i\pm1/2}$.

3. *Averaging.* We average this function over each grid cell

$$\boldsymbol{Q}_i^{n+1} = \frac{1}{\Delta x}\int_{x_{i-1/2}}^{x_{i+1/2}} \tilde{\boldsymbol{q}}(x, t_{n+1})\mathrm{d}x. \tag{2.14}$$

   We can piece together the Riemann solutions provided that the waves from two adjacent interfaces have not started to interact. This condition is usually met when the *Courant–Friedrichs–Lewy* (CFL) condition is satisfied (no wave passes through more than one grid cell within Δt):

$$\frac{c_{max}\Delta t}{\Delta x} \leq 1, \tag{2.15}$$

   where c_{max} represents the largest wave speed over the computational domain.

Godunov's method was initially used to solve the Euler equations in gas dynamics (Godunov, 1959, 1962). The flux $\boldsymbol{F}^n_{i+1/2}$ was determined from the exact solution to the Riemann problem for the Euler equations. Approximate Riemann solvers are used today because they are faster. Godunov's method is robust and stable when the CFL condition is met. When approximate solvers are used, this may not be the case, and thus special care has to be paid to robustness and stability. Moreover, Godunov's method tends to smear out solutions near discontinuities. By using limiters, approximate Riemann solvers deal more efficiently with discontinuities. They also build numerical solutions as linear combinations of travelling discontinuities—they do not use rarefaction waves, which are therefore approximated as discontinuities. Transonic waves[1] may need more care.

The crux in Godunov's method lies in the determination of the flux terms in Eq. (2.6). These fluxes are defined by Eq. (2.5):

$$\boldsymbol{F}^n_{i\pm1/2} = \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} \boldsymbol{f}(\boldsymbol{q}(x_{i\pm1/2}, t))\mathrm{d}t.$$

When we solve a succession of Riemann problems, the integrand f is usually constant because either the solution to the Riemann problem involves a rarefaction wave (in this case, the line $x = x_{i-1/2}$ is a characteristic curve) or a shock wave (in that case, the shock wave separates two constant-state domains). Therefore, in both cases, the value $\boldsymbol{q}$ is a constant that can be expressed as a function denoted by $\boldsymbol{q}^{\downarrow}$ by LeVeque (2002) of the left and right states:

$$\boldsymbol{q} = \boldsymbol{q}^{\downarrow}(\boldsymbol{Q}^n_{i-1}, \boldsymbol{Q}^n_i) \tag{2.16}$$

and thus we can define the value taken by $\boldsymbol{q}$ at the interface $x_{i-1/2}$ as

$$\boldsymbol{Q}^n_{i-1/2} = \boldsymbol{q}^{\downarrow}(\boldsymbol{Q}^n_{i-1}, \boldsymbol{Q}^n_i) \tag{2.17}$$

and the flux at the interface $x_{i-1/2}$:

$$\boldsymbol{F}^n_{i-1/2} = \boldsymbol{f}(\boldsymbol{q}^{\downarrow}(\boldsymbol{Q}^n_{i-1}, \boldsymbol{Q}^n_i)) = \boldsymbol{\mathcal{F}}(\boldsymbol{Q}^n_{i-1}, \boldsymbol{Q}^n_i) \tag{2.18}$$

To sum up, we can implement Godunov's method by following the three following steps:

1. Solve the Riemann problem at each interface $x_{i-1/2}$ and derive the value $\boldsymbol{q}^{\downarrow}(\boldsymbol{Q}^n_{i-1}, \boldsymbol{Q}^n_i)$.
2. Compute the flux $\boldsymbol{F}^n_{i-1/2}$ using Eq. (2.18).
3. Update the cell values $\boldsymbol{Q}^n_i$ by using the flux-differencing equation (2.6).

2.3 Wave decomposition for linear systems

2.3.1 Introductive example

In Clawpack, we will use a variant of Godunov's method based on the wave decomposition seen in Chapter 1 and close to the wave-propagation interpretation of the upwind method seen in § 2.1.2. There are other strategies such as *flux differencing* (Toro, 2001; LeVeque, 2002; Guinot, 2010). The advantage of wave decomposition over other approaches is that it can also be applied to non-conservative equations.

Let us illustrate how Clawpack proceeds with the flux estimation by considering a problem of dimension $m = 3$. Let us assume that we have three different eigenvalues such that $\lambda_1 < 0 < \lambda_2 < \lambda_3$. As shown by Fig. 2.4, from the node $x_{i-1/2}$ emerge three characteristics $x_{i-1/2} + \lambda_i t$, which will create three discontinuities in $\tilde{\boldsymbol{q}}$ at time t_{n+1}. Similarly, three characteristics are issued from $x_{i+1/2} + \lambda_i t$. The

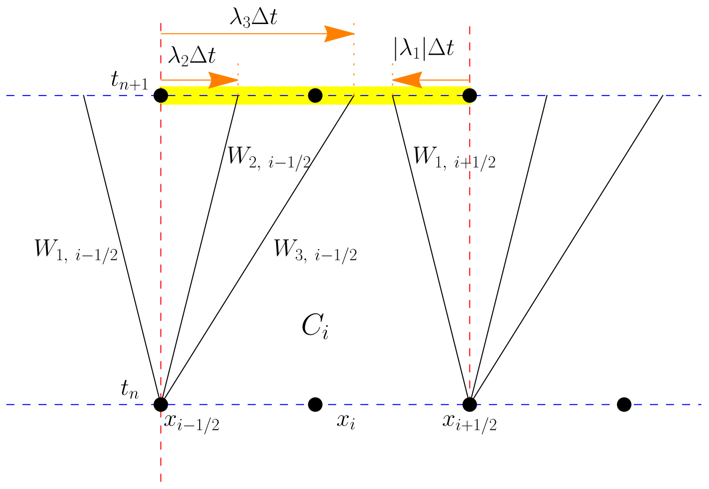


Figure 2.4 Wave structure for the nodes $x_{i-1/2}$ and $x_{i+1/2}$. At time t_n, we consider a series of Riemann problems with $\tilde{\boldsymbol{q}}_i^n(x, t_n) = \boldsymbol{Q}_i^n$. We can compute $\tilde{\boldsymbol{q}}(x, t)$ the solution at later times by considering the waves $\boldsymbol{W}_i$. The solution is given by Eq. (2.20). The next step is to average the value of $\tilde{\boldsymbol{q}}(x, t_{n+1})$ over the interval $[x_{i-1/2}, x_{i+1/2}]$ (coloured in yellow).

coloured segment in Fig. 2.4 is crossed by three waves: $x_{i-1/2} + \lambda_2 t$, $x_{i-1/2} + \lambda_3 t$, and $x_{i+1/2} + \lambda_1 t$. The other waves will not cause any change to this segment.

Recall that in Chapter 1, we learned from Eq. (1.31) that the initial jump in the Riemann problem at $x_{i-1/2}$ can be decomposed into three waves:

$$\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1} = \sum_{k=1}^{m} \boldsymbol{W}_{k,i-1/2}, \tag{2.19}$$

where $\boldsymbol{W}_{k,i-1/2}$ is collinear with the right eigenvectors: $\boldsymbol{W}_{k,i-1/2} = \alpha_{k,i-1/2}\boldsymbol{w}_{k,i-1/2}$ where

$$\boldsymbol{\alpha} = \boldsymbol{L} \cdot \Delta\boldsymbol{Q}_{i-1/2}$$

and

$$\Delta\boldsymbol{Q}_{i-1/2} = \boldsymbol{Q}_i - \boldsymbol{Q}_{i-1}.$$

As explained in § 1.1.6 for linear hyperbolic systems, the solution is given by Eqs. (1.35)–(1.36), which tells us that for the double Riemann problem considered here, we have

$$\tilde{\boldsymbol{q}} = \boldsymbol{Q}_i^n + \boldsymbol{W}_{1,i+1/2} - \boldsymbol{W}_{2,i-1/2} - \boldsymbol{W}_{3,i-1/2}. \tag{2.20}$$

Let us examine how these waves modify the value of $\boldsymbol{Q}_i$ at time t_{n+1}. If we consider the first wave, we can see that it will modify the value of $\boldsymbol{q}$ over a fraction of the grid cell $|\lambda_1|\Delta t/\Delta x$ by the amount

$$|\lambda_1|\frac{\Delta t}{\Delta x}\boldsymbol{W}_{1,i+1/2}$$

relative to the initial value $\boldsymbol{Q}_i$ (as $\lambda_1 < 0$, we have to take its absolute value). Similarly, for the 2- and 3-waves, the value of $\boldsymbol{q}$ is changed over a fraction of the grid cell $\lambda_2\Delta t/\Delta x$ and $\lambda_3\Delta t/\Delta x$, respectively, by the amount

$$-\lambda_2\frac{\Delta t}{\Delta x}\boldsymbol{W}_{2,i-1/2} - \lambda_3\frac{\Delta t}{\Delta x}\boldsymbol{W}_{3,i-1/2}.$$

[1]see Fig. 2.6 for a quick definition.

If we sum the contributions, we obtain the averaged value of $\tilde{\boldsymbol{q}}(x, t_{n+1})$ over the interval $[x_{i-1/2}, x_{i+1/2}]$:

$$\begin{aligned} \boldsymbol{Q}_i^{n+1} &= \boldsymbol{Q}_i^n + |\lambda_1| \frac{\Delta t}{\Delta x} \boldsymbol{W}_{1,i+1/2} - \lambda_2 \frac{\Delta t}{\Delta x} \boldsymbol{W}_{2,i-1/2} - \lambda_3 \frac{\Delta t}{\Delta x} \boldsymbol{W}_{3,i-1/2} && (2.21) \\ &= \boldsymbol{Q}_i^n - \frac{\Delta t}{\Delta x} \left(\lambda_2 \boldsymbol{W}_{2,i-1/2} + \lambda_3 \boldsymbol{W}_{3,i-1/2} + \lambda_1 \boldsymbol{W}_{1,i+1/2} \right). && (2.22) \end{aligned}$$

2.3.2 General formulation

The result obtained in § 2.3.1 can be readily generalised to arbitrary hyperbolic systems. Let us introduce the notation

$$\lambda^+ = \max(\lambda, 0) \text{ and } \lambda^- = \min(\lambda, 0). \tag{2.23}$$

The updated value $\boldsymbol{Q}_i^{n+1}$ is then

$$\boldsymbol{Q}_i^{n+1} = \boldsymbol{Q}_i^n - \frac{\Delta t}{\Delta x} \left(\sum_{k=1}^{m} \lambda_k^+ \boldsymbol{W}_{k,i-1/2} + \sum_{k=1}^{m} \lambda_k^- \boldsymbol{W}_{k,i+1/2 \cdot} \right), \tag{2.24}$$

The cell average depends on the right-going waves from $x_{i-1/2}$ and left-going waves from $x_{i+1/2}$.

LeVeque (2002) introduced a shorthand notation

$$\begin{aligned} \boldsymbol{\mathcal{A}}^+ \cdot \Delta \boldsymbol{Q}_{i-1/2} &= \sum_{k=1}^{m} \lambda_k^+ \boldsymbol{W}_{k,i-1/2}, && (2.25) \\ \boldsymbol{\mathcal{A}}^- \cdot \Delta \boldsymbol{Q}_{i+1/2} &= \sum_{k=1}^{m} \lambda_k^- \boldsymbol{W}_{k,i+1/2}, && (2.26) \end{aligned}$$

which are interpreted as *fluctuations*: $\boldsymbol{\mathcal{A}}^+ \cdot \Delta \boldsymbol{Q}_{i-1/2}$ represents the effect of all right-going waves from $x_{i-1/2}$ (where there is a discontinuity $\Delta \boldsymbol{Q}_{i-1/2} = \boldsymbol{Q}_i - \boldsymbol{Q}_{i-1}$) on the cell average at time t_{n+1}, whereas $\boldsymbol{\mathcal{A}}^- \cdot \Delta \boldsymbol{Q}_{i+1/2}$ represents the effect of all left-going waves from $x_{i+1/2}$. This formulation that holds for linear problems will be generalised to nonlinear problems.

For linear problems, we can relate the operators $\boldsymbol{\mathcal{A}}^+$ and $\boldsymbol{\mathcal{A}}^-$ to the matrix $\boldsymbol{A}$. We use the spectral matrix $\boldsymbol{\Lambda}$, namely the diagonal matrix whose entries are the eigenvalues of $\boldsymbol{A}$. Making use of Eq. (1.8)

$$\boldsymbol{A} = \boldsymbol{R} \cdot \boldsymbol{\Lambda} \cdot \boldsymbol{R}^{-1} = \boldsymbol{R} \cdot \boldsymbol{\Lambda} \cdot \boldsymbol{L}$$

where the right eigenvector matrix $\boldsymbol{R} = [\boldsymbol{w}_1, \ldots, \boldsymbol{w}_m]$ is a matrix whose columns are made of the right eigenvectors, and

$$\boldsymbol{\Lambda} = \boldsymbol{\Lambda}^- + \boldsymbol{\Lambda}^+ = \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \ldots & 0 & \lambda_m \end{bmatrix} = \begin{bmatrix} \lambda_1^- & 0 & \ldots & 0 \\ 0 & \lambda_2^- & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \ldots & 0 & \lambda_m^- \end{bmatrix} + \begin{bmatrix} \lambda_1^+ & 0 & \ldots & 0 \\ 0 & \lambda_2^+ & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \ldots & 0 & \lambda_m^+ \end{bmatrix},$$

we can define the negative and positive parts of $\boldsymbol{A}$:

$$\begin{aligned} \boldsymbol{A}^- &= \boldsymbol{R} \cdot \boldsymbol{\Lambda}^- \cdot \boldsymbol{L}, \\ \boldsymbol{A}^+ &= \boldsymbol{R} \cdot \boldsymbol{\Lambda}^+ \cdot \boldsymbol{L}. \end{aligned}$$

Note that $\boldsymbol{A} = \boldsymbol{A}^{-} + \boldsymbol{A}^{+}$. We can calculate $\boldsymbol{A}^{+} \cdot \Delta \boldsymbol{Q}_{i-1/2}$

$$\begin{aligned}
\boldsymbol{A}^{+} \cdot \Delta \boldsymbol{Q}_{i-1/2} &= \boldsymbol{R} \cdot \boldsymbol{\Lambda}^{+} \cdot \boldsymbol{L} \cdot \boldsymbol{Q}_{i-1/2},\\
&= \boldsymbol{R} \cdot \boldsymbol{\Lambda}^{+} \cdot \boldsymbol{\alpha}_{i-1/2},\\
&= \sum_{k=1}^{m} \alpha^{k}_{i-1/2} \lambda^{+}_{k} \boldsymbol{w}_{k},\\
&= \sum_{k=1}^{m} \lambda^{+}_{k} \boldsymbol{W}^{k}_{i-1/2},\\
&= \boldsymbol{\mathcal{A}}^{+} \cdot \Delta \boldsymbol{Q}_{i-1/2},
\end{aligned}$$

where we have defined the wave

$$\boldsymbol{W}^{k}_{i-1/2} = \alpha^{k}_{i-1/2} \boldsymbol{w}_{k}.$$

Similarly we have

$$\boldsymbol{\mathcal{A}}^{-} \cdot \Delta \boldsymbol{Q}_{i+1/2} = \boldsymbol{A}^{-} \cdot \Delta \boldsymbol{Q}_{i+1/2} = \sum_{k=1}^{m} \lambda^{-}_{k} \boldsymbol{W}^{k}_{i+1/2}.$$

For the linear advection problem, the fluctuations can be calculated by simply multiplying $\boldsymbol{A}^{\pm}$ with the jump $\Delta \boldsymbol{Q}_{i\pm 1/2}$. For the nonlinear case, the calculation is more complicated, but we will continue to use the operators $\boldsymbol{\mathcal{A}}^{\pm}$ to represent the effects of right- and left-going waves on the value of $\boldsymbol{Q}^{n+1}_{i}$.

2.3.3 Interface flux

Clawpack approach

Note that the at interface between two cells, the interface value can be written (see Eq. (1.38)):

$$\boldsymbol{Q}^{\downarrow}_{i-1/2} = \boldsymbol{q}^{\downarrow}(\boldsymbol{Q}_{i-1}, \boldsymbol{Q}_{i}) = \boldsymbol{Q}_{i-1} + \sum_{\lambda_k < 0} \boldsymbol{W}^{k}_{i-1/2}. \tag{2.27}$$

We then deduce that for a linear system, the flux at the interface is:

$$\boldsymbol{F}^{n}_{i-1/2} = \boldsymbol{f}(\boldsymbol{Q}^{\downarrow}_{i-1/2}) = \boldsymbol{A} \cdot \boldsymbol{Q}^{\downarrow}_{i-1/2} = \boldsymbol{A} \cdot \boldsymbol{Q}_{i-1} + \sum_{\lambda_k < 0} \boldsymbol{A} \cdot \boldsymbol{W}^{k}_{i-1/2}. \tag{2.28}$$

As $\boldsymbol{W}_{k}$ is an eigenvector of $\boldsymbol{A}$, we can rearrange the terms

$$\boldsymbol{F}^{n}_{i-1/2} = \boldsymbol{A} \cdot \boldsymbol{Q}_{i-1} + \sum_{k=1}^{m} \lambda^{-}_{k} \boldsymbol{W}^{k}_{i-1/2} = \boldsymbol{A} \cdot \boldsymbol{Q}_{i-1} + \boldsymbol{A}^{-} \cdot \Delta \boldsymbol{Q}_{i-1/2}. \tag{2.29}$$

Concurrently, we have

$$\boldsymbol{Q}^{\downarrow}_{i-1/2} = \boldsymbol{Q}_{i} - \sum_{\lambda_k > 0} \boldsymbol{W}^{k}_{i-1/2}, \tag{2.30}$$

and consequently

$$\boldsymbol{F}^{n}_{i-1/2} = \boldsymbol{A} \cdot \boldsymbol{Q}_{i} - \sum_{k=1}^{m} \lambda^{+}_{k} \boldsymbol{W}^{k}_{i-1/2} = \boldsymbol{A} \cdot \boldsymbol{Q}_{i} - \boldsymbol{A}^{+} \cdot \Delta \boldsymbol{Q}_{i-1/2}. \tag{2.31}$$

Similarly, at the other interface, we have

$$
\begin{aligned}
\boldsymbol{F}^n_{i+1/2} &= \boldsymbol{A}\cdot\boldsymbol{Q}_{i+1} - \sum_{k=1}^{m}\lambda_k^{+}\boldsymbol{W}^k_{i+1/2} = \boldsymbol{A}\cdot\boldsymbol{Q}_{i+1} - \boldsymbol{A}^{+}\cdot\Delta\boldsymbol{Q}_{i+1/2}.\\
&= \boldsymbol{A}\cdot\boldsymbol{Q}_{i} + \sum_{k=1}^{m}\lambda_k^{-}\boldsymbol{W}^k_{i+1/2} = \boldsymbol{A}\cdot\boldsymbol{Q}_{i} + \boldsymbol{A}^{-}\cdot\Delta\boldsymbol{Q}_{i+1/2}.
\end{aligned}
$$

This expression will be generalised to nonlinear systems (which, once linearised, involve only shock waves), for which we will assume that

$$
\boldsymbol{F}^n_{i-1/2} = \boldsymbol{f}(\boldsymbol{Q}_{i-1}) + \boldsymbol{\mathcal{A}}^{-}\Delta\boldsymbol{Q}_{i-1/2}. \tag{2.32}
$$

or equivalently:

$$
\boldsymbol{F}^n_{i-1/2} = \boldsymbol{f}(\boldsymbol{Q}_{i}) - \boldsymbol{\mathcal{A}}^{+}\Delta\boldsymbol{Q}_{i-1/2}. \tag{2.33}
$$

Alternative: Harten–Lax–van Leer's approach

Can we proceed differently for nonlinear systems? For a nonlinear problem, the theoretical expression of the flux is more complicated, but there are approximate methods, such as the Harten–Lax–van Leer (HHL) solver, that can be used to estimate the flux (Harten *et al.*, 1983; Toro, 2019). In the HLL approach, we focus on the lowest and highest wave velocities λ_1 and λ_m. If $\lambda_1 < \lambda_m < 0$, then $\boldsymbol{Q}_{i-1/2} = \boldsymbol{Q}_i$. If $0 < \lambda_1 < \lambda_m$, then $\boldsymbol{Q}_{i-1/2} = \boldsymbol{Q}_{i-1}$. The only unknown case is $\lambda_1 < 0$ and $\lambda_m > 0$. Let us define X_L such that $x_L = x_{i-1/2} + \lambda_1\Delta t$ and x_R such that $x_R = x_{i-1/2} + \lambda_m\Delta t$ (see Fig. 2.5).

Integrating the hyperbolic equation (2.1) over $[x_L, x_{i-1/2}] \times [t_n, t_{n+1}]$ gives

$$
\int_{x_L}^{x_{i-1/2}} \boldsymbol{q}(x,\ t_{n+1})\mathrm{d}x = \int_{x_L}^{x_{i-1/2}} \boldsymbol{q}(x,\ t_n)\mathrm{d}x + \int_{t_n}^{t_{n+1}} \boldsymbol{f}(\boldsymbol{q}(x_L,\ t))\mathrm{d}t - \int_{t_n}^{t_{n+1}} \boldsymbol{f}(\boldsymbol{q}(x_{i-1/2},\ t))\mathrm{d}t,
$$

and if $\Delta x = x_{i-1/2} - x_{i-3/2}$ is chosen such that it lies in the domain controlled by the initial conditions $\boldsymbol{Q}_{i-1}$ or $\boldsymbol{Q}_i$ as imposed by the CLF condition (in other words, the characteristics originating from $x_{i-1/2}$ and $x_{i-3/2}$ do not cross in the integrating domain $[x_{i-3/2}, x_{i-1/2}] \times [t_n, t_{n+1}]$), then we can rearrange the terms

$$
\int_{x_L}^{x_{i-1/2}} \boldsymbol{q}(x,\ t_{n+1})\mathrm{d}x = (x_{i-1/2} - x_L)\boldsymbol{Q}_{i-1} + \Delta t\boldsymbol{f}(\boldsymbol{Q}_{i-1}) - \Delta t\boldsymbol{F}_{i-1/2}.
$$

This gives us the relation:

$$
\boldsymbol{F}_{i-1/2} = \boldsymbol{f}(\boldsymbol{Q}_{i-1}) - \lambda_1\boldsymbol{Q}_{i-1} - \frac{1}{\Delta t}\int_{-\Delta x}^{0} \boldsymbol{q}(x,\ t_{n+1})\mathrm{d}x. \tag{2.34}
$$

We now need to evaluate the last term implying $\boldsymbol{q}(x,\ t_{n+1})$ that is unknown. For this purpose, we redo the calculation above, but by integrating over $[x_L,\ x_R]$

$$
\int_{x_L}^{x_R} \boldsymbol{q}(x,\ t_{n+1})\mathrm{d}x = \int_{x_L}^{x_{i-1/2}} \boldsymbol{q}(x,\ t_n)\mathrm{d}x + \int_{t_n}^{t_{n+1}} \boldsymbol{f}(\boldsymbol{q}(x_L,\ t))\mathrm{d}t - \int_{t_n}^{t_{n+1}} \boldsymbol{f}(\boldsymbol{q}(x_R,\ t))\mathrm{d}t.
$$

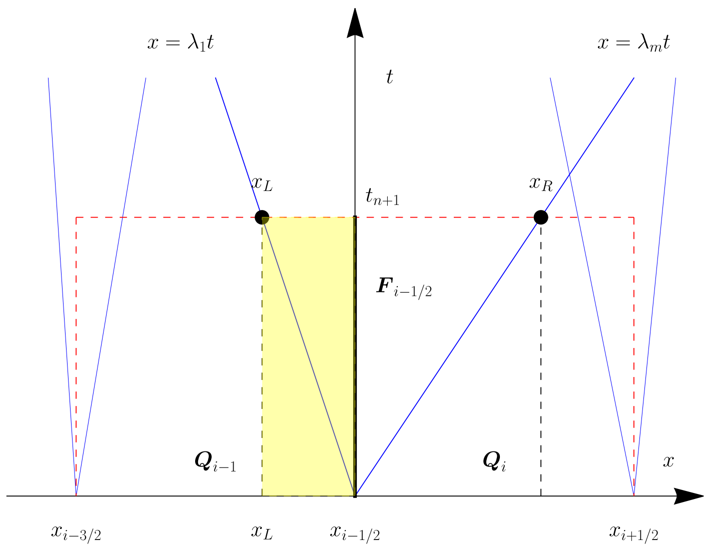


Figure 2.5 Calculating the interface flux $\boldsymbol{F}_{i-1/2}$ in the HLL method. The yellow area is the integrating domain $[x_{i-3/2}, x_{i-1/2}] \times [t_n, t_{n+1}]$.

The contributions to the right-hand side are easy to calculate by interpreting the various terms in Fig. 2.5

$$\begin{aligned}
\int_{x_L}^{x_{i-1/2}} \boldsymbol{q}(x,\ t_n)\mathrm{d}x &= -\lambda_1 \Delta t \boldsymbol{Q}_{i-1} + \lambda_m \Delta t \boldsymbol{Q}_i,\\
\int_{t_n}^{t_{n+1}} \boldsymbol{f}(\boldsymbol{q}(x_L,\ t))\mathrm{d}t &= \boldsymbol{f}(\boldsymbol{Q}_{i-1})\Delta t,\\
\int_{t_n}^{t_{n+1}} \boldsymbol{f}(\boldsymbol{q}(x_R,\ t))\mathrm{d}t &= \boldsymbol{f}(\boldsymbol{Q}_i)\Delta t.
\end{aligned}$$

We then deduce that

$$\int_{x_L}^{x_R} \boldsymbol{q}(x,\ t_{n+1})\mathrm{d}x = \Delta t\left(\lambda_m \boldsymbol{Q}_i - \lambda_1 \boldsymbol{Q}_{i-1} + \boldsymbol{f}(\boldsymbol{Q}_{i-1}) - \boldsymbol{f}(\boldsymbol{Q}_i)\right),$$

and thus the mean value $\boldsymbol{Q}^{LR}$ of $\boldsymbol{q}(x,\ t_{n+1})$ over the domain $[x_L,\ x_R]$ is

$$\boldsymbol{Q}^{LR} = \frac{1}{x_R - x_L}\int_{x_L}^{x_R} \boldsymbol{q}(x,\ t_{n+1})\mathrm{d}x = \frac{\lambda_m \boldsymbol{Q}_i - \lambda_1 \boldsymbol{Q}_{i-1} + \boldsymbol{f}(\boldsymbol{Q}_{i-1}) - \boldsymbol{f}(\boldsymbol{Q}_i)}{\lambda_m - \lambda_1}$$

We can use this expression to evaluate the integral on the right-hand side of Eq. (2.34), and we can deduce the flux at $x_{i-1/2}$

$$\boldsymbol{F}_{i-1/2} = \frac{\lambda_m \boldsymbol{f}(\boldsymbol{Q}_{i-1}) - \lambda_1 \boldsymbol{f}(\boldsymbol{Q}_i) + \lambda_1\lambda_m(\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1})}{\lambda_m - \lambda_1}. \tag{2.35}$$

To sum up, the HLL solver determines the flux $\boldsymbol{F}_{i-1/2}$ without trying to determine the value of $\boldsymbol{Q}_{i-1/2}$ as in the Godunov method:

$$\boldsymbol{F}_{i-1/2} = \begin{cases} \boldsymbol{f}(\boldsymbol{Q}_{i-1}) & \text{if } \lambda_1 > 0\\ \dfrac{\lambda_m \boldsymbol{f}(\boldsymbol{Q}_{i-1}) - \lambda_1 \boldsymbol{f}(\boldsymbol{Q}_i) + \lambda_1\lambda_m(\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1})}{\lambda_m - \lambda_1} & \text{if } \lambda_1 < 0 < \lambda_m\\ \boldsymbol{f}(\boldsymbol{Q}_i) & \text{if } \lambda_m < 0 \end{cases} \tag{2.36}$$

2.4 Approximate Riemann solvers for nonlinear problems

Earlier in this chapter, we have seen that a general time-marching algorithm to solve the hyperbolic equation (2.1) is given by Eq. (2.6):

$$\boldsymbol{Q}_i^{n+1} = \boldsymbol{Q}_i^n - \frac{\Delta t}{\Delta x}\left(\boldsymbol{F}_{i+1/2}^n - \boldsymbol{F}_{i-1/2}^n\right).$$

At each interface $x_{i-1/2}$, the flux function is given by

$$\boldsymbol{F}_{i-1/2}^n = \boldsymbol{f}(\boldsymbol{Q}_{i-1/2}^n),$$

where $\boldsymbol{Q}_{i-1/2}^n$ is the value of $\boldsymbol{Q}$ obtained along the ray $x = x_{i-1/2}$. It depends on the values $\boldsymbol{Q}_i^n$ and $\boldsymbol{Q}_{i-1}^n$ of either side of the interface. In the absence of a source terme, $\boldsymbol{Q}_i^n$ remains constant along this ray.

2.4.1 Scalar problems

Scalar Riemann problems are associated with five possible wave configurations (see Fig. 2.6):

(a) Left-going shock wave: $Q_{i-1/2}^n = Q_i^n$.

(b) Left-going rarefaction wave: $Q_{i-1/2}^n = Q_i^n$.

(c) Transonic[2] rarefaction wave: $Q_{i-1/2}^n = q_s(Q_i^n, Q_i^n)$. This is the only case for which we cannot set the $Q_{i-1/2}$ value in a simple way. Further calculations are needed to evaluate the value q_s. The unknown value q_s satisfies

$$Q_{i-1}^n < q_s < Q_i^n,$$

and this is associated with the vertical ray, its characteristic speed is zero. Therefore, q_s is the solution to

$$f'(q_s) = 0. \tag{2.37}$$

(d) Right-going rarefaction wave: $Q_{i-1/2}^n = Q_{i-1}^n$.

(e) Right-going shock wave: $Q_{i-1/2}^n = Q_{i-1}^n$.

For a convex scalar flux, we can summarise all these possibilities

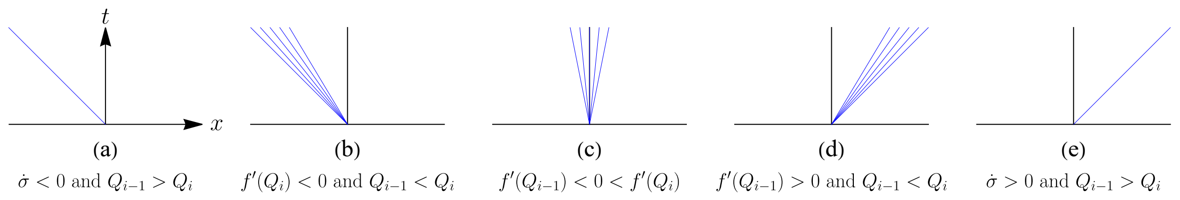
$$F_{i-1/2}^n = \begin{cases} f(Q_{i-1}^n) & \text{if } Q_{i-1}^n > q_s \text{ and } \dot\sigma > 0 \\ f(Q_i^n) & \text{if } Q_i^n < q_s \text{ and } \dot\sigma < 0 \\ f(q_s) & \text{if } Q_{i-1}^n < q_s < Q_i^n, \end{cases} \tag{2.38}$$

where the shock speed $\dot\sigma$ is given by:

$$\dot\sigma = \frac{f(Q_i^n) - f(Q_{i-1}^n)}{Q_i^n - Q_{i-1}^n}.$$

Proof. It can be seen that cases (a) and (b) correspond to left-going waves with the following possibilities:

[2]It is called transonic because it moves with velocity 0. In gas dynamics, this happens when one of the eigenvalues $u \pm c$ (c: sound speed) takes the value 0, thus when the fluid moves at the same speed as sound. In Fig. 2.6(c) the fluid is accelerated from a subsonic velocity to a supersonic one through a rarefaction wave.



t
x
(a)
$\dot\sigma < 0$ and $Q_{i-1} > Q_i$
(b)
$f'(Q_i) < 0$ and $Q_{i-1} < Q_i$
(c)
$f'(Q_{i-1}) < 0 < f'(Q_i)$
(d)
$f'(Q_{i-1}) > 0$ and $Q_{i-1} < Q_i$
(e)
$\dot\sigma > 0$ and $Q_{i-1} > Q_i$

Figure 2.6 The five possible solutions to a scalar Riemann problem: (a) left-going shock wave; (b) left-going rarefaction wave; (c) transonic rarefaction wave; (d) right-going rarefaction wave; and (e) right-going shock wave.

(a) if initially $Q_{i-1}^n > Q_i^n$, the wave is a shock propagating at velocity $\dot\sigma < 0$, and the characteristic velocity is negative: $f'(Q_i^n) < 0$. Since f is assumed to be convex (and thus $f'(q)$ is an increasing function of q) and q_s is defined as $f'(q_s) = 0$, then $Q_i^n < q_s$.

(b) if initially $Q_{i-1}^n < Q_i^n$, then the wave is a rarefaction wave. The largest velocity must be negative: $f'(Q_i^n) < 0$. Since f is convex, then

$$f'(Q_{i-1}) < \frac{f(Q_i^n) - f(Q_{i-1}^n)}{Q_i^n - Q_{i-1}^n} < f'(Q_i),$$

and thus the ratio $\dot\sigma$ is also negative.

We repeat the reasoning for cases (c) to (e). □

A more compact way used in Clawpack is given

$$F_{i-1/2}^n = \begin{cases} \min\limits_{Q_{i-1}^n \le q \le Q_i^n} f(q) & \text{if } Q_{i-1}^n \le Q_i^n, \\ \max\limits_{Q_i^n \le q \le Q_{i-1}^n} f(q) & \text{if } Q_{i-1}^n \ge Q_i^n, \end{cases} \tag{2.39}$$

because q_s is the global minium or maximum of f. This equation can be generalised to the non-convex case.

Remark. We have seen in § 1.2.2 that the Lax entropy condition is an extra condition imposed to shock waves for them to be physically admissible. A shock wave must dissipate energy, not create energy (or from a thermodynamical standpoint, entropy increases through a shock, and does not decrease). A shock wave satisfies the Lax entropy condition if its speed $\dot\sigma$ lies between bounds fixed by the initial data

$$f'(Q_{i-1}^n) > \dot\sigma > f'(Q_i^n). \tag{2.40}$$

For a scalar problem, the time-marching algorithm to solve the hyperbolic equation (2.1) is given by Eq. (2.6):

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

or equivalently by using the notation $F_{i-1/2}^n = f(Q_{i-1/2}^{\downarrow})$

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\left[f(Q_{i+1/2}^{\downarrow}) - f(Q_i^n) - \left(f(Q_{i-1/2}^{\downarrow}) - f(Q_i^n)\right)\right].$$

We can make an analogy with the formulation for linear equations, which emphasizes the role of fluctuations (see § 2.3). We put the equation above in a form consistent with LeVeque's notation:

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\left(\mathcal{A}^+\Delta Q_{i-1/2} + \mathcal{A}^-\Delta Q_{i+1/2}\right),$$

where the operator $\mathcal{A}^{\pm}$ represent flux fluctuations, and are defined by

$$\mathcal{A}^{+}\Delta Q_{i-1/2} = f(Q_i^n) - f(Q^{\downarrow}_{i-1/2}),$$

$$\mathcal{A}^{-}\Delta Q_{i+1/2} = f(Q^{\downarrow}_{i+1/2}) - f(Q_i^n).$$

High-resolution techniques involve defining the wave $W_{i-1/2}$ and speed $\dot{\sigma}_{i-1/2}$ associated with the Riemann problem:

$$W_{i-1/2} = Q_i - Q_{i-1},$$

$$\dot{\sigma}_{i-1/2} = \begin{cases} \dfrac{f(Q_i^n) - f(Q_{i-1}^n)}{Q_i^n - Q_{i-1}^n} & \text{if } Q_i \neq Q_{i-1}, \\ f'(Q_i^n) & \text{if } Q_i = Q_{i-1}. \end{cases}$$

When the Riemann solution is a shock wave, the speed chosen is the one given by the Rankine-Hugoniot equation. When it is a rarefaction wave, we have seen in § 1.4.3 that the Rankine-Hugoniot equation provides a correct estimate of the actual wave speed, and the wave behaviour can be approximated locally by a shock wave even if the latter would not satisfy the entropy condition (2.40). The big advantage is that we can treat all waves as (local) shock waves regardless of their actual nature. When the solution is not a transonic wave, we can also express the fluctuations as:

$$\mathcal{A}^{+}\Delta Q_{i-1/2} = \dot{\sigma}^{+}_{i-1/2} W_{i-1/2}, \tag{2.41}$$

$$\mathcal{A}^{-}\Delta Q_{i+1/2} = \dot{\sigma}^{-}_{i+1/2} W_{i+1/2}, \tag{2.42}$$

where $\dot{\sigma}^{+} = \max(\dot{\sigma}, 0)$ and $\dot{\sigma}^{-} = \min(\dot{\sigma}, 0)$. Equation (2.41) is used in Clawpack for solving scalar problems.

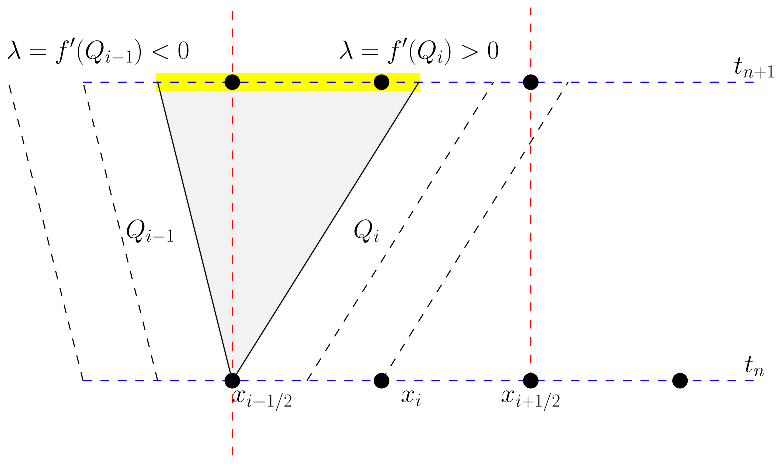


Figure 2.7 A transonic wave occurs whenever the velocity on the left and the right of the interface $x_{i-1/2}$ are not of the same sign. In this case, part of the interval $[x_{i-1/2}, x_{i+1/2}]$ at time t_{n+1}, highlighted in yellow in the figure, is not affected by the conditions at time t_n. There is thus a transonic wave that propagates information in the gray wedge around the interface $x_{i-1/2}$.

When the Riemann solution consists of a transonic rarefaction wave, the fluctuation terms $\mathcal{A}^{\pm}\Delta Q_{i\pm1/2}$ need to be corrected using an *entropy fix*. In Clawpack, the wave W_i and speed $\dot{\sigma}_i$ are first computed, and from them, we determine the fluctuations using Eq. (2.41). If $f'(Q_{i-1}^n) < 0 < f'(Q_i^n)$

(see Fig. 2.7), then the fluctuations in Eq. (2.41) are replaced by one of the equations (for the interface from which the transonic wave originates):

$$\mathcal{A}^+\Delta Q_{i-1/2} = f(Q_i^n) - f(q_s), \tag{2.43}$$
$$\mathcal{A}^-\Delta Q_{i+1/2} = f(q_s) - f(Q_i^n). \tag{2.44}$$

Although this approach based on an entropy fix is unnecessary for scalar problems, it is easy to generalise to nonlinear systems of hyperbolic equations, for which there is no easy way to determine the rarefaction wave structure exactly.

2.4.2 Systems of equations

The method used for scalar problems can be generalised to systems of hyperbolic equations. The crux lies in the determination of the interface value $\boldsymbol{Q}^{\downarrow}_{i-1/2}$ (the value of $\boldsymbol{q}$ along the interface). This value is usually one of the intermediate states that connect the left and right states $\boldsymbol{Q}_i^n$ and $\boldsymbol{Q}_{i-1}^n$ through a series of shock and rarefaction waves, and thus it can be expressed as a function of these two states:

$$\boldsymbol{Q}^{\downarrow}_{i-1/2} = \boldsymbol{Q}^{\downarrow}_{i-1/2}(\boldsymbol{Q}_{i-1}^n, \boldsymbol{Q}_i^n).$$

When $\boldsymbol{Q}^{\downarrow}_{i-1/2}$ is part of a transonic rarefaction wave, additional work is required to determine the wave structure.

The computational approach to solving the nonlinear Riemann problem is the same as the one taken for linear problems. When dealing with Godunov's equation (2.6), Clawpack still uses the wave-propagation form

$$\begin{aligned}\boldsymbol{Q}_i^{n+1} &= \boldsymbol{Q}_i^n - \frac{\Delta t}{\Delta x}\left(\boldsymbol{F}_{i+1/2}^n - \boldsymbol{F}_{i-1/2}^n\right),\\ &= \boldsymbol{Q}_i^n - \frac{\Delta t}{\Delta x}\left(\boldsymbol{\mathcal{A}}^+\Delta\boldsymbol{Q}_{i-1/2}^n + \boldsymbol{\mathcal{A}}^-\Delta\boldsymbol{Q}_{i+1/2}^n\right),\end{aligned} \tag{2.45}$$

where the operators $\boldsymbol{\mathcal{A}}^\pm$ represent the fluctuations, and are defined by generalising the linear case (see § 2.3.3):

$$\boldsymbol{\mathcal{A}}^-\Delta\boldsymbol{Q}_{i+1/2}^n = \boldsymbol{f}(\boldsymbol{Q}^{\downarrow n}_{i+1/2}) - \boldsymbol{f}(\boldsymbol{Q}_i^n), \tag{2.46}$$
$$\boldsymbol{\mathcal{A}}^+\Delta\boldsymbol{Q}_{i-1/2}^n = \boldsymbol{f}(\boldsymbol{Q}_i^n) - \boldsymbol{f}(\boldsymbol{Q}^{\downarrow n}_{i-1/2}), \tag{2.47}$$

These definitions are useful when the solution to the Riemann problem is a transonic wave.

When the solution is a shock or rarefaction wave, the fluctuations can also be approximated by considering that in the close vicinity of the initial state the solution behaves like a shock wave (see § 1.4.3), and like in the linear case, the fluctuations are given by:

$$\boldsymbol{\mathcal{A}}^-\Delta\boldsymbol{Q}_{i+1/2}^n = \sum_{k=1}^{M_w} \dot\sigma^-_{k,i+1/2}\boldsymbol{W}^n_{k,i+1/2}, \tag{2.48}$$
$$\boldsymbol{\mathcal{A}}^+\Delta\boldsymbol{Q}_{i-1/2}^n = \sum_{k=1}^{M_w} \dot\sigma^+_{k,i-1/2}\boldsymbol{W}^n_{k,i-1/2}, \tag{2.49}$$

where M_w is the number of waves (usually $M_w = m$), $\dot\sigma_{k,i+1/2}$ is the speed of the kth wave $\boldsymbol{W}_{k,i+1/2}$ at $x_{i-1/2}$, $\dot\sigma^- = \min(0, \dot\sigma)$ and $\dot\sigma^+ = \max(0, \dot\sigma)$. When the solution involves only shocks, the kth wave speed s_k is the k eigenvalue ($\dot\sigma_k = \lambda_k$) and the Eqs. (2.48)–(2.49) give the same results as Eqs. (2.46)–(2.47) do, but in the general case, they are just an approximation of the interface fluxes. When the solution involves rarefaction waves, their speed varies spatially, but a correct approximation is the following:

$$\dot{\sigma}_{k,i-1/2} = \frac{\lambda_{k,i-1} + \lambda_{k,i}}{2}.$$

The computational cost is high if we use exact Riemann solvers. A variety of approximate Riemann solvers have been proposed to reduce this cost (Toro, 2001; LeVeque, 2002). If a linearised Riemann solver (see below) is used, then the operators $\boldsymbol{\mathcal{A}}^{\pm}$ are based on the positive and negative parts $\hat{\boldsymbol{A}}^{\pm}$ of the linearised solver $\hat{\boldsymbol{A}}$, the wave speeds $\dot{\sigma}_k$ match its eigenvalues $\hat{\lambda}_k$, and the waves $\boldsymbol{W}_k$ are right eigenvectors of $\hat{\boldsymbol{A}}$. An entropy fix can applied to modify the flux fluctuations.

Linearised solvers

Nonlinear equations

$$\frac{\partial}{\partial t}\boldsymbol{q} + \frac{\partial}{\partial x}\boldsymbol{f}(\boldsymbol{q}) = 0$$

can be linearised when the initial values $\boldsymbol{q}_l$ and $\boldsymbol{q}_r$ are sufficiently close to each other, and put in the linear form

$$\frac{\partial \boldsymbol{q}}{\partial t} + \hat{\boldsymbol{A}} \cdot \frac{\partial \boldsymbol{q}}{\partial x} = 0,$$

for each interface $x_{i-1/2}$. The constant matrix $\hat{\boldsymbol{A}}$ is an approximation of $\boldsymbol{f}'(\boldsymbol{q})$ when $\boldsymbol{Q}_{i-1}$ and $\boldsymbol{Q}_i$ tend to $\boldsymbol{q}$. The approximate solution involves m shock waves that are proportional to the right eigenvectors $\boldsymbol{q}_{i-1/2}$ of $\boldsymbol{A}$ and move at speed $\dot{\sigma}_{i-1/2} = \lambda_{i-1/2}$ given by the eigenvalues of $\boldsymbol{A}$. One possibility among other options is to set

$$\boldsymbol{A} = \boldsymbol{f}'\left(\frac{\boldsymbol{Q}_i + \boldsymbol{Q}_{i-1}}{2}\right).$$

An alternative choice is

$$\hat{\boldsymbol{A}}_{i-1/2} = \frac{1}{2}(\boldsymbol{f}'(\boldsymbol{Q}_{i-1}) + \boldsymbol{f}'(\boldsymbol{Q}_i)).$$

This approach works fine when the solution is continuous, but may run into trouble when the solution involves discontinuities. Other constraints are needed to avoid the difficulties. The Roe function (studied later) is an example of linearised solvers (Roe, 1981).

When using the linearised Roe solver, the approximate solution involve only shock waves. As shock and rarefaction waves have the same behaviour in the close vicinity of a given state (see § 1.3.2), the approximate solution is expected to mimic the exact solution. There are, however, situations in which the agreement is no longer good: indeed, we have seen for the nonlinear case that when $\lambda(Q_{i-1}) < 0 < \lambda(Q_i)$, then the left and right states Q_i and Q_{i-1} are connected by a rarefaction wave (see § 2.4.1). For hyperbolic systems, the same problem will arise whenever one or more eigenvalues λ^k satisfy $\lambda^k_{i-1} < 0 < \lambda^k_i$. The Riemann solver must be modified to handle this case properly. Such a modification is called an *entropy fix*.

Two-wave solvers

Several approximate solvers are based on the idea that the Riemann solution can be approximated by picking up two of the m waves, $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$, and defining an intermediate state $\boldsymbol{Q}_*$ such that $\boldsymbol{W}_1 = \boldsymbol{Q}_* - \boldsymbol{Q}_l$ and $\boldsymbol{W}_2 = \boldsymbol{Q}_r - \boldsymbol{Q}_*$. The Rankine-Hugoniot implies that $\boldsymbol{f}(\boldsymbol{Q}_l) - \boldsymbol{f}(\boldsymbol{Q}_*) = \dot{\sigma}_1 \boldsymbol{W}_1$ and $\boldsymbol{f}(\boldsymbol{Q}_r) - \boldsymbol{f}(\boldsymbol{Q}_*) = \dot{\sigma}_2 \boldsymbol{W}_2$. By adding these two equations, we end up with a system of m equations

$$\boldsymbol{f}(\boldsymbol{Q}_r) - \boldsymbol{f}(\boldsymbol{Q}_l) = \dot{\sigma}_1 \boldsymbol{W}_1 + \dot{\sigma}_2 \boldsymbol{W}_2,$$

which gives

$$\boldsymbol{Q}_* = \frac{\boldsymbol{f}(\boldsymbol{Q}_r) - \boldsymbol{f}(\boldsymbol{Q}_l) - \dot{\sigma}_2 \boldsymbol{Q}_r + \dot{\sigma}_1 \boldsymbol{Q}_l}{\dot{\sigma}_1 - \dot{\sigma}_2}.$$

The various solvers proposed so far differ by the choice of the speeds $\dot{\sigma}_1$ and $\dot{\sigma}_2$ along with the waves $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$. Lax-Friedrichs and Harten-Lax-van Leer (HLL) are classic solvers (see § 2.3.3 for a rationale).

The advantage of HLL solvers is that they usually do not need an entropy fix to compute transonic rarefaction waves. As they involve only two waves (thereby ignoring all other waves), they may lead to poorer resolutions for systems made of $m > 2$ equations (Toro, 2019).

2.5 Roe solver

The Roe solver linearises the governing equation (2.1):

$$\frac{\partial \boldsymbol{q}}{\partial t} + \hat{\boldsymbol{A}}_{i-1/2} \frac{\partial \boldsymbol{q}}{\partial x} = 0.$$

The matrix $\hat{\boldsymbol{A}}_{i-1/2}$ is constructed so that it approximates $\boldsymbol{f}'(\boldsymbol{q})$ in the neighbourhood of $\boldsymbol{Q}_i$ and $\boldsymbol{Q}_{i-1}$ and satisfies the conditions

1. Continuity condition:
$$\hat{\boldsymbol{A}}_{i-1/2} \to \boldsymbol{f}'(\boldsymbol{q}) \text{ when } \boldsymbol{Q}_{i-1}, \boldsymbol{Q}_i \to \boldsymbol{q}.$$
2. Hyperbolicity: $\hat{\boldsymbol{A}}_{i-1/2}$ is diagonisable, with m right eigenvectors $\hat{\boldsymbol{w}}_{k,i-1/2}$ associated with eigenvalues $\lambda_{k,i-1/2}$.
3. Roe linearisation. This third property states that if $\boldsymbol{Q}_{i-1}$ and $\boldsymbol{Q}_i$ are connected by a single wave $\boldsymbol{W}_{i-1/2} = \boldsymbol{Q}_i - \boldsymbol{Q}_{i-1}$ in the original Riemann problem, then $\boldsymbol{W}_{i-1/2}$ should also be an eigenvector of $\hat{\boldsymbol{A}}_{i-1/2}$:
$$\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1}) = \hat{\boldsymbol{A}}_{i-1/2} \cdot (\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1}) = \dot{\sigma}(\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1}), \tag{2.50}$$
where $\dot{\sigma}$ is the wave speed: $\dot{\sigma} = \lambda_{k,i-1/2}$ since $\boldsymbol{W}_{i-1/2}$ is a right eigenvector of $\hat{\boldsymbol{A}}_{i-1/2}$.

Formally, the matrix $\hat{\boldsymbol{A}}_{i-1/2}$ can be determined by integrating the Jacobian over a straight-line path $\boldsymbol{q}(\xi) = \boldsymbol{Q}_{i-1} + \xi(\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1})$:

$$\hat{\boldsymbol{A}}_{i-1/2} = \int_0^1 \frac{\mathrm{d}\boldsymbol{f}(\boldsymbol{q}(\xi))}{\mathrm{d}\boldsymbol{q}} \mathrm{d}\xi, \tag{2.51}$$

for $0 \le \xi \le 1$. Indeed, we have

$$\begin{aligned} \boldsymbol{f}(\boldsymbol{Q}) - \boldsymbol{f}(\boldsymbol{Q}_{i-1}) &= \int_0^1 \frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\xi} \mathrm{d}\xi \\ &= \int_0^1 \frac{\mathrm{d}\boldsymbol{f}(\boldsymbol{q}(\xi))}{\mathrm{d}\boldsymbol{q}} \cdot \frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}\xi} \mathrm{d}\xi \\ &= \left(\int_0^1 \frac{\mathrm{d}\boldsymbol{f}(\boldsymbol{q}(\xi))}{\mathrm{d}\boldsymbol{q}} \mathrm{d}\xi \right) \cdot (\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1}), \end{aligned}$$

since the ξ derivative of q is $\boldsymbol{q}' = \boldsymbol{Q}_i - \boldsymbol{Q}_{i-1}$. By comparing the equation above with Eq. (2.50), we deduce that the linearised matrix is given by Eq. (2.51).

There is no guarantee that the resulting matrix is diagonisable with m real eigenvalues and that it takes an analytical form. By making a change of variable, Roe (1981) showed that this difficulty can often be overcome (LeVeque, 2002, see pp. 317-323).

2.6 Two-wave solver: HLL solver

The idea underpinning the HLL solver's derivation is that the solution to the Riemann problem consists of two shock waves separating an intermediate state from the left and right initial states (see also § 2.3.3). The speeds $\dot{\sigma}_1$ and $\dot{\sigma}_2$ of these waves are given by the Rankine-Hugoniot equation

$$\boldsymbol{f}(\boldsymbol{Q}_{i-1}) - \boldsymbol{f}(\boldsymbol{Q}_*) = \dot{\sigma}_1(\boldsymbol{Q}_{i-1} - \boldsymbol{Q}_*), \tag{2.52}$$

$$\boldsymbol{f}(\boldsymbol{Q}_*) - \boldsymbol{f}(\boldsymbol{Q}_i) = \dot{\sigma}_2(\boldsymbol{Q}_* - \boldsymbol{Q}_i) \tag{2.53}$$

Solving Eqs. (2.52) and (2.53) for $\boldsymbol{Q}_*$ and $\boldsymbol{F}_* = \boldsymbol{f}(\boldsymbol{Q}_*)$ gives

$$\boldsymbol{Q}_* = \frac{\dot{\sigma}_2\boldsymbol{Q}_i - \dot{\sigma}_1\boldsymbol{Q}_{i-1}}{\dot{\sigma}_2 - \dot{\sigma}_1} + \frac{\boldsymbol{F}_{i-1} - \boldsymbol{F}_i}{\dot{\sigma}_2 - \dot{\sigma}_1} \tag{2.54}$$

$$\boldsymbol{F}_* = \frac{\dot{\sigma}_2\boldsymbol{F}_{i-1} - \dot{\sigma}_1\boldsymbol{F}_i}{\dot{\sigma}_2 - \dot{\sigma}_1} - \dot{\sigma}_2\dot{\sigma}_2\frac{\boldsymbol{Q}_{i-1} - \boldsymbol{Q}_i}{\dot{\sigma}_2 - \dot{\sigma}_1}, \tag{2.55}$$

with $\boldsymbol{F}_i = \boldsymbol{f}(\boldsymbol{Q}_i)$. For the Harten–Lax–van-Leer (HLL) solver, the speeds are defined as the lower and upper bounds of all characteristic speeds. Einfeldt (1988) suggested

$$\dot{\sigma}_{1,\, i-1/2} = \min_{1\le k\le m}\left(\min(\lambda_{k,\, i}, \hat{\lambda}_{k,\, i-1/2})\right), \tag{2.56}$$

$$\dot{\sigma}_{2,\, i-1/2} = \max_{1\le k\le m}\left(\max(\lambda_{k,\, i-1}, \hat{\lambda}_{k,\, i-1/2})\right), \tag{2.57}$$

where $\lambda_{k,\, i}$ is the kth eigenvalue of the Jacobian $\boldsymbol{f}'(\boldsymbol{Q}_i)$ and $\hat{\lambda}_{k,\, i-1/2}$ is kth eigenvalue of the Roe matrix (linearised Jacobian). The resulting scheme is called the HLLE solver.

2.7 Alternative: the f-wave method

An alternative approach to the wave decomposition is to first split the jump in $\boldsymbol{f}$ into f-waves:

$$\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1}) = \sum_{k=1}^{m_w} \boldsymbol{Z}_{k,i-1/2}, \tag{2.58}$$

moving at speeds $\dot{\sigma}_{k,i-1/2}$, then express the fluctuations in terms of the f-waves. This method is useful to study the second-order accuracy of wave-propagation methods or in the context of spatially-varying flux functions $\boldsymbol{f}(\boldsymbol{q}, x)$ (LeVeque, 2002, see § 15.5). It also guarantees that approximate Riemann solvers are conservative.

When dealing with a linear or linearised problems, we can decompose $\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1})$ as a linear combination of the right eigenvectors $\hat{\boldsymbol{w}}_{k,i-1/2}$ of the linearised matrix $\hat{\boldsymbol{A}}_{i-1/2}$:

$$\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1}) = \sum_{k=1}^{m_w} \beta^k_{i-1/2}\hat{\boldsymbol{w}}^k_{i-1/2}, \tag{2.59}$$

where the coefficient vector $\boldsymbol{\beta}_{i-1/2}$ is the solution to the linear system (2.59):

$$\boldsymbol{\beta}_{i-1/2} = \boldsymbol{R}^{-1}\cdot(\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1})) = \boldsymbol{L}\cdot(\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1})). \tag{2.60}$$

The f-waves are then

$$\boldsymbol{Z}^k_{i-1/2} = \beta^k_{i-1/2}\hat{\boldsymbol{w}}^k_{i-1/2}. \tag{2.61}$$

These f-waves are related to the waves $\boldsymbol{W}_{k,i-1/2}$ when the wave speeds are nonzero

$$\boldsymbol{W}^k_{i-1/2} = \frac{\boldsymbol{Z}^k_{i-1/2}}{\dot{\sigma}^k_{i-1/2}}, \tag{2.62}$$

and the fluctuations are

$$\boldsymbol{A}^-_{i-1/2}\Delta\boldsymbol{Q}_{i-1/2} = \sum_{k:\ \dot{\sigma}_k<0} \boldsymbol{Z}^k_{i-1/2}, \tag{2.63}$$

$$\boldsymbol{A}^+_{i-1/2}\Delta\boldsymbol{Q}_{i-1/2} = \sum_{k:\ \dot{\sigma}_k>0} \boldsymbol{Z}^k_{i-1/2}. \tag{2.64}$$

Another advantage of the f-wave decomposition is the possibility of including the source term into this decomposition (Ketcheson *et al.*, 2013). The f-wave approach is highly recommended when the solution is close to a steady state, as it leads to a well-balanced system. Let us assume that we are concerned with the equation

$$\frac{\partial}{\partial t}\boldsymbol{q} + \frac{\partial}{\partial x}\boldsymbol{f}(\boldsymbol{q}) = \boldsymbol{S}(\boldsymbol{q}, x) \tag{2.65}$$

where $\boldsymbol{S}(\boldsymbol{q}, x)$ is the source term. We now express the flux difference at the interface $x_{i-1/2}$

$$\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1}) - \Delta x\boldsymbol{S}(\boldsymbol{Q}_i, \boldsymbol{Q}_{i-1}, x_{i-1/2}) = \sum_{k=1}^{m_w} \boldsymbol{Z}^k_{i-1/2} = \beta^k_{i-1/2}\hat{\boldsymbol{w}}^k_{i-1/2}, \tag{2.66}$$

where $\boldsymbol{S}(\boldsymbol{Q}_i, \boldsymbol{Q}_{i-1}, x_{i-1/2})$ is a representative value of the source term $\boldsymbol{S}$ at the interface $x_{i-1/2}$, which depends on $\boldsymbol{Q}_i$ and $\boldsymbol{Q}_{i-1}$.

2.8 High-resolutions methods

2.8.1 *Scalar problems*

Derivation of the time-marching algorithm

Let us consider the scalar advection equation

$$\frac{\partial q}{\partial t} + \bar{u}\frac{\partial q}{\partial x} = 0, \tag{2.67}$$

where $\bar{u} > 0$ denotes the advection velocity. We assume that from the cell average Q_i^n at time t_n, we can infer a piecewise-polynomial approximation order p

$$\tilde{q}(x, t_n) = \sum_{k=0}^{p} a_{k,i}(x - x_i)^k = Q_i^n + \sum_{k=1}^{p} a_{k,i}(x - x_i)^k \text{ over } x_{i-1/2} \le x \le x_{i+1/2}, \tag{2.68}$$

where the coefficients $a_{k,i}$ satisfy

$$a_{0,i} = Q_i^n \text{ and } \int_{x_{i-1/2}}^{x_{i+1/2}} \sum_{k=1}^{p} a_{k,i}(x - x_i)^k \mathrm{d}x = \sum_{k=1}^{p} a_{k,i}\frac{((-1)^k + 1)}{k+1}\left(\frac{\Delta x}{2}\right)^{k+1} = 0$$

so that the cell average of $\tilde{q}$ is Q_i^n.

The solution to Eq. (2.67) subject to the initial condition (2.68) is

$$q(x, t_{n+1}) = \tilde{q}(x - \bar{u}\Delta t, t^n). \tag{2.69}$$

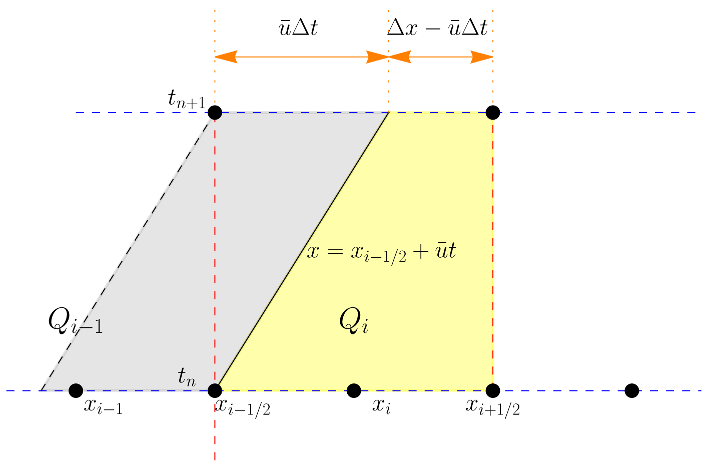


Figure 2.8 Domain of influence: the yellow area shows the time domain influenced by the condition $q^n(x, t_n)$ over $[x_{i-1/2}, x_{i+1/2}]$ at time t^{n+1} while the grey are shows the domain influenced by $q^n(x, t_n)$ over $[x_{i-3/2}, x_{i-1/2}]$.

Integrating Eq. (2.69) over $[x_{i-1/2}, x_{i+1/2}]$ gives

$$\begin{aligned} Q_i^{n+1} &= \frac{1}{\Delta x}\int_{x_{i-1/2}}^{x_{i+1/2}} q(x, t^{n+1})\mathrm{d}x \\ &= \frac{1}{\Delta x}\int_{x_{i-1/2}}^{x_{i+1/2}} \tilde{q}(x - \bar{u}\Delta t, t^n)\mathrm{d}x \\ &= \frac{1}{\Delta x}\int_{x_{i-1/2}-\bar{u}\Delta t}^{x_{i+1/2}-\bar{u}\Delta t} \tilde{q}(\xi, t^n)\mathrm{d}\xi \end{aligned}$$

and taking into account the domain of influence (see Fig. 2.8) gives

$$\begin{aligned} Q_i^{n+1} = Q_{i-1}^n \frac{\bar{u}\Delta t}{\Delta x} + Q_i^n\left(1 - \frac{\bar{u}\Delta t}{\Delta x}\right) &+ \frac{1}{\Delta x}\sum_{k=1}^{p}\int_{x_{i-1/2}-\bar{u}\Delta t}^{x_{i-1/2}} a_{k,i-1}(x - x_{i-1})^k\mathrm{d}x \\ &+ \frac{1}{\Delta x}\sum_{k=1}^{p}\int_{x_{i-1/2}}^{x_{i+1/2}-\bar{u}\Delta t} a_{k,i}(x - x_i)^k\mathrm{d}x. \end{aligned}$$

To first order in Δx and second order in Δt, this expression reads

$$Q_i^{n+1} = Q_i^n - \frac{\bar{u}\Delta t}{\Delta x}(Q_i^n - Q_{i-1}^n) - \frac{1}{2}\frac{\bar{u}\Delta t}{\Delta x}(\Delta x - \bar{u}\Delta t)(a_{1,i} - a_{1,i-1}). \quad (2.70)$$

Alternative 1

This result can be derived differently (which will be useful when extending the method to higher dimensions). The second-order Taylor expansion of $q(x, t_{n+1} - \Delta t)$ is

$$\begin{aligned} q(x, t_n) = q(x, t_{n+1} - \Delta t) &= q(x, t_{n+1}) - \Delta t\frac{\partial}{\partial t}q(x, t_{n+1}) + \frac{\Delta t^2}{2}\frac{\partial}{\partial t}q(x, t_{n+1}), \\ &= q(x, t_{n+1}) + \bar{u}\Delta t\frac{\partial}{\partial x}q(x, t_{n+1}) + \frac{\bar{u}^2\Delta t^2}{2}\frac{\partial^2}{\partial x^2}q(x, t_{n+1}), \end{aligned} \quad (2.71)$$

since

$$\frac{\partial q}{\partial t} = -\bar{u}\frac{\partial q}{\partial x},$$
$$\frac{\partial^2 q}{\partial t^2} = -\bar{u}\frac{\partial^2 q}{\partial t \partial x} = -\bar{u}\frac{\partial}{\partial x}\left(\frac{\partial q}{\partial t}\right) = \bar{u}^2\frac{\partial^2 q}{\partial x^2}.$$

By integrating Eq. (2.71) over $[x_{i-1/2}, x_{i+1/2}]$ and dividing by Δx gives

$$Q_i^n = Q_i^{n+1} + \bar{u}\frac{\Delta t}{\Delta x}\int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial}{\partial x}q(x, t_{n+1})\mathrm{d}x + \frac{\bar{u}^2\Delta t^2}{2\Delta x}\int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial^2}{\partial x^2}q(x, t_{n+1})\mathrm{d}x, \tag{2.72}$$

where the derivatives should be understood in the sense of distributions. Recall that for any piecewise $\mathcal{C}^1$ function q with a jump at x_j

$$q(x,t) = \left\{ \begin{array}{ll} Q_{i-1} + a_{i-1}(x - x_{i-1}) & \text{if } x < x_j, \\ Q_i + a_i(x - x_i) & \text{if } x > x_j \end{array} \right.$$

the derivative of its distribution is

$$T'[q] = [\![q]\!]\delta(x - x_j) + \left\{ \begin{array}{ll} a_{i-1} & \text{if } x < x_j, \\ a_i & \text{if } x > x_j \end{array} \right.$$

where δ is the Dirac function and the jump in q is

$$[\![q]\!] = \lim_{\substack{x\to x_j\\ x>x_j}} q(x,t) - \lim_{\substack{x\to x_j\\ x<x_j}} q(x,t) = Q_i + a_i(x_j - x_i) - Q_{i-1} - a_{i-1}(x_j - x_{i-1}).$$

The second derivative of the associate distribution is

$$T''[q] = [\![q']\!]\delta(x - x_j),$$

where $[\![q']\!] = a_i - a_{i-1}$.

Moreover Eq (2.69) tells us that $q(x, t_{n+1}) = \tilde{q}(x - \bar{u}\Delta t, t^n)$, and thus we can express the integral terms of Eq. (2.72) as

$$\begin{aligned}\int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial}{\partial x}q(x, t_{n+1})\mathrm{d}x &= \int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial}{\partial x}\tilde{q}(x - \bar{u}\Delta t, t_n)\mathrm{d}x \\ &= \int_{x_{i-1/2}-\bar{u}\Delta t}^{x_{i+1/2}-\bar{u}\Delta t} \frac{\partial}{\partial x}\tilde{q}(x, t_n)\mathrm{d}x. \end{aligned} \tag{2.73}$$

Given that the function $\tilde{q}$ is discontinuous at $x_j = x_{i-1/2} = x_i - \Delta x/2$, the jumps are

$$[\![\tilde{q}]\!] = Q_i^n - Q_{i-1}^n - \frac{\Delta x}{2}(a_{i-1} + a_i) \text{ and } [\![\tilde{q}']\!] = a_i - a_{i-1}.$$

With these two definitions, we can decompose the integral term of Eq. (2.73):

$$\begin{aligned}\int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial}{\partial x}\tilde{q}(x, t_{n+1})\mathrm{d}x &= a_{i-1}\bar{u}\Delta t + (\Delta x - \bar{u}\Delta t)a_i + [\![\tilde{q}]\!], \\ &= Q_i^n - Q_{i-1}^n + (a_i - a_{i-1})\frac{\Delta x}{2} - (a_i - a_{i-1})\bar{u}\Delta t, \end{aligned} \tag{2.74}$$

and

$$\int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial^2}{\partial x^2}\tilde{q}(x, t_{n+1})\mathrm{d}x = [\![\tilde{q}']\!] = a_i - a_{i-1}. \tag{2.75}$$

By substituting Eqs. (2.74) and (2.75) into Eq. (2.72), we find the second-order (in time) approximation of Q_i^{n+1} given by Eq. (2.70):

$$Q_i^{n+1} = Q_i^n - \frac{\bar{u}\Delta t}{\Delta x}(Q_i^n - Q_{i-1}^n) - \frac{1}{2}\frac{\bar{u}\Delta t}{\Delta x}(\Delta x - \bar{u}\Delta t)(a_i - a_{i-1}).$$

If we now assume that $\bar{u} < 0$, then we find:

$$\int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial}{\partial x}\tilde{q}(x, t_{n+1})\mathrm{d}x = -a_i\bar{u}\Delta t + (\Delta x + \bar{u}\Delta t)a_{i+1} + [\![\tilde{q}]\!],$$
$$= Q_{i+1}^n - Q_i^n + (a_i - a_{i+1})\frac{\Delta x}{2} + (a_i - a_{i+1})\bar{u}\Delta t,$$

and

$$\int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial^2}{\partial x^2}\tilde{q}(x, t_{n+1})\mathrm{d}x = [\![\tilde{q}']\!] = a_{i+1} - a_i.$$

We deduce that Q_i^{n+1} is given by

$$Q_i^{n+1} = Q_i^n - \frac{\bar{u}\Delta t}{\Delta x}(Q_{i+1}^n - Q_i^n) + \frac{1}{2}\frac{\bar{u}\Delta t}{\Delta x}(\Delta x + \bar{u}\Delta t)(a_{i+1} - a_i). \tag{2.76}$$

We can merge Eqs. (2.70) and (2.76) into a single equation

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}(\bar{u}^+\Delta Q_{i-1/2} + \bar{u}^-\Delta Q_{i+1/2}) - \frac{|\bar{u}|\Delta t}{2}\left(1 - \frac{|\bar{u}|\Delta t}{\Delta x}\right)\Delta a, \tag{2.77}$$

where $\bar{u}^+ = \max(\bar{u}, 0)$, $\bar{u}^- = \min(\bar{u}, 0)$, $\Delta Q_{i-1/2} = Q_i - Q_{i-1}$, $\Delta Q_{i+1/2} = Q_{i+1} - Q_i$, and

$$\Delta a = \begin{cases} a_{i+1} - a_i & \text{if } \bar{u} < 0, \\ a_i - a_{i-1} & \text{if } \bar{u} > 0. \end{cases}$$

Alternative 2

Let us assume again that $\bar{u} > 0$. Equation (2.70) can also be obtained by applying the time-marching algorithm (2.6):

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\left(F_{i+1/2}^n - F_{i-1/2}^n\right), \tag{2.78}$$

where the interface flux is

$$F_{i\pm1/2}^n = \frac{1}{\Delta t}\int_{t_n}^{t_{n+1}} \bar{u}q(x_{i\pm1/2}, t)\mathrm{d}t,$$
$$= \frac{\bar{u}}{\Delta t}\int_{t_n}^{t_{n+1}} \tilde{q}(x_{i\pm1/2} - \bar{u}t)\mathrm{d}t$$
$$= \frac{1}{\Delta t}\int_{x_{i\pm1/2}-\bar{u}\Delta t}^{x_{i\pm1/2}} \tilde{q}(\xi)\mathrm{d}\xi,$$

from which we deduce the following expressions to first order

$$F_{i-1/2} = \bar{u}Q_{i-1} + \frac{1}{2}a_{i-1}\bar{u}(\Delta x - \bar{u}\Delta t), \tag{2.79}$$

and

$$F_{i+1/2} = \bar{u}Q_i + \frac{1}{2}a_i\bar{u}(\Delta x - \bar{u}\Delta t). \tag{2.80}$$

Using these expressions in Eq. (2.78), we obtain Eq. (2.70) again. We can interpret the latter by noting that

$$\bar{u}(Q_i^n - Q_{i-1}^n) = \bar{u}\Delta Q_{i-1/2} = \bar{u}^+\Delta Q_{i-1/2} + \bar{u}^-\Delta Q_{i+1/2},$$

where $\bar{u}^+ = \max(\bar{u}, 0)$, $\bar{u}^- = \min(\bar{u}, 0)$, and defining the flux corrections when $\bar{u} > 0$

$$\hat{F}_{i-1/2} = \frac{a_{i-1}}{2}\bar{u}(\Delta x - \bar{u}\Delta t) \text{ and } \hat{F}_{i+1/2} = \frac{a_i}{2}\bar{u}(\Delta x - \bar{u}\Delta t).$$

we can recast Eq. (2.70) in the following form

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\left(\bar{u}^+\Delta Q_{i-1/2} + \bar{u}^-\Delta Q_{i+1/2}\right) - \frac{\Delta t}{\Delta x}\left(F'_{i+1/2} - F'_{i-1/2}\right). \tag{2.81}$$

When $\bar{u} < 0$, then the second-order (in time) flux corrections are

$$\hat{F}_{i-1/2} = -\frac{a_i}{2}\bar{u}(\Delta x + \bar{u}\Delta t) \text{ and } \hat{F}_{i+1/2} = -\frac{a_{i+1}}{2}\bar{u}(\Delta x + \bar{u}\Delta t).$$

Alternative 3

It is possible to cast the time-marching algorithm (2.70) in a slightly different form. This new form will lead to a set of coupled ordinary differential equations (method of lines). To that end, we need new variables (see Fig. 2.9): at each interface $x_{i-1/2}$, we define the value of q on its right:

$$q^+_{i-1/2} = Q_i^n - a_i\frac{\Delta x}{2}, \tag{2.82}$$

while the value of q on the left of $x_{i-1/2}$ is

$$q^-_{i-1/2} = Q_{i-1}^n + a_{i-1}\frac{\Delta x}{2}. \tag{2.83}$$

The jump of q at the interface is denoted by

$$\Delta q_{i-1/2} = q^+_{i-1/2} - q^-_{i-1/2} = Q_i^n - Q_{i-1}^n - \frac{\Delta x}{2}(a_{i-1} + a_i) \tag{2.84}$$

while the q variation over $[x_{i-1/2}, x_{i+1/2}]$ is denoted by

$$\Delta q_i = q^-_{i+1/2} - q^+_{i-1/2} = a_i\Delta x. \tag{2.85}$$

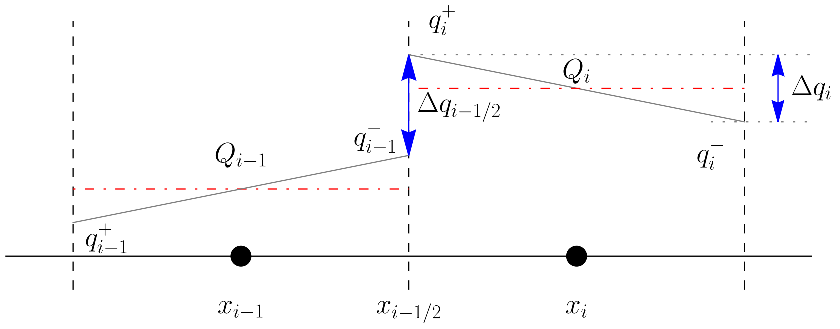


Figure 2.9 Notation.

Substituting Eqs. (2.82) and (2.83) into the time-marching equation (2.70) (for $\bar{u} > 0$) and keeping first-order terms in Δx and Δt leads to

$$Q_i^{n+1} = Q_i^n - \frac{\bar{u}\Delta t}{\Delta x}\left(q^+_{i-1/2} - q^-_{i-1/2} + a_i\Delta x\right) = Q_i^n - \frac{\bar{u}\Delta t}{\Delta x}\left(\Delta q_{i-1/2} + \Delta q_i\right).$$

If we now take the case $\bar{u} < 0$ into account and generalise the previous equatio, we arrive at the following variant of the time-marching equation (2.70):

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\left(\bar{u}^+\Delta q_{i-1/2} + \bar{u}^-\Delta q_{i+1/2} + \bar{u}\Delta q_i\right),$$

and if we take the limit $\Delta t \to 0$, then we obtain a set of coupled ordinary differential equations

$$\frac{\partial Q_i}{\partial t} = -\frac{1}{\Delta x}\left(\bar{u}^+\Delta q_{i-1/2} + \bar{u}^-\Delta q_{i+1/2} + \bar{u}\Delta q_i\right) \text{ for } 1 \le i \le n. \tag{2.86}$$

This equation is used in SharpClaw, which implements *essentially non-oscillating* (ENO) and *weighted essentially non-oscillating* (WENO) algorithms to extend the accuracy of the numerical solutions of hyperbolic problems to arbitrarily high order accuracy by using (i) spatial reconstruction of the solution at each time step t^{n+1} and (ii) a high-order accurate ODE solver (often based a Runge–Kutta algorithm) (Ketcheson *et al.*, 2013). ENO and WENO algorithms are detailed elsewhere (Shu, 1998, 2020).

Choice of slopes

There are three obvious choices for slope a_i when reconstructing a piecewise-linear approximation of $q(x,t)$:

- Centred slope (Fromm)
$$a_i = \frac{Q^n_{i+1} - Q^n_{i-1}}{2\Delta x}.$$
- Upwind slope (beam-warming)
$$a_i = \frac{Q^n_i - Q^n_{i-1}}{\Delta x}.$$
- Downwind slope (Lax–Wendroff)
$$a_i = \frac{Q^n_{i+1} - Q^n_i}{\Delta x}.$$

High-resolution methods improve accuracy when the solution is smooth, but they also tend to smear discontinuities and create spurious fluctuations. A strategy to avoid this issue is to apply a second-order correction wherever the solution is smooth, but to use a first-order scheme near discontinuities. This can be achieved by limiting the slope a_i. Methods implementing this strategy are called *slope-limiter methods*. An example of these methods is the minmod slope:

$$a_i = \text{minmod}\left(\frac{Q^n_i - Q^n_{i-1}}{\Delta x}, \frac{Q^n_{i+1} - Q^n_i}{\Delta x}\right),$$

where the minmod function defined by:

$$\text{minmod}(a,b) = \begin{cases} a \text{ if } |a| < |b| \text{ and } ab > 0, \\ b \text{ if } |b| < |a| \text{ and } ab > 0, \\ 0 \text{ if } ab \le 0, \end{cases}$$

returns the argument that is smaller in absolute value when the arguments are of the same sign, but zero when they are of different sign (which indicates a local minimum or maximum). A variant is the *superbee* limiter:

$$a_i = \text{maxmod}\left(a_i^{(1)}, a_i^{(2)}\right),$$

where the maxmod function returns the largest argument in magnitude and

$$a_i^{(1)} = \text{minmod}\left(\frac{Q_{i+1}^n - Q_i^n}{\Delta x}, 2\frac{Q_i^n - Q_{i-1}^n}{\Delta x}\right),$$

$$a_i^{(2)} = \text{minmod}\left(2\frac{Q_{i+1}^n - Q_i^n}{\Delta x}, \frac{Q_i^n - Q_{i-1}^n}{\Delta x}\right).$$

Slope-limiter methods were initially developed by van Leer in an approach called *monotonic upstream-centered scheme for conservation laws* (MUSCL) (van Leer, 1979).

Alternative: flux-limiter methods

In the time-marching algorithm (2.79), the interface fluxes were given by Eqs. (2.79) and (2.80). Since the slope a_{i-1} is expressed as the difference $\Delta Q_{i-1/2}$ between the cell averages Q_i and Q_{i-1} when applying a slope-limited method, then this means that we actually express the flux (2.79) as

$$\begin{aligned} F_{i-1/2} &= \bar{u}Q_{i-1} + \frac{1}{2}a_{i-1}\bar{u}\Delta x\left(1 - \frac{\bar{u}\Delta t}{\Delta x}\right)a_{i-1}, \\ &= \bar{u}^-Q_i + \bar{u}^+Q_{i-1} + \frac{1}{2}a_{i-1}|\bar{u}|\left(1 - \left|\frac{\bar{u}\Delta t}{\Delta x}\right|\right)a_{i-1}\Delta x, \\ &= \bar{u}^-Q_i + \bar{u}^+Q_{i-1} + \frac{1}{2}a_{i-1}|\bar{u}|\left(1 - \left|\frac{\bar{u}\Delta t}{\Delta x}\right|\right)\delta(\Delta Q_{i-1/2}), \end{aligned}$$

where δ is a function that modulates the difference $\Delta Q_{i-1/2}$. By doing so, we can turn a slope-limiter method into a flux-limiter method:

$$\delta(\Delta Q_{i-1/2}) = \phi(\theta_{i-1/2})\delta(\Delta Q_{i-1/2}), \tag{2.87}$$

where the function ϕ measures the degree of smoothness near $x_{i-1/2}$:

$$\phi(\Delta Q_{i-1/2}) = \frac{\delta(\Delta Q_{j-1/2})}{\delta(\Delta Q_{i-1/2})} \text{ with } j = \begin{cases} i-1 \text{ if } |\bar{u}| > 0, \\ i+1 \text{ if } |\bar{u}| < 0. \end{cases} \tag{2.88}$$

Typically, when the function is smooth at $x_{i-1/2}$, then ϕ should be close to unity, whereas it takes much higher or lower values if there is a discontinuity. For linear methods, we can encode the previous choices of a_i by setting

- Upwind scheme: $\phi(\theta) = 0$.
- Downwind slope (Lax–Wendroff): $\phi(\theta) = 1$.
- Upwind slope (beam-warming): $\phi(\theta) = \theta$.
- Centred slope (Fromm): $\phi(\theta) = \frac{1}{2}(1 + \theta)$.

For high-resolution methods, limiters of common use are:

- minmod: $\phi(\theta) = \text{minmod}(1, \theta)$.

- superbee: $\phi(\theta) = \max(0, \min(1, 2\theta), \min(2, \theta))$.
- monotonised centred (MC): $\phi(\theta) = \max(0, \min((1+\theta)/2, 2, 2\theta))$.
- van Leer:
$$\phi(\theta) = \frac{\theta + |\theta|}{1 + |\theta|}.$$

All these methods are *total-variation diminishing* (TVD):

$$\mathrm{TV}(Q^{n+1}) \leq \mathrm{TV}(Q^n), \tag{2.89}$$

where

$$\mathrm{TV}(Q^{n+1}) = \sum_i |Q_i - Q_{i+1}|.$$

The total variation (2.89) is the discretisation of the total variation for continuous functions (or distributions)

$$\mathrm{TV}(q) = \int |f'(q)| \mathrm{d}x.$$

The exact solution to the linear advection equation does not change shape over time, and thus its total variation must be constant in time. For the numerical solution to the linear advection equation, we expect that the total variation does not increase if no spurious oscillations are created by the numerical scheme.

A theorem due to Harten (1983) guarantees that if ϕ varies within the range

$$0 \leq \phi(\theta) \leq \mathrm{minmod}(1, \theta),$$

then the method is TVD. Sweby (1984) found that second-order TVD methods must lie in the region bounded by the Lax–Wendroff and beam-warming curves (see Fig. 2.10).

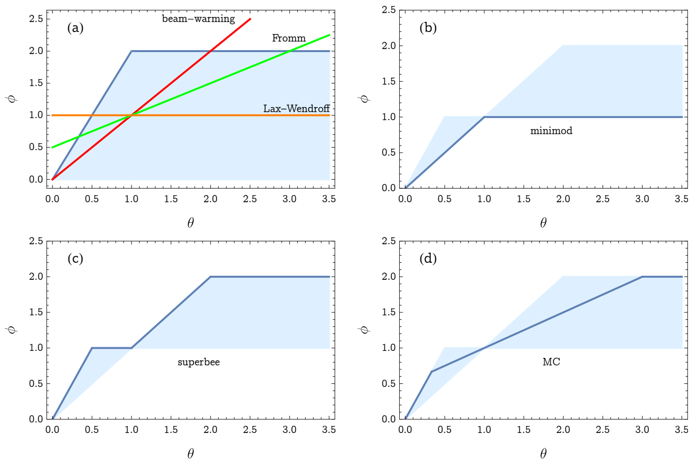


Figure 2.10 (a) Comparison of the beam-warming, Fromm, and Lax–Wendroff limiters. The coloured area shows where the function values must lie for the method to be TVD. (b) Minmod limiter. The coloured area shows where the function values must lie for the method to be second-order TVD. (c) Superbee limiter. (d) MC limiter.

2.8.2 *Linear systems*

The Lax–Wendroff method

Let us consider the following linear hyperbolic problem

$$\frac{\partial \boldsymbol{q}}{\partial t} + \boldsymbol{A}\cdot\frac{\partial \boldsymbol{q}}{\partial x} = 0. \tag{2.90}$$

where $\boldsymbol{q} = (q_k)_{1\leq k\leq n}$ is the unknown vector, $\boldsymbol{A} = \boldsymbol{R}\cdot\boldsymbol{\Lambda}\cdot\boldsymbol{L}$ is a $m\times m$ matrix with right eigenvectors $\boldsymbol{w}_k$ and eigenvalues λ_k ($1\leq k\leq m$). $\boldsymbol{L}$ is the left-eigenvector matrix ($\boldsymbol{L} = \boldsymbol{R}^{-1}$) whose rows are made of the left eigenvectors $\boldsymbol{v}_k$, and $\boldsymbol{\Lambda} = \mathrm{diag}(\lambda_{1\leq k\leq n})$ is the spectral matrix.

The second-order Taylor expansion of $\boldsymbol{q}(x, t_n+\Delta t)$ is

$$\boldsymbol{q}(x, t_n+\Delta t) = \boldsymbol{q}(x,t_n) + \Delta t\frac{\partial}{\partial t}\boldsymbol{q}(x,t_n) + \frac{1}{2}\Delta t^2\frac{\partial^2}{\partial t^2}\boldsymbol{q}(x,t_n). \tag{2.91}$$

From Eq. (2.90), we deduce that

$$\frac{\partial \boldsymbol{q}}{\partial t} = -\boldsymbol{A}\cdot\frac{\partial \boldsymbol{q}}{\partial x},$$

and after differentiating it with respect to time gives

$$\begin{aligned}\frac{\partial^2 \boldsymbol{q}}{\partial t^2} &= -\boldsymbol{A}\cdot\frac{\partial}{\partial x}\left(-\boldsymbol{A}\cdot\frac{\partial \boldsymbol{q}}{\partial x}\right),\\ &= \boldsymbol{A}^2\cdot\frac{\partial^2 \boldsymbol{q}}{\partial x^2}.\end{aligned}$$

Equation (2.90) can be recast

$$\boldsymbol{q}(x, t_n+\Delta t) = \boldsymbol{q}(x,t_n) - \Delta t\boldsymbol{A}\cdot\frac{\partial \boldsymbol{q}}{\partial x}(x,t_n) + \frac{1}{2}\Delta t^2\boldsymbol{A}^2\cdot\frac{\partial^2 \boldsymbol{q}}{\partial x^2}(x,t_n). \tag{2.92}$$

We discretise the spatial derivatives of Eq. (2.92) using central finite differences:

$$\frac{\partial \boldsymbol{q}}{\partial x} = \frac{\boldsymbol{Q}_{i+1}^n - \boldsymbol{Q}_{i-1}^n}{2\Delta x} \text{ and } \frac{\partial^2 \boldsymbol{q}}{\partial x^2} = \frac{\boldsymbol{Q}_{i+1}^n + \boldsymbol{Q}_{i-1}^n - 2\boldsymbol{Q}_i^n}{\Delta x^2}.$$

We arrive at the Lax–Wendroff method, which is second-order accurate:

$$\boldsymbol{Q}_i^{n+1} = \boldsymbol{Q}_i^n - \frac{\Delta t}{2\Delta x}\boldsymbol{A}\cdot(\boldsymbol{Q}_{i+1}^n - \boldsymbol{Q}_{i-1}^n) + \frac{\Delta t^2}{2\Delta x^2}\boldsymbol{A}^2\cdot(\boldsymbol{Q}_{i+1}^n + \boldsymbol{Q}_{i-1}^n - 2\boldsymbol{Q}_i^n), \tag{2.93}$$

which leads to the following flux function

$$\boldsymbol{F}_{i-1/2}^n = \frac{1}{2}\boldsymbol{A}\cdot(\boldsymbol{Q}_i^n + \boldsymbol{Q}_{i-1}^n) - \frac{\Delta t}{2\Delta x}\boldsymbol{A}^2\cdot(\boldsymbol{Q}_i^n - \boldsymbol{Q}_{i-1}^n). \tag{2.94}$$

Introducing the negative and positive parts of $\boldsymbol{A}$ along with its absolute part (see § 2.3.2):

$$\boldsymbol{A} = \boldsymbol{A}^+\boldsymbol{A}^- \text{ and } |\boldsymbol{A}| = \boldsymbol{A}^+ - \boldsymbol{A}^-,$$

we can transform the sum $\boldsymbol{A}\cdot(\boldsymbol{Q}_i^n + \boldsymbol{Q}_{i-1}^n)$ into

$$\begin{aligned}\boldsymbol{A}\cdot(\boldsymbol{Q}_i^n + \boldsymbol{Q}_{i-1}^n) =& \boldsymbol{A}^+\cdot\boldsymbol{Q}_{i-1}^n + \boldsymbol{A}^-\cdot\boldsymbol{Q}_i^n + \boldsymbol{A}^-\cdot\boldsymbol{Q}_{i-1}^n + \boldsymbol{A}^+\cdot\boldsymbol{Q}_i^n,\\ &= \boldsymbol{A}^+\cdot\boldsymbol{Q}_{i-1}^n + \boldsymbol{A}^-\cdot\boldsymbol{Q}_i^n + (\boldsymbol{A}^+ - |\boldsymbol{A}|)\cdot\boldsymbol{Q}_{i-1}^n + (\boldsymbol{A}^+ + |\boldsymbol{A}|)\cdot\boldsymbol{Q}_i^n,\\ &= 2\boldsymbol{A}^+\cdot\boldsymbol{Q}_{i-1}^n + 2\boldsymbol{A}^-\cdot\boldsymbol{Q}_i^n + |\boldsymbol{A}|\cdot(\boldsymbol{Q}_i^n - \boldsymbol{Q}_{i-1}^n).\end{aligned}$$

We can then express the flux function (2.94) in the more traditional form, which makes the first-order flux correction apparent:

$$\boldsymbol{F}^n_{i-1/2} = \boldsymbol{A}^+ \cdot \boldsymbol{Q}^n_{i-1} + \boldsymbol{A}^- \cdot \boldsymbol{Q}^n_i + \frac{1}{2}|\boldsymbol{A}| \cdot \left(\boldsymbol{1} - \frac{\Delta t}{\Delta x}|\boldsymbol{A}|\right) \cdot (\boldsymbol{Q}^n_i - \boldsymbol{Q}^n_{i-1}). \qquad (2.95)$$

Like for the scalar case, we would like to limit the flux correction wherever it is needed. This limitation cannot be applied abruptly to the vector $\Delta \boldsymbol{Q}_{i-1/2} = \boldsymbol{Q}^n_i - \boldsymbol{Q}^n_{i-1}$ itself, but to its components along the wave propagation direction $\Delta \boldsymbol{w}_k$. To that end, we decompose $\Delta \boldsymbol{Q}_{i-1/2}$ in the base $(\boldsymbol{w}_k)_{1\leq k\leq n}$

$$\Delta \boldsymbol{Q}_{i-1/2} = \sum_{k=1}^{m} \alpha^k_{i-1/2} \boldsymbol{w}_k,$$

and we then apply the limiter function ϕ (which can be any function used in the scalar case) to each component

$$\hat{\alpha}^k_{i-1/2} = \phi(\theta^k_{i-1/2}),$$

where the function θ examines the degree of smoothness

$$\theta^k_{i-1/2} = \frac{\theta^k_{I-1/2}}{\theta^k_{i-1/2}} \text{ with } I = \left\{ \begin{array}{ll} i-1 & \text{if } \lambda_k > 0, \\ i & \text{if } \lambda_k < 0 \end{array} \right. .$$

To sum up, the Lax–Wendroff method leads to a flux function of the form

$$\boldsymbol{F}^n_{i-1/2} = \boldsymbol{A}^+ \cdot \boldsymbol{Q}^n_{i-1} + \boldsymbol{A}^- \cdot \boldsymbol{Q}^n_i + \hat{\boldsymbol{F}}^n_{i-1/2}, \qquad (2.96)$$

where the correction term is based on a flux-limiter function

$$\hat{\boldsymbol{F}}^n_{i-1/2} = \frac{1}{2}|\boldsymbol{A}| \cdot \left(\boldsymbol{1} - \frac{\Delta t}{\Delta x}|\boldsymbol{A}|\right) \cdot \sum_{k=1}^{m} \hat{\alpha}^k_{i-1/2} \boldsymbol{w}_k = \frac{1}{2}\sum_{k=1}^{m} |\lambda_k| \left(1 - \frac{\Delta t}{\Delta x}|\lambda_k|\right) \hat{\alpha}^k_{i-1/2} \boldsymbol{w}_k. \qquad (2.97)$$

Alternative 1

Let us first assume that the time step Δt satisfies the CFL condition, or in other words, the waves from adjacent interfaces $x_{i\pm1/2}$ cross the line t_{n+1} inside the interval $[x_{i-3/2}, x_{i+3/2}]$. Let us introduce the variable change (see § 1.1.4)

$$\boldsymbol{s} = \boldsymbol{L} \cdot \boldsymbol{q}$$

We use the same reasoning as in § 2.8.1. Let us make a second-order expansion of $\boldsymbol{s}(x, t + \Delta t)$ and make use of Eq. (2.90) to link time and space derivatives:

$$\begin{aligned} \boldsymbol{s}(x, t_{n+1} - \Delta t) &= \boldsymbol{s}(x, t_{n+1}) - \Delta t \frac{\partial \boldsymbol{s}}{\partial t}(x, t_{n+1}) + \frac{1}{2}\Delta t^2 \frac{\partial^2 \boldsymbol{s}}{\partial t^2}(x, t_{n+1}) + O(\Delta t^2) \\ &= \boldsymbol{s}(x, t_{n+1}) - \Delta t \boldsymbol{\Lambda} \cdot \frac{\partial \boldsymbol{s}}{\partial x}(x, t_{n+1}) - \frac{1}{2}\Delta t^2 \boldsymbol{\Lambda}^2 \cdot \frac{\partial^2 \boldsymbol{s}}{\partial x^2}(x, t_{n+1}) \end{aligned}$$

Integration of this equation over $[x_{i-1/2}, x_{i+1/2}]$ leads to

$$\boldsymbol{L}. \cdot (\boldsymbol{Q}^{n+1}_i - \boldsymbol{Q}^n_i) = \frac{\Delta t}{\Delta x} \boldsymbol{\Lambda} \cdot \int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial \boldsymbol{s}}{\partial x}(x, t_{n+1}) \mathrm{d}x + \frac{1}{2}\frac{\Delta t^2}{\Delta x} \boldsymbol{\Lambda}^2 \cdot \int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial^2 \boldsymbol{s}}{\partial x^2}(x, t_{n+1}) \mathrm{d}x. \qquad (2.98)$$

We can express $\boldsymbol{s}(x, t_{n+1})$ as a function of what happened at time t_n:

$$\boldsymbol{s}(x, t_{n+1}) = \boldsymbol{L} \cdot \tilde{\boldsymbol{q}}(x - \lambda_k \Delta t, t) = \sum_{k=1}^{m} q_k(x - \lambda_k \Delta t, t) \boldsymbol{v}_k.$$

We can express the integral terms of Eq. (2.98) as

$$\int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial}{\partial x}\boldsymbol{s}(x,t_{n+1})\mathrm{d}x = \sum_{k=1}^{m}\boldsymbol{v}_k\int_{x_{i-1/2}}^{x_{i+1/2}}\frac{\partial}{\partial x}\tilde{q}_k(x-\lambda_k\Delta t,t_n)\mathrm{d}x,$$
$$= \sum_{k=1}^{m}\boldsymbol{v}_k\int_{x_{i-1/2}-\lambda_k\Delta t}^{x_{i+1/2}-\lambda_k\Delta t}\frac{\partial}{\partial x}\tilde{q}_k(x,t_n)\mathrm{d}x. \tag{2.99}$$

Given that the function $\tilde{q}_k$ is discontinuous, the jumps are

$$[\![\tilde{q}_k]\!] = Q^n_{i,k} - Q^n_{i-1,k} - \frac{\Delta x}{2}(a_{i-1,k}+a_{i,k}) \text{ and } [\![\tilde{q}'_k]\!] = a_{i,k}-a_{i-1,k},$$

for $\lambda_k > 0$, whereas for $\lambda_k < 0$ we have:

$$[\![\tilde{q}_k]\!] = Q^n_{i+1,k} - Q^n_{i,k} - \frac{\Delta x}{2}(a_{i+1,k}+a_{i,k}) \text{ and } [\![\tilde{q}'_k]\!] = a_{i+1,k}-a_{i,k},$$

We deduce that for $\lambda_k > 0$, we have

$$\int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial}{\partial x}\boldsymbol{s}(x,t_{n+1})\mathrm{d}x = \sum_{k=1}^{m}\left(Q^n_{i,k}-Q^n_{i-1,k}+(a_{i,k}-a_{i-1,k})\frac{\Delta x}{2}-(a_{i,k}-a_{i-1,k})\lambda_k\Delta t\right)\boldsymbol{v}_k, \tag{2.100}$$

whereas for $\lambda_k < 0$, we have

$$\sum_{k=1}^{m}\left(Q^n_{i+1,k}-Q^n_{i,k}+(a_{i,k}-a_{i+1,k})\frac{\Delta x}{2}+(a_{i,k}-a_{i+1,k})\lambda_k\Delta t\right)\boldsymbol{v}_k. \tag{2.101}$$

For the second derivative, we have

$$\int_{x_{i-1/2}}^{x_{i+1/2}} \frac{\partial^2}{\partial x^2}\boldsymbol{s}(x,t_{n+1})\mathrm{d}x = \sum_{k=1}^{m}(a_{i,k}-a_{i-1,k})\,\boldsymbol{v}_k \text{ for } \lambda_k>0, \tag{2.102}$$
$$= \sum_{k=1}^{m}(a_{i+1,k}-a_{i,k})\,\boldsymbol{v}_k \text{ for } \lambda_k<0, \tag{2.103}$$

By substituting Eqs. (2.100)–(2.102) into Eq. (2.98), we find that for $\lambda_k > 0$, the second-order (in time) approximation of Q^{n+1}_i is given by:

$$Q^{n+1}_{i,k} = Q^n_{i,k} - \frac{\lambda_k\Delta t}{\Delta x}(Q^n_{i,k}-Q^n_{i-1,k}) - \frac{1}{2}\frac{\lambda_k\Delta t}{\Delta x}(\Delta x-\lambda_k\Delta t)(a_{i,k}-a_{i-1,k}),$$

while for $\lambda_k < 0$, we have

$$Q^{n+1}_{i,k} = Q^n_{i,k} - \frac{\lambda_k\Delta t}{\Delta x}(Q^n_{i+1,k}-Q^n_{i,k}) + \frac{1}{2}\frac{\lambda_k\Delta t}{\Delta x}(\Delta x+\lambda_k\Delta t)(a_{i+1,k}-a_{i,k}).$$

We can provide a vector form

$$\boldsymbol{Q}^{n+1}_i = \boldsymbol{Q}^n_i - \frac{\Delta t}{\Delta x}\left(\boldsymbol{F}^n_{i+1/2}-\boldsymbol{F}^n_{i-1/2}\right), \tag{2.104}$$

where the flux is

$$\boldsymbol{F}^n_{i-1/2} = \boldsymbol{A}^+\cdot\boldsymbol{Q}^n_{i-1} + \boldsymbol{A}^-\cdot\boldsymbol{Q}^n_i + \hat{\boldsymbol{F}}^n_{i-1/2} \tag{2.105}$$

and the flux correction $\hat{\boldsymbol{F}}^n_{i-1/2}$ is

$$\boldsymbol{F}^n_{i-1/2} = \frac{1}{2}|\boldsymbol{A}^+|\cdot\left(1-|\boldsymbol{A}|\frac{\Delta t}{\Delta x}\right)\cdot(\Delta x\boldsymbol{a}_{i-1/2}) + \frac{1}{2}|\boldsymbol{A}^-|\cdot\left(1-|\boldsymbol{A}|\frac{\Delta t}{\Delta x}\right)\cdot(\Delta x\boldsymbol{a}_{i+1/2}). \tag{2.106}$$

If we select the upwind slope $\Delta x\boldsymbol{a}_{i-1/2} = \Delta\boldsymbol{Q}_{i-1/2}$, then the flux function takes the form

$$\boldsymbol{F}^n_{i-1/2} = \frac{1}{2}|\boldsymbol{A}^+|\cdot\left(1-|\boldsymbol{A}|\frac{\Delta t}{\Delta x}\right)\cdot\Delta\boldsymbol{Q}_{i-1/2} + \frac{1}{2}|\boldsymbol{A}^-|\cdot\left(1-|\boldsymbol{A}|\frac{\Delta t}{\Delta x}\right)\cdot\Delta\boldsymbol{Q}_{i+1/2} \tag{2.107}$$

This form shows differences with the Lax-Wendroff expression (2.95) in that it depends on the propagation direction.

Alternative 2

In § 2.8.1, we presented Alternative 3, where the Sharpclaw method was outlined. Generalising Eq. (2.86) leads to a system of coupled ordinary differential equations:

$$\frac{\partial \boldsymbol{Q}_i}{\partial t} = -\frac{1}{\Delta x}\left(\boldsymbol{A}^+ \cdot \Delta \boldsymbol{q}_{i-1/2} + \boldsymbol{A}^- \cdot \Delta \boldsymbol{q}_{i+1/2} + \boldsymbol{A}\cdot \Delta \boldsymbol{q}_i\right) \text{ for } 1 \le i \le n. \tag{2.108}$$

where $\Delta \boldsymbol{q}_{i-1/2} = \boldsymbol{q}^+_{i-1/2} - \Delta \boldsymbol{q}^l_{i-1/2}$ and $\Delta \boldsymbol{q}_i = \boldsymbol{q}^-_{i+1/2} - \boldsymbol{q}^+_{i+1/2}$ (see Fig. 2.9 for the notation).

Note that this equation can be obtained heuristically in a quite general way. The time-marching algorithm is

$$\boldsymbol{Q}_i^{n+1} = \boldsymbol{Q}_i^n - \frac{\Delta t}{\Delta x}\left(\boldsymbol{F}^n_{i+1/2} - \boldsymbol{F}^n_{i-1/2}\right), \tag{2.109}$$

and in the limit of $\Delta t \to 0$, we have:

$$\frac{\partial \boldsymbol{Q}_i}{\partial t} = -\frac{1}{\Delta x}\left(\boldsymbol{F}_{i+1/2} - \boldsymbol{F}_{i-1/2}\right), \tag{2.110}$$

where the flux are given by

$$\boldsymbol{F}_{i-1/2} = \boldsymbol{A}\cdot \boldsymbol{q}^{\downarrow}_{i-1/2}.$$

where $\boldsymbol{q}^{\downarrow}_{i-1/2}$ is the value of $\boldsymbol{q}$ along the interface $x = x_{i-1/2}$. At short times $\delta t \ll 1$, this interface value is

$$\boldsymbol{q}^{\downarrow}_{i-1/2}(\delta t) = \boldsymbol{q}^0_{i-1/2} + \delta t \partial_t \boldsymbol{q}^{\downarrow}_{i-1/2}(0) + O(\delta t^2),$$

where the leading-order term $\boldsymbol{q}^0_{i-1/2}$ is the solution to the Riemann problem where the left and right states are $\boldsymbol{q}^-_{i-1/2}$ and $\boldsymbol{q}^+_{i-1/2}$ (see Eq. (1.35)):

$$\boldsymbol{q}^0_{i-1/2} = \boldsymbol{q}^+_{i-1/2} - \sum_{k:\ \lambda_k>0} \boldsymbol{W}^k_{i-1/2}.$$

Similarly, at the interface $x = x_{i-1/2}$, we have (see Eq. (1.36)):

$$\boldsymbol{q}^0_{i+1/2} = \boldsymbol{q}^-_{i+1/2} + \sum_{k:\ \lambda_k<0} \boldsymbol{W}^k_{i+1/2}.$$

To leading order in time, the interfaces fluxes are thus:

$$\begin{aligned}\boldsymbol{F}_{i-1/2} =& \boldsymbol{A}\cdot\left(\boldsymbol{q}^+_{i-1/2} - \sum_{k:\ \lambda_k>0} \boldsymbol{W}^k_{i-1/2}\right) = \boldsymbol{A}\cdot \boldsymbol{q}^+_{i-1/2} - \boldsymbol{A}^+\cdot \Delta \boldsymbol{q}_{i-1/2},\\ \boldsymbol{F}_{i+1/2} =& \boldsymbol{A}\cdot\left(\boldsymbol{q}^-_{i+1/2} + \sum_{k:\ \lambda_k<0} \boldsymbol{W}^k_{i+1/2}\right) = \boldsymbol{A}\cdot \boldsymbol{q}^-_{i+1/2} + \boldsymbol{A}^-\cdot \Delta \boldsymbol{q}_{i-1/2}.\end{aligned}$$

Taking the flux difference leads to the system of equations (2.108).

2.8.3 Extension to nonlinear equations

High-resolutions methods developed for linear systems can be extended to nonlinear systems. This extension does cause trouble when the solutions involve shock waves, but may be more tricky when they involve rarefaction waves, in particular transonic waves. They take the form

$$\boldsymbol{Q}_i^{n+1} = \boldsymbol{Q}_i^n - \frac{\Delta t}{\Delta x}\left(\boldsymbol{\mathcal{A}}^-\Delta \boldsymbol{Q}^n_{i+1/2} + \boldsymbol{\mathcal{A}}^+\Delta \boldsymbol{Q}^n_{i-1/2}\right) - \frac{\Delta t}{\Delta x}\left(\hat{\boldsymbol{F}}^n_{i+1/2} - \hat{\boldsymbol{F}}^n_{i-1/2}\right), \tag{2.111}$$

where the operators $\boldsymbol{\mathcal{A}}^{\pm}$ are defined by Eqs. (2.46) and (2.46). When the solution does not involve a transonic wave, it can be computed using Eqs. (2.48) and (2.49)

$$\boldsymbol{\mathcal{A}}^{-}\Delta\boldsymbol{Q}_{i+1/2}^{n} = \sum_{k=1}^{M_w} s_{k,i+1/2}^{-}\boldsymbol{W}_{k,i+1/2}^{n}, \tag{2.112}$$

$$\boldsymbol{\mathcal{A}}^{+}\Delta\boldsymbol{Q}_{i-1/2}^{n} = \sum_{k=1}^{M_w} s_{k,i-1/2}^{+}\boldsymbol{W}_{k,i-1/2}^{n}, \tag{2.113}$$

where M_w is the number of waves (in most cases $M_w = m$) and $s_{i-1/2}^{k}$ is speed of the kth wave at $x_{i-1/2}$. An entropy fix is needed to handle transonic waves properly.

Equation (2.111) also involves the flux correction $\hat{\boldsymbol{F}}_{i+1/2}^{n}$:

$$\hat{\boldsymbol{F}}_{i+1/2}^{n} = \frac{1}{2}\sum_{k=1}^{M_w}|s_{i-1/2}^{k}|\left(1-\frac{\Delta t}{\Delta x}|s_{i-1/2}^{k}|\right)\tilde{\boldsymbol{W}}_{i-1/2}^{k} \tag{2.114}$$

where $\tilde{\boldsymbol{W}}_{i-1/2}^{k}$ is the limited version of the kth wave $\boldsymbol{W}_{i-1/2}^{k}$ obtained by comparing this wave with the jump $\boldsymbol{W}_{I-1/2}^{k}$ in the upwind direction ($I = i-1$ is $s_{i-1/2}^{k} > 0$ and $I = i+1$ is $s_{i-1/2}^{k} < 0$) (LeVeque, 2002, see Chaps. 6 and 15). Here arises a problem specific to nonlinear systems. Generally, the vectors $\boldsymbol{W}_{I-1/2}^{k}$ and $\boldsymbol{W}_{i-1/2}^{k}$ are not collinear, which makes it difficult to compare their components in the limiter function (LeVeque, 2002, see Sec. 9.13). A strategy used in Clawpack to solve this issue is to project the vector $\boldsymbol{W}_{I-1/2}^{k}$ onto $\boldsymbol{W}_{i-1/2}^{k}$ and compare their components.

2.9 Source term

We are concerned with the following hyperbolic equation with a source term:

$$\frac{\partial}{\partial t}\boldsymbol{q} + \frac{\partial}{\partial x}\boldsymbol{f}(\boldsymbol{q}) = \boldsymbol{S}, \tag{2.115}$$

where $\boldsymbol{q}$ is a vector with m components representing the unknowns, $\boldsymbol{f}$ is the flux function, and $\boldsymbol{S}$ is the source term, possibly a function of $\boldsymbol{q}$, its spatial derivatives, and position x.

2.9.1 Fractional-step method

One strategy to solve Eq. (2.115) is the fractional-step method, which involves splitting the problem into subproblems:

- Solving the homogenous equation $\partial_t\boldsymbol{q} + \partial_x\boldsymbol{f}(\boldsymbol{q}) = 0$.
- Updating the solution by solving the ordinary differential equation

$$\frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}t} = \boldsymbol{S}(\boldsymbol{q}).$$

This method is first-order accurate (in time) regardless of the accuracy of each method used for solving the subproblems. For instance, if we want to solve the linear advection equation with a source term

$$\frac{\partial}{\partial t}q + \bar{u}\frac{\partial}{\partial x}q = S(q), \tag{2.116}$$

then combining the upwind method for the first subproblem and the forward Euler method for the second one leads to consider the following set of equations involving an intermediate state Q_i^*:

$$\begin{aligned} Q_i^* &= Q_i^n - \frac{\bar{u}\Delta t}{\Delta x}(Q_i^n - Q_{i-1}^n) \\ Q_i^{n+1} &= Q_i^* + \Delta t S(Q_i^*). \end{aligned}$$

2.9.2 Strang splitting method

This method used an intermediate time step $\Delta t/2$, we start by solving the first subproblem with this time step to obtain the intermediate state $\boldsymbol{Q}^*$. We then solve the second subproblem over the full time step Δt, with $\boldsymbol{Q}^*$ as the initial value and obtain $\boldsymbol{Q}^{**}$, and we eventually return to the first subproblem and solve it again over the time step $\Delta t/2$ with $\boldsymbol{Q}^{**}$ as the initial value. This method is second-order accurate (in time).

If we take the example of linear advection (2.116) again, these three steps leads to the following scheme:

$$\begin{aligned} Q_i^* &= Q_i^n - \frac{\bar{u}\Delta t}{2\Delta x}(Q_i^n - Q_{i-1}^n) \\ Q_i^{**} &= Q_i^* + \Delta t S(Q_i^*), \\ Q_i^{n+1} &= Q_i^n - \frac{\bar{u}\Delta t}{2\Delta x}(Q_i^* - Q_{i-1}^*). \end{aligned}$$

2.9.3 Stiff source terms and implicit methods

When the second subproblem involves a *stiff*[3] differential equation, it may be necessary to use an implicit method for solving the second subproblem. For simple source terms, the trapezoidal method is usually efficient:

$$Q_i^{n+1} = Q_i^* + \frac{\Delta t}{2}(S(Q_i^*) + S(Q_i^{n+1}). \tag{2.117}$$

This method usually does not generate insurmountable problems since there is no spatial coupling, and thus Eq. (2.117) can be solved in one time step. The Bank–Coughran–Fichtner method (also called *TR-BDF2 method*) offers better performances. It is based on a two-stage Runge–Kutta approach:

$$\begin{aligned} Q_i^{**} &= Q_i^* + \frac{\Delta t}{2}(S(Q_i^*) + S(Q_i^{**}) \\ Q_i^{n+1} &= \frac{1}{3}\left(4Q_i^{**} - Q_{i-1}^* + \Delta t S(Q_i^{n+1})\right). \end{aligned}$$

When the source term involves high-order spatial derivatives of $\boldsymbol{q}$ (e.g., diffusive terms), the implicit method usually requires more work. For instance, if the source term involves diffusion

$$\frac{\partial}{\partial t}q + \bar{u}\frac{\partial}{\partial x}q = \mu\frac{\partial^2}{\partial x^2}q, \tag{2.118}$$

with μ a constant, then the spatial discretisation of the second-order derivative involves the neighbouring cells

$$\frac{\partial^2}{\partial x^2}q \approx \frac{Q_{i-1} - 2Q_i + Q_{i+1}}{\Delta x^2},$$

[3]Loosely speaking, stiff equations may have solutions that exhibit significant variations over short distances. Explicit methods are usually not efficient for handling these equations. Indeed, explicit methods usually requires to select time steps $\Delta t = O(\Delta x^2)$ to be stable.

and therefore the trapezoidal equation (2.117) turns into the Crank-Nicolson method

$$Q_i^{n+1} = Q_i^* + \mu\frac{\Delta t}{2}\left(\frac{Q_{i-1}^{n+1} - 2Q_i^{n+1} + Q_{i+1}^{n+1}}{\Delta x^2} + \frac{Q_{i-1}^* - 2Q_i^* + Q_{i+1}^*}{\Delta x^2}\right). \tag{2.119}$$

This method can be still improved by using the TR-BDF2 method rather than the trapezoidal equation.

2.9.4 *Well-balanced algorithms*

The hyperbolic equation (2.118) with a nonzero source term admits a steady-state solution $\boldsymbol{q}_{ss}$, which is the solution to the ordinary differential equation

$$\frac{\mathrm{d}}{\mathrm{d}x}\boldsymbol{f}(\boldsymbol{q}_{ss}) = \boldsymbol{S}(\boldsymbol{q}_{ss}). \tag{2.120}$$

Time-marching algorithms may find the right steady-state solution, but as the advection and sources are handled separately in two distinct subproblems, time-marching algorithms may not converge, but oscillate in time around the steady-state solution.

This issue has led to the development of *well-balanced algorithms*. An algorithm is said to be well-balanced if it is able to provide not only the time-dependent solution to a boundary initial value problem, but also the steady-state solution.

2.10 Boundary conditions

There is a large literature on boundary conditions for hyperbolic equations: existence and uniqueness of solutions, typology of boundary conditions, numerical implementation of boundary conditions, etc. We will not deal with all these issues here, but we will assume that the (physical) boundary conditions of the domain are known and that we are trying to write them in a numerical form consistent with the finite-volume method. In Clawpack, the strategy adopted is to extend the computational domain by using ghost cells (see 2.11). In the general case, we need $m_{bc} = 2$ ghost cells for each boundary so that we can use the high-resolution methods seen in § 2.8.

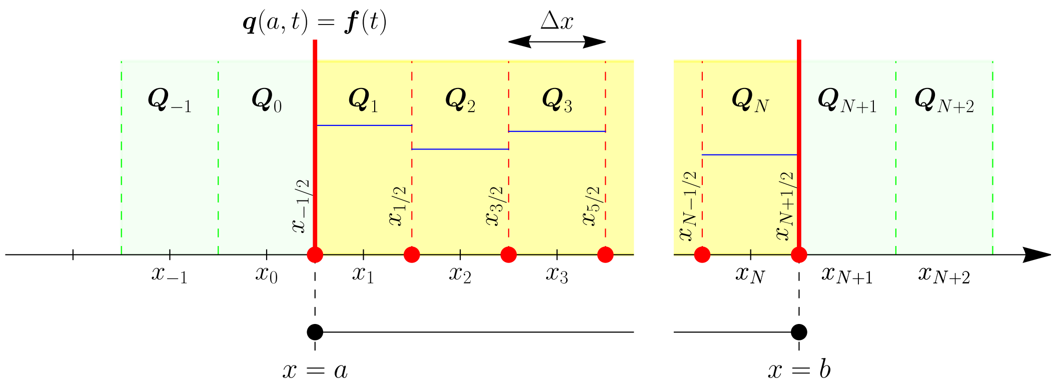


Figure 2.11 Computational grid and its extension to include ghost cells. The yellow-coloured area represents the computational domain $[a, b]$, while the green-coloured areas show the extended domain for handling the boundary conditions.

2.10.1 Periodic conditions

Periodic conditions imply that what goes out of the interval at $x = b$ must enter from the left into $x = a$. When $m_{bc} = 2$ ghost cells are used, this condition on the left of the computational domain can be written as:

$$\boldsymbol{Q}_{-1} = \boldsymbol{Q}_{N-1} \text{ and } \boldsymbol{Q}_0 = \boldsymbol{Q}_N,$$

while on its right, it writes

$$\boldsymbol{Q}_{N+1} = \boldsymbol{Q}_1 \text{ and } \boldsymbol{Q}_{N+2} = \boldsymbol{Q}_2.$$

2.10.2 Extrapolation

In many (if not most) problems, the computational domain includes only a portion of the physical space in which the physical process studied takes place. Its extension is thus often arbitrary, and the boundaries conditions required mathematically and numerically to solve the problem do are not physical, but artificial. We refer to these artificial boundaries as *transmissive*, *absorbing* or *non-reflecting boundary conditions* (LeVeque, 2002; Toro, 2009). We would like the waves to cross these boundaries, when passing out of the computational domain, without generating disturbances or spurious reflections. Problems may arise whenever the characteristic speeds at the boundary take positive and negative values (like in subcritical flow problems), implying that there are incoming waves whose characteristics are fixed by what happens outside the computational domain. The simplest method to set appropriate boundary conditions is to assume that the solution behaves smoothly near boundaries, and thus its values can be obtained by extrapolating those determined in the computational domain.

We could determine the value $\boldsymbol{Q}_{N+1}$ by assuming that the solution can be continued by mere extrapolation:

$$\boldsymbol{Q}_{N+1} = \boldsymbol{Q}_N + \frac{\boldsymbol{Q}_N - \boldsymbol{Q}_{N-1}}{\Delta x}\Delta x = 2\boldsymbol{Q}_N - \boldsymbol{Q}_{N-1},$$

which is the discretisation of the first-order expansion $q(x + \mathrm{d}x) = q(x) + q'(x)\mathrm{d}x$. First-order extrapolation can lead to stability problems and is generally not recommended (LeVeque, 2002). Instead, a zero-order extrapolation is a simpler technique:

$$\boldsymbol{Q}_{N+1} = \boldsymbol{Q}_N \text{ and } \boldsymbol{Q}_{N+2} = \boldsymbol{Q}_N$$

This strategy is often effective, but sometimes it may fail to produce the right result.

2.10.3 Solid Wall

One-dimensional equations

Let us consider that at the left boundary $x = a$, there is a solid wall and q vanishes at this point:

$$q(a, t) = 0.$$

For one-dimensional problem associated with a positive eigenvalue at $x = a$ (incoming wave with velocity $\lambda(a, t) > 0$), then we just have to set

$$Q_0 = 0.$$

In the opposite case (outgoing wave with velocity $\lambda(a, t) < 0$), then there is no inflow, and we can use a zero-order extrapolation:

$$Q_0 = Q_1.$$

Systems of linear equations

For multidimensional problems, more work is required. We will outline a technique for implementing boundary conditions representing the wall condition, which is effective for the Euler and Saint-Venant equations as well as a number of systems of equations in fluid mechanics. Let us assume that one of the unknowns q^2 is velocity u while the other q^1 is pressure p (or depth h). LeVeque (2002) noted that when extending the computational domain to the left by setting

$$u(x,t) = -u(-x,t) \text{ and } p(x,t) = p(-x,t) \text{ for } x \leq a$$

then we have

$$u(a,t) = -u(-a,t)$$

and thus

$$u(a,t) = 0.$$

In that case, the ghost cells are defined as follows:

$$Q_0^1 = Q_1^1 \text{ and } Q_0^2 = -Q_1^2, \tag{2.121}$$

and

$$Q_{-1}^1 = Q_2^1 \text{ and } Q_{-1}^2 = -Q_2^2. \tag{2.122}$$

The result can be generalized to more complicated boundary conditions when the governing equations involve a (linearised) Jacobian matrix in the form

$$\boldsymbol{A} = \begin{bmatrix} u_a & p \\ q & u \end{bmatrix} \tag{2.123}$$

associated with eigenvalues $\lambda_1 = u_a - \sqrt{pq}$ and $\lambda_2 = u_a + \sqrt{pq}$, left and right eigenvectors

$$\boldsymbol{v}_1 = \frac{1}{2}\begin{bmatrix} -\sqrt{q/p} \\ 1 \end{bmatrix}, \boldsymbol{v}_2 = \frac{1}{2}\begin{bmatrix} \sqrt{q/p} \\ 1 \end{bmatrix}, \boldsymbol{w}_1 = \begin{bmatrix} -\sqrt{p/q} \\ 1 \end{bmatrix} \text{ and } \boldsymbol{w}_2 = \begin{bmatrix} \sqrt{p/q} \\ 1 \end{bmatrix},$$

where u_a, p, and q are constants. Typical examples include the shallow water and the acoustic equations. We would like to solve the Riemann problem at $x = a$ (see Fig. 2.12) and determine the intermediate state $\boldsymbol{Q}_*$.

From Eq. (1.36), we can connect the intermediate $\boldsymbol{Q}_* = (Q_*^1, Q_*^2)$ to the right state:

$$\boldsymbol{Q}_* = \boldsymbol{Q}_1 - \alpha_2 \boldsymbol{w}_2,$$

where α_2 is the coefficient satisfying:

$$\boldsymbol{\alpha} = (\alpha_1, \alpha_2) = \boldsymbol{L} \cdot \Delta\boldsymbol{Q} \text{ where } \Delta\boldsymbol{Q} = \boldsymbol{Q}_1 - \boldsymbol{Q}_0$$

Let us assume that there is no variation in the first component of $\boldsymbol{Q}_*$ (zero-order extrapolation):

$$Q_0^1 = Q_1^1 \Rightarrow \Delta Q^1 = 0.$$

Let us now focus on the second component of $\boldsymbol{Q}_*$ (carrying information on the velocity u):

$$Q_*^2 = Q_1^2 - \alpha_2 w_2^2 = Q_1^2 - \frac{1}{2}(Q_1^2 - Q_0^2). \tag{2.124}$$

If we want to impose the zero velocity at $x = a$, then we must set $Q_*^2 = 0$ and Eq. (2.124) tells us that

$$Q_1^2 - \frac{1}{2}(Q_1^2 - Q_0^2) = 0 \Rightarrow Q_0^2 = -Q_1^2$$

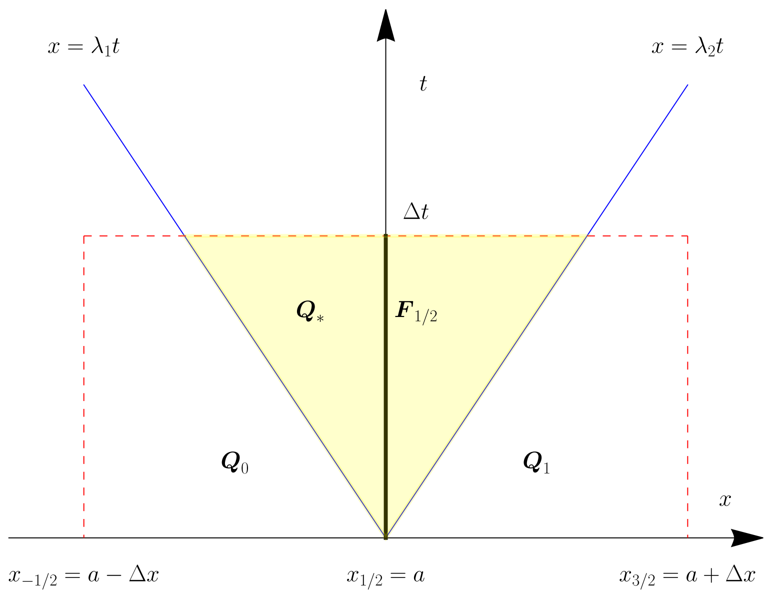


Figure 2.12 Riemann problem at the boundary $x = a$.

consistently with what is given by Eq. (2.121). If we now impose an oscillating wall as the left boundary condition

$$u(a,t) = U(t),$$

with U the wall velocity, then we must set $Q_*^2 = U$ and Eq. (2.124) tells us that

$$Q_1^2 - \frac{1}{2}(Q_1^2 - Q_0^2) = U \Rightarrow Q_0^2 = 2U - Q_1^2$$

Systems of nonlinear equations

We can establish a similar result when knowing the rarefaction wave's structure. Let us take the example of the Saint-Venant equations (see Fig. 2.13). We assume that the intermediate state $\boldsymbol{Q}_* = (h_*, h_*\bar{u}_*)$ is connected to the right state $\boldsymbol{Q}_1 = (h_1, h_1\bar{u}_1)$ by a 2-rarefaction wave (1.128):

$$\bar{u}_1 - 2(\sqrt{gh_1} - \sqrt{gh_*}) = \bar{u}_*,$$

and is connected to left right state $\boldsymbol{Q}_0 = (h_0, h_0\bar{u}_0)$ by a 1-rarefaction wave (1.128):

$$\bar{u}_0 + 2(\sqrt{gh_0} - \sqrt{gh_*}) = \bar{u}_*.$$

If we add these two equations, then we obtain

$$\bar{u}_1 + \bar{u}_0 + 2\sqrt{gh_0} - 2\sqrt{gh_1} = 2\bar{u}_*.$$

If we set $h_0 = h_1$, then we must define the boundary condition for $\bar{u}_0$

$$\bar{u}_0 = 2\bar{u}_* - \bar{u}_1.$$

If the boundary involves a solid wall moving at velocity $U(t)$, then $\bar{u}_* = U$, and thus we find again Eq. (2.121)

$$\bar{u}_0 = 2U - \bar{u}_1.$$

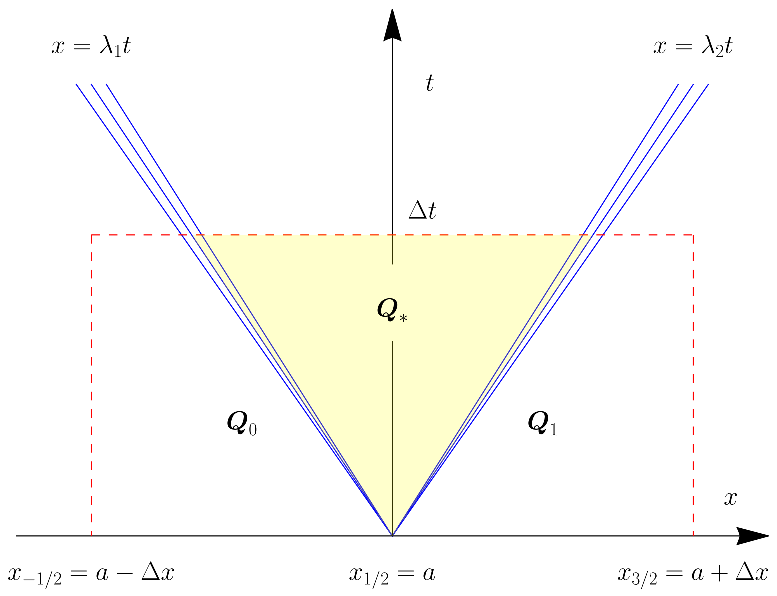


Figure 2.13 Riemann problem at the boundary $x = a$ with an intermediate state separated from the left and right states by rarefaction waves.

2.10.4 *Inflow boundary conditions*

One-dimensional equations

Let us consider that we impose a Dirichlet boundary condition at the left boundary (see Fig. 2.14):

$$q(a, t) = g(t). \tag{2.125}$$

We assume that the problem is linear or linearised so that information is propagated at constant velocity λ and the characteristic curves are straight lines $x = x_0 + \lambda t$. Setting the boundary value Q_0 amounts to determining the integral

$$Q_0^n = \frac{1}{\Delta x}\int_{a-\Delta x}^{a} q(x, t_n)\mathrm{d}x. \tag{2.126}$$

For any characteristic $x = x_0 + \lambda(t - t_n)$ with $a - \Delta x < x_0 < a$, the characteristic curve crosses the line $x = a$ at time

$$a = x_0 + \lambda(t - t_n) \Rightarrow t = t_n + \frac{a - x_0}{\lambda}.$$

The value of q is constant and given by $g(t)$. We can put the integral (2.126) in the following form:

$$Q_0^n = \frac{1}{\Delta x}\int_{a-\Delta x}^{a} g\left(t_n + \frac{a - x_0}{\lambda}\right)\mathrm{d}x_0.$$

We make the following change of variable

$$\tau = t_n + \frac{a - x_0}{\lambda} \Rightarrow \mathrm{d}\tau = \frac{\mathrm{d}x_0}{\lambda}.$$

This shows that the value of Q_0^n is

$$Q_0^n = \frac{\lambda}{\Delta x}\int_{t_n}^{t_n + \Delta x/\lambda} g\left(\tau\right)\mathrm{d}\tau.$$

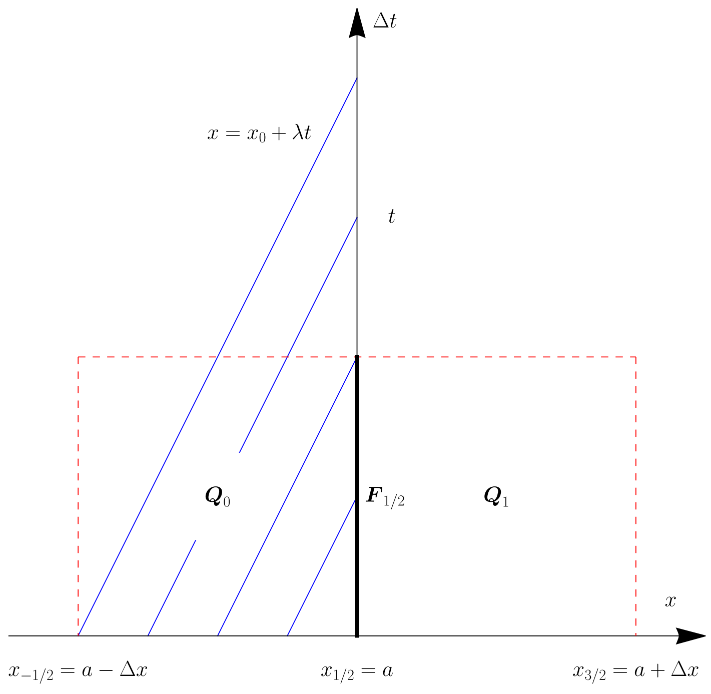


Figure 2.14 Dirichlet condition $q(a, t) = g(t)$ imposed at $x = a$.

LeVeque (2002) suggests approximating this integral by using the mean value theorem (evaluated at the midpoint $t_n + \Delta x/(2\lambda)$):

$$Q_0^n = g\left(t_n + \frac{\Delta x}{2\lambda}\right). \tag{2.127}$$

Similarly we have:

$$Q_{-1}^n = g\left(t_n + \frac{3\Delta x}{2\lambda}\right). \tag{2.128}$$

2.11 Implementation in Clawpack

Clawpack is a Fortran-based library developed to solve hyperbolic partial differential equations in the form:

$$\kappa \frac{\partial}{\partial t}\boldsymbol{q} + \nabla \cdot \boldsymbol{f}(\boldsymbol{q}) = \boldsymbol{S}, \tag{2.129}$$

where κ is the capacity function (or constant), $\boldsymbol{q}$ the unknown, $\boldsymbol{f}$ the flux function, and $\boldsymbol{S}$ the source term.

2.11.1 *Clawpack installation*

Prerequisites

Linux is best suited to run the Clawpack library. Installation requires a few additional libraries (see www.clawpack.org/prereqs.html)

- Compiler: gfortran (available from most linux distributions) or ifort (which needs a license, free for academic activities).
- Python: version 3, scipy, numpy, pip, meson-python and git
- It can be useful to install anaconda. This environment makes it possible to manage the python packages and offers several functionalities like jupyter (a system of Python-based notebooks that can be read by a web browser), spyder (a scientific environment written for Python), R, and julia. Jupyter notebooks available from github can be read locally on the computer or via a web interface such as nbviewer.jupyter.org/.

Following the procedure with pip (see www.clawpack.org/installing.html) is usually the easy way to install Clawpack:

```
pip install --src=$HOME/clawpack_src --user --no-build-isolation -e \
    git+https://github.com/clawpack/clawpack.git@v5.9.2#egg=clawpack
```

Finally, it is necessary to edit the `.bashrc` file by providing the required environment variables

```
export CLAW=$HOME/clawpack-v5.9.2
export FC=gfortran
```

Build process

As of version 5.9.2, Clawpack used Meson-Python for the build process of the python modules. When modifying the clawpack working directory (including updates or use of earlier versions of Clawpack), the paths should be set properly by using the command (see /www.clawpack.org/installing_pip.html)

```
cd $CLAW
pip install --user --no-build-isolation -e ./
```

Makefile

Using the `openmp` library and specifying the number of threads, it is possible to make the Clawpack computations much faster (see www.clawpack.org/openmp.html). This can be implemented by setting the compiler flags in the prompt command:

```
export FFLAGS='-O2 -fopenmp'  # or hardwire FFLAGS in the Makefile
make new
export OMP_NUM_THREADS=2
```

or directly in the Makefile:

```
export FFLAGS='-O2 -fopenmp'  # or hardwire FFLAGS in the Makefile
make new
export OMP_NUM_THREADS=2
```

Similarly when using the Lapack or Blas library (linear algebra), it is possible to specify the appropriate compiler flag in the Makefile:

```
FFLAGS ?= -O2 -fopenmp
OMP_NUM_THREADS=4
```

The following commands are implemented (the complete list is also available by executing the command `make␣help`):

- `make␣new` makes a new compilation of everything.
- `make␣clobber` deletes all the intermediary files (*.o, *.mo, output).
- `make␣.output` runs the code. The results are by default placed in the subdirectory named `_output`.
- `make␣.plots` runs the code and plots the results using the python script `setplot`. The graphic files are placed in the subdirectory named `_plots`.
- `make␣.data` provides the data files using the python script `setrun`.

2.11.2 Legacy Clawpack

In its original form developed by Randall LeVeque, Clawpack has been based on a set of Fortran 77 routines (LeVeque, 2002).

The main programme was originally located in the file driver.f. This file allocated storage for the arrays used by Clawpack. This is now done automatically, and the user does not need to fill this file. This programme then calls claw1ez, which reads the file claw.data created by the python script setrun.py (it can be created by typing make .data).

The initial condition is contained in the file qinit.f. We should define the cell average value Q_i over the entire domain, but for a continuous function q, this average value is the value taken by q at x_i (cell midpoint): $Q_i = q(x_i)$.

The initial conditions are processed in the file bc1.f. The type of boundary conditions is prescribed in the file claw.data.

The Riemann solver is contained in the file rp1.f. The idea is to decompose any discontinuity into a set of waves $\boldsymbol{W}_k$ moving ar velocity s^k (see § 2.3):

$$\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1} = \sum_{k=1}^{m} \boldsymbol{W}_k$$

avec m is the wave number (which is usually equal to the dimension of the system). For Godunov's method, the value $\boldsymbol{Q}_i$ is updated as follows:

$$\boldsymbol{Q}_i^{n+1} = \boldsymbol{Q}_i^n - \frac{\Delta t}{\Delta x}(\boldsymbol{A}^+ \cdot \Delta\boldsymbol{Q}_{i-1/2} + \boldsymbol{A}^- \cdot \Delta\boldsymbol{Q}_{i+1/2}),$$

where we distinguish between the left-going wave (coming from the right endpoint $x_{i+1/2}$):

$$\boldsymbol{A}^+ \cdot \Delta\boldsymbol{Q}_{i+1/2} = \sum_k \min(s^k_{i+1/2}, 0)\boldsymbol{W}^k_{i+1/2},$$

and the right-going wave

$$\boldsymbol{A}^- \cdot \Delta\boldsymbol{Q}_{i+1/2} = \sum_k \max(s^k_{i-1/2}, 0)\boldsymbol{W}^k_{i-1/2},$$

The left-going wave is zero if $s^k_{i+1/2} > 0$ (because the wave moves only from right to left) and the right-going wave is zero if $s^k_{i-1/2} < 0$.

The Riemann solver needs two input data: the two arrays `ql` and `qr` related to the values of q on the left and right of every cell. High-resolution methods require further information. Note that for the Riemann solver at the interface $x_{i-1/2}$, we use the following notation for referring to cells $i-1$ and

i: `qr(i-1,:)`$= \boldsymbol{q}^r_{i-1/2}$ and `ql(i-1,:)`$= \boldsymbol{q}^l_{i-1/2}$, and in this notation, left and right refer to the left and right of the cell i or $i-1$, and not what happens relative to the interface.

The solver provides:

- the functions `amdq` (literally "a minus delta q", which is the vector $\boldsymbol{A}^- \cdot \Delta\boldsymbol{Q}_{i+1/2}$) and `apdq` (vecteur $\boldsymbol{A}^+ \cdot \Delta\boldsymbol{Q}_{i+1/2}$),
- `wave` (the wave $\boldsymbol{W}^k_{i-1/2}$), and
- `s` (the speed $s^k_{i-1/2}$).

Caveat. Note[4] that the Riemann problem at the interface $x_{i\text{-}1/2}$ between cells i-1 and i has the following input data (see Fig. 2.15):

- Left state: $q^R_{i\text{-}1}$ = qr(:, i-1).
- Right state: $q^L_{i\text{-}1}$ = ql(: ,i).

This notation is ambiguous since until now, q_l has been used to denote the left state of a Riemman problem, while q_r denotes its right state.

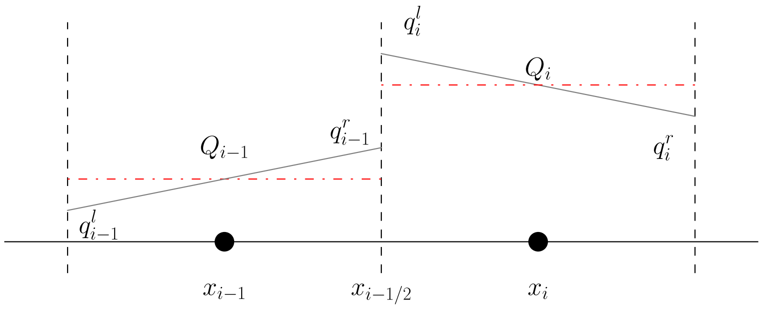


Figure 2.15 Notation for the Riemann problem in Clawpack.

There are many other routines, which are not always required. They are called by the main driver by default, but do not return anything. Among the most important:

- `setprob.f`: the routine `claw1ez` calls `setprob.f` before each execution, which makes it possible to initialise some parameters.
- `setaux.f`: the routine `claw1ez` calls the routine `setaux.f` before each execution to initialise the auxiliary variables (for instance, bed topography).
- `b4step1.f`: the routine `claw1` calls the routine `bc4step1.f` before each step to perform additional tasks.
- `src1.f`: if the equation involves a source term, this file is used to correct the solution to the homogenous equation.

[4]See www.clawpack.org/riemann.html.

2.11.3 AMRClaw

AMRClaw is the Clawpack version that provides Adaptive Mesh Refinement (AMR) capabilities in 1, 2 or 3 space dimensions (Berger & LeVeque, 1998). Depending on the mesh refinement criteria (which may be automatic or user-selected), the mesh may be locally refined to improve numerical accuracy. Numerical solutions are recorded as a series of nested grids. Computation is done using a coarse mesh when the solution behaves smoothly, and is done with finer resolution when this is locally required. See the AMRClaw webpage for further information.

GeoClaw, based on the Clawpack library and devoted to large-scale free-surface flows over topography, uses the AMR toolbox (George, 2006, 2008, 2011).

2.11.4 Pyclaw

Pyclaw is a python package that offers a convenient framework for pre- and post-processing information, interfacing and running Clawpack or Sharpclaw (Ketcheson *et al.*, 2012; Mandli *et al.*, 2016). It can call Fortran or Python routines. Interfaced with PyWENO and PETSc, Pyclaw provides extended functionality in terms of parallel computing (Ketcheson *et al.*, 2012). In particular, PyClaw can be run in parallel using the PETSc library.

PyClaw includes two types of solvers:

- Classic solvers are those originally developed in Clawpack 4 (algorithms for solving one- to three-dimensional problems).
- SharpClaw solvers are higher-order wave propagation routines using WENO reconstruction and Runge–Kutta integration, but reserved for one- and two-dimensional problems for the moment.

Solver initialisation takes one argument (the Riemann solver's name). For instance, if one selects `riemann.acoustics_1D` from the Riemann repository to solve one-dimensional acoustic problems, then one can initialise the solver as follows

```python
from clawpack import pyclaw
from clawpack import riemann
solver = pyclaw.ClawSolver1D(riemann.acoustics_1D)
```

and if one prefers the Sharpclaw solver

```python
solver = pyclaw.SharpClawSolver1D(riemann.acoustics_1D)
```

If the solver is defined in a script or in a function, then one must initialise the ClawSolver object and specify the number of equations and waves:

```python
solver = pyclaw.ClawSolver1D(solver_name)
solver.num_waves = 1
solver.num_eqn = 1
solver.kernel_language = 'Python'
```

When using a Clawpack solver from the Riemann repository, the code automatically determines the numbers of waves and equations. Fortran solvers can be used in python using the numpy command `f2py`. See www.clawpack.org/pyclaw/problem.html and /www.clawpack.org/riemann.html for further information to set up your own problem and your own Riemann solver.

It is also possible to call solvers written in fortran. To that end, the fortran file should first be compiled using the `numpy` command called `f2py` to wrap fortran files to python. This command compiles all sources and builds an extension module containing the wrappers. In a command prompt, this achieved using the command

```
f2py -c my_riemann_solver.f90 -m solver_name
```

or it can be done in python

```
python -m numpy.f2py my_riemann_solver.f90 -m solver_name
```

If successful, the command generates a file like `solver_name.cpython-39-x86_64-linux-gnu.so`. This file can then be imported in Pyclaw:

```
import solver_name
...
solver.kernel_language = 'Fortran'
solver = pyclaw.ClawSolver1D(solver_name)
```

The classic and SharpClaw solvers have two important differences:

- The classic solver involves the integral of the source term over a step (implemented using `solver.step_source`), whereas SharpClaw needs the instantaneous value of the source term (implemented using `solver.dq_src`).
- The list of options and their possible values differ from one variant to the other.

See www.clawpack.org/pyclaw/solvers.html for further information.

A PyClaw simulation provides a set of frames, each one being the solution at a given time (depending on the selected number of outputs and final time). By default, these frames are written in a directory named `_outdir` unless the option `claw.output_format` is set to `None`. If `claw.keep_copy` is set to `True`, then output frames are also saved in memory in the list `claw.frames`. The solution is available at any time from `claw.frames[i].q` where i the frame number ($i = 0$: initial condition, $i = -1$ final state). See www.clawpack.org/pyclaw/output.html for further information.

Pyclaw simulations involve Solution objects, which evolve as a result of the action of a Solver object depending on the Domain and State objects. A Pyclaw Solution comprises one or several Domain and State objects (see Fig. 2.16). Each State object inhabits a Grid, composed of Dimension objects that define the extents of the Domain. See www.clawpack.org/pyclaw/classes.html.

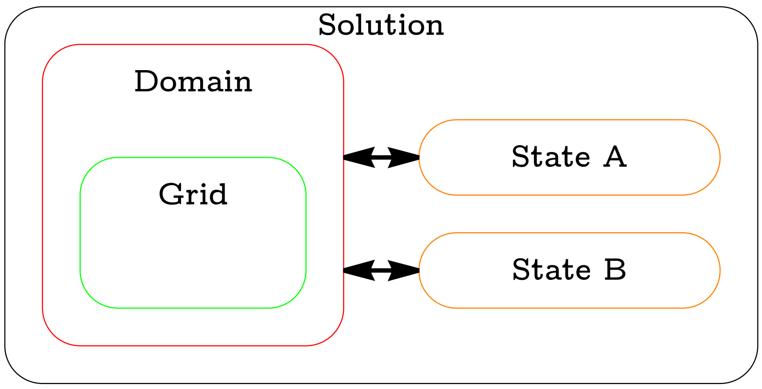


Figure 2.16 Pyclaw solution structure.

2.11.5 Development version

A read-only development version of Clawpack can be created (see www.clawpack.org/dev/developers.html) for those interested in testing the applications under development.

To clone clawpack from github, use the following where you to create your development folder

```
git clone https://github.com/clawpack/clawpack.git
cd clawpack
git submodule init
git submodule update
```

You can then update clawpack to the most recent version

```
source pull_all.sh
```

For instance, if you want to install dclaw, then write

```
git clone https://github.com/geoflows/dclaw
```

2.11.6 Implementation of boundary conditions in Clawpack

Classic Clawpack

In the classic version of Clawpack, the boundary conditions are handled in the file `bc1.f` (one-dimensional problems) or `bc2.f` (two-dimensional problems). A number of classic boundary conditions are proposed:

- extrapolation,
- periodic, or
- wall.

It is possible to include a custom boundary conditions. The type of boundary condition is specified in the `setrun.py` file. For instance, we want to use custom boundary conditions, we write:

```
clawdata.bc_lower[0] = 'user'   # at xlower
clawdata.bc_upper[0] = 'user'   # at xupper
```

In fortran, it is possible to use negative indices in arrays, and as a consequence, the notation is very close to the mathematical one: $\boldsymbol{Q}_{-1}$ is denoted by $q(i,-1)_{1\leq i\leq m}$. For instance, the period boundary condition is written as:

```
dimension q(meqn,1-mbc:mx+mbc)
do ibc=1,mbc
         do m=1,meqn
            q(m,1-ibc) = q(m,mx+1-ibc)
         end do
end do
```

Pyclaw

Pyclaw uses an extended array of `q` called `qbc`. If we assume that we use $m_{bc} = 2$ ghost cells at each boundary, then

- `qbc[0]` and `qbc[1]` represent the left boundary conditions.
- We have `qbc[2]=q[0]` and so on.
- `qbc[-1]` and `qbc[-2]` represent the right boundary conditions.

The boundary conditions are called in the setup routine:

```
solver.bc_lower[0] = pyclaw.BC.custom
solver.user_bc_lower = inlet_bc
solver.bc_upper[0] = pyclaw.BC.custom
solver.user_bc_upper = outlet_bc
```

CHAPTER 3 Linear advection

3.1 Linear advection equation

3.1.1 Governing equation and solution

We are studying the linear advection equation

$$\frac{\partial q}{\partial t} + a\frac{\partial q}{\partial x} = 0, \tag{3.1}$$

where $a \neq 0$ is a constant called the advection velocity. The solution to the Riemann problem

$$q(x,0) = \begin{cases} q_l \text{ if } x < 0 \\ q_r \text{ if } x > 0 \end{cases}$$

is straightforward

$$q(x,t) = \begin{cases} q_l \text{ if } x < at \\ q_r \text{ if } x > at \end{cases}$$

Equation (3.1) can be written in characteristic form

$$\frac{1}{\mathrm{d}t} = \frac{a}{\mathrm{d}x} = \frac{0}{\mathrm{d}q},$$

and thus the solution is any function in the form

$$q(x,t) = F(x - at).$$

In particular, when the initial condition is $q(x,0) = q_0(x)$, then

$$q(x,t) = q_0(x - at). \tag{3.2}$$

We can implement the method detailed in § 2.4.1 by defining the wave and fluctuations

$$W_{i-1/2} = Q_i - Q_{i-1}, \tag{3.3}$$
$$A^+\Delta Q_{i-1/2} = \dot{\sigma}^+_{i-1/2} W_{i-1/2}, \tag{3.4}$$
$$A^-\Delta Q_{i+1/2} = \dot{\sigma}^-_{i+1/2} W_{i+1/2}, \tag{3.5}$$

where $\dot{\sigma}^+ = \max(a,0)$ and $\dot{\sigma}^- = \min(a,0)$.

3.1.2 Implementation in Clawpack

In the recent versions of Clawpack, any application needs a number of files (see § 2.11.2):

- The `Makefile` for linking the executables and running the code.
- Two python files: `setrun.py` generate the data needed by clawpack, while `setplot.y` specifies how to plot the numerical results. `setrun.py` generates two data files: `setprob.data`, which contains model parameters (here the advection velocity a) and `claw.data`, which includes all the essential information needed by clawpack.
- The following fortran files:
  - `qinit.f90` provides the initial values
  - `setprob.f90` reads the model parameters from the file `setprob.data`.

Note that the Riemann solver uses the arrays `ql` and `qr`: `ql(i)` is the value of q on the left of the cell, whereas `qr(i)` denotes the value of q on its right (see Fig. 2.15). The Riemann problem at the interface $x_{i-1/2}$ between cells i and $i-1$ thus involves the difference between the left state `ql(i-1,:)` and right state `ql(i,:)`. This is why the wave is defined as $wave(1,1,i) = ql(1,i) - qr(1,i-1)$ in the following. This notation is potentially confusing because throughout this document, q_r and q_l refer to the right and left states, respectively. This notation is consistent with the notation used in Fig. 2.9 (see also the section devoted to high-resolution methods in § 2.8).

In classic Clawpack, the algorithm for the solver is quite simple.

```
! =====================================================
subroutine rp1(maxm,meqn,mwaves,maux,mbc,mx,ql,qr,auxl,auxr,wave,s,amdq,
    apdq)
! =====================================================

    implicit double precision (a-h,o-z)

    dimension wave(meqn, mwaves, 1-mbc:maxm+mbc)
    dimension    s(mwaves,1-mbc:maxm+mbc)
    dimension   ql(meqn, 1-mbc:maxm+mbc)
    dimension   qr(meqn, 1-mbc:maxm+mbc)
    dimension apdq(meqn, 1-mbc:maxm+mbc)
    dimension amdq(meqn, 1-mbc:maxm+mbc)

!     # advection velocity:
!     # (should be set in setprob.f)
    common /cparam/ a

!     # define the wave:
    do 20 i = 2-mbc, mx+mbc

!     # Compute the waves.

        wave(1,1,i) = ql(1,i) - qr(1,i-1)
        s(1,i) = a
    20 END DO


!     # compute the leftgoing and rightgoing flux differences:
!     # Note s(1,i) < 0   and   s(2,i) > 0.

```

```fortran
    do 220 m=1,meqn
        do 220 i = 2-mbc, mx+mbc
            amdq(m,i) = dmin1(0.d0,s(1,i))*wave(m,1,i)
            apdq(m,i) = dmax1(0.d0,s(1,i))*wave(m,1,i)
    220 END DO

    return
    end subroutine rp1
```

3.1.3 Example

Let us consider Eq. (3.1) over the $[0,1]$ interval with $a = 0.5\,\mathrm{m{\cdot}s^{-1}}$. The boundary conditions are cyclic:

$$q(1,t) = q(0,t).$$

The initial condition is the superposition of a Gaussian and a unit box

$$q_0 = \exp\left(-\beta(x-x_0)^2\right) + U(x)$$

with $\beta = 200$ m^{-2} and $x_0 = 0.6$ m, and

$$U(x) = \begin{cases} 0 \text{ if } x < 0.1 \text{ or } x > 0.4 \\ 1 \text{ if } 0.1 \leq x \leq 0.4 \end{cases}$$

Figure 3.1 shows the comparison between the exact and numerical solutions.

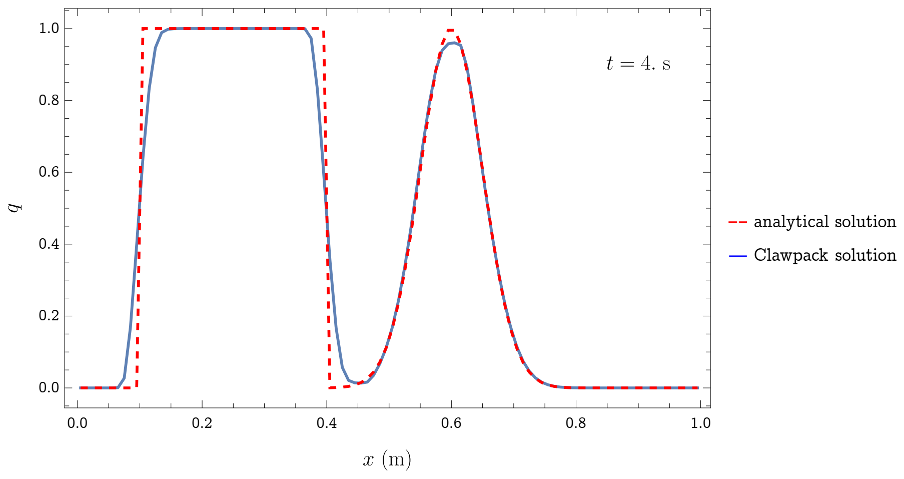


Figure 3.1 Comparison of the numerical and analytical solutions at time $t = 4$ s. Computation with $n = 100$ cells. We used a second-order scheme (Law–Wendrow–LeVeque).

Figure 3.1 compares the effects of three flux limiters on the numerical solution. In this test, the MC and superbee limiters perform better than the mindmod limiter, and naturally far better that the simple Godunov scheme.

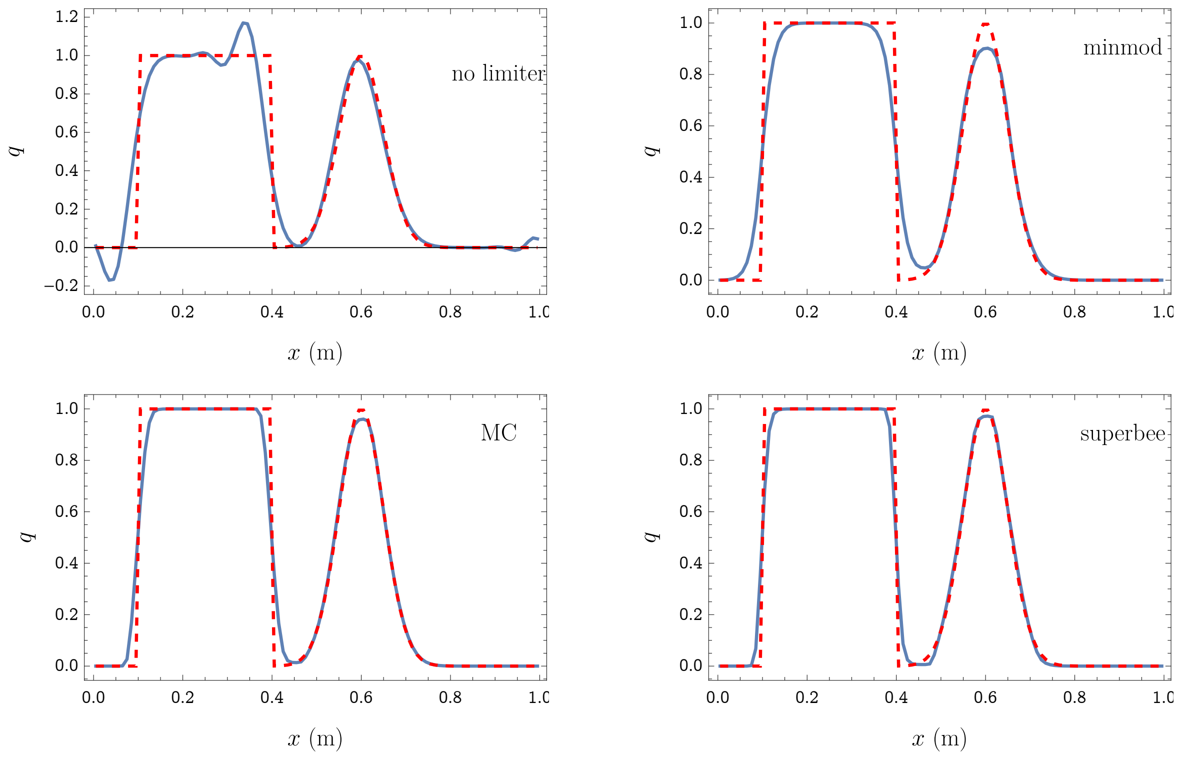


Figure 3.2 Comparison of the numerical and analytical solutions at time $t = 4$ s. Computation with $n = 100$ cells. We tested a first-order Godunov scheme (no limiter) and three second-order schemes (limiters: MC, minmod, and superbee).

3.1.4 Implementation in AMRClaw

AMRClaw allows us to compute high-accuracy solutions by refining the mesh locally. Using AMRClaw requires a few changes in the setup files:

- Makefile: the variables `CLAW_PKG` and `EXE` must be set to `xamr` and `amrclaw\verb`, respectively.
- Setup.py: a number of variables related to mesh refinement must be defined in the setrun.py file. See the AMRClaw webpage devoted to this file.

See also the AMRClaw webpage for general information.

Here we show an example of numerical solution obtained using AMRClaw with three grid levels. This is the same example as the one seen above. Figure 3.3 shows that AMRClaw performs better than the classic Clawpack version (see Figure 3.2).

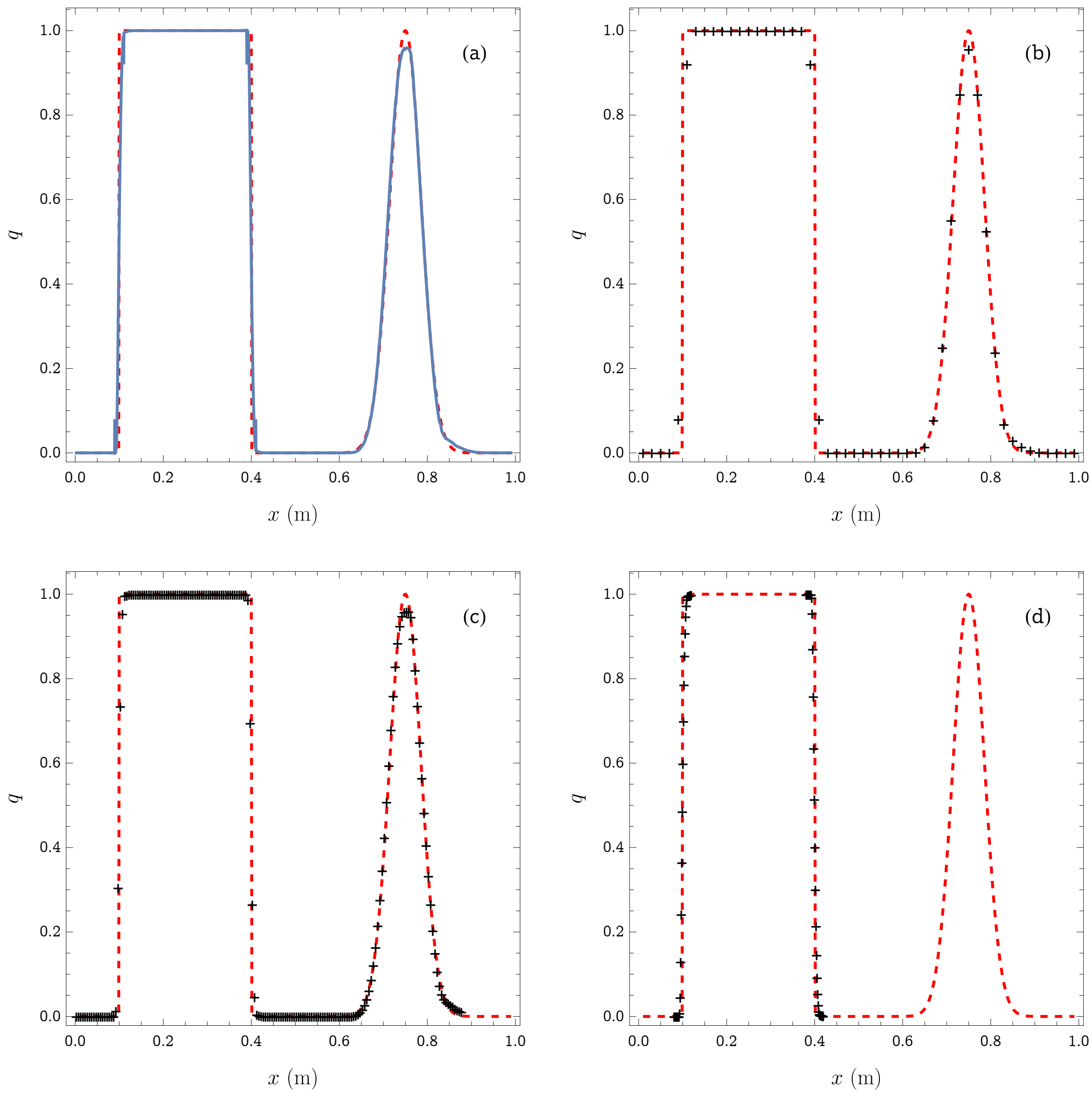


Figure 3.3 Comparison of the numerical and analytical solutions at time $t = 4$ s. Computation with $n = 100$ cells. We used AMRClaw with three grids. (a) Comparison of the numerical solution with the exact solution. (b) Coarse mesh (grid level 1) with $\Delta x = 0.02$ m. (c) Medium mesh (grid level 2) with $\Delta x = 0.005$ m. (b) Fine mesh (grid level 3) with $\Delta x = 0.00125$ m.

3.1.5 *Implementation in Pyclaw*

Classic solvers

To solve a problem using Pyclaw, we will define the following objects:

- The controller handles the running, output, and can be used for plotting the solution (we can run Pyclaw without using a controller, but it makes life easier).
- The time interval over which the solution is computed.
- The solver for the governing equation to be solved. In Pyclaw, a number of solvers are available. They are defined by the class `ClawSolverxD` or `SharpClawSolverxD`, where `x` = 1, 2, 3 is the problem's spatial dimension. SharpClaw algorithms involve on WENO reconstruction and Runge-Kutta time stepping (Ketcheson *et al.*, 2013). User-defined solvers can also be used.
- The computational domain is the domain over which the problem is solved.
- The solution is an object that initially contains the initial data, and then after iterations, the solutions at specific times.

Contrary to the classic Clawpack notation, Pyclaw considers that the variables `q_l` and `q_r` refer to the states left and right states at each interface $x_{i-1/2}$. This is a source of confusion when working with both classic Clawpack and Pyclaw.

```python
%matplotlib inline

from numpy import sqrt, exp, cos, logical_and, where
from clawpack import riemann
from clawpack import pyclaw

def advection(q_l,q_r,aux_l,aux_r,problem_data):
    r"""
    1d linear advection riemann solver
    """
    import numpy as np
    num_eqn = 1
    num_waves = 1

    # Convenience
    num_rp = q_l.shape[1]

    # Return values
    wave = np.empty( (num_eqn, num_waves, num_rp) )
    s = np.empty( (num_waves, num_rp) )
    amdq = np.empty( (num_eqn, num_rp) )
    apdq = np.empty( (num_eqn, num_rp) )

    # Local values
    delta = np.empty(np.shape(q_l))

    delta = q_r - q_l
    a =  problem_data['a']

    # Compute the wave
    # 1-Wave
    wave[0,0,:] = delta
    s[0,:] = a

    # Compute the left going and right going fluctuations
    for m in range(num_eqn):
        amdq[m,:] = min(a,0) * wave[m,0,:]
        apdq[m,:] = max(a,0) * wave[m,0,:]

    return wave, s, amdq, apdq


def setup(outdir='./_output',  output_style=1):

    solver = pyclaw.ClawSolver1D(advection)
    solver.num_waves = 1
    solver.num_eqn = 1
    solver.kernel_language = 'Python'
    solver.limiters = pyclaw.limiters.tvd.MC
    solver.bc_lower[0] = pyclaw.BC.periodic
    solver.bc_upper[0] = pyclaw.BC.periodic
    solver.order = 1 #1: Godunov, 2: Lax-Wendrow-LeVeque

    x = pyclaw.Dimension(0.0, 1.0, 100, name='x')
    domain = pyclaw.Domain(x)
    num_eqn = 1

    state = pyclaw.State(domain, num_eqn)

    a = 0.5  # advection velocity
    state.problem_data['a'] = a

    xc = domain.grid.x.centers
    beta = 100
    gamma = 0
    x0 = 0.75
    state.q[0, :] = exp(-beta * (xc-x0)**2) + where(logical_and(xc > 0.1,
   xc < 0.4),  1, 0)

    claw = pyclaw.Controller()
    claw.solution = pyclaw.Solution(state, domain)
    claw.solver = solver
    claw.outdir = outdir
    claw.output_style = output_style
    claw.tfinal = 4.0
    claw.num_output_times = 10
    claw.keep_copy = True
    #claw.setplot = setplot

    return claw


def setplot(plotdata):
    """
    Plot solution using VisClaw.
    """
    plotdata.clearfigures()  # clear any old figures,axes,items data

    plotfigure = plotdata.new_plotfigure(name='q', figno=1)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.ylimits = [-.2,1.0]
    plotaxes.title = 'q'

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
    plotitem.plot_var = 0
    plotitem.plotstyle = '-o'
    plotitem.color = 'b'
    plotitem.kwargs = {'linewidth':2,'markersize':5}

    return plotdata


def qtrue(x,t):
    """
    The true solution, for comparison.
    """
    from numpy import mod, exp, where, logical_and
    beta = 100
    gamma = 0
    x0 = 0.75
    u = claw.solution.state.problem_data['a']
    xm = x - u*t
    xm = mod(xm, 1.)   # because of periodic boundary conditions
    q = exp(-beta * (xm-x0)**2) + where(logical_and(xm > 0.1, xm < 0.4),
   1, 0)
    return q
```

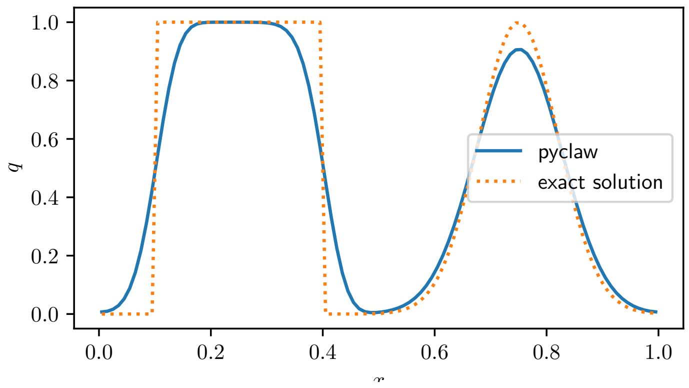
We run the code

```
claw = setup()
claw.run()

from clawpack.visclaw import data
from clawpack.visclaw import frametools
plotdata = data.ClawPlotData()
plotdata.setplot = setplot
claw.plotdata = frametools.call_setplot(setplot,plotdata)

frame = claw.load_frame(10)
f=claw.plot_frame(frame)
```

Next, we plot the solution for time $t = 2$ s. The figure compares the numerical and exact solutions at time $t = 2$ s. As we used a first-order scheme (Godunov), the numerical viscosity smooths out sharp variations in q.

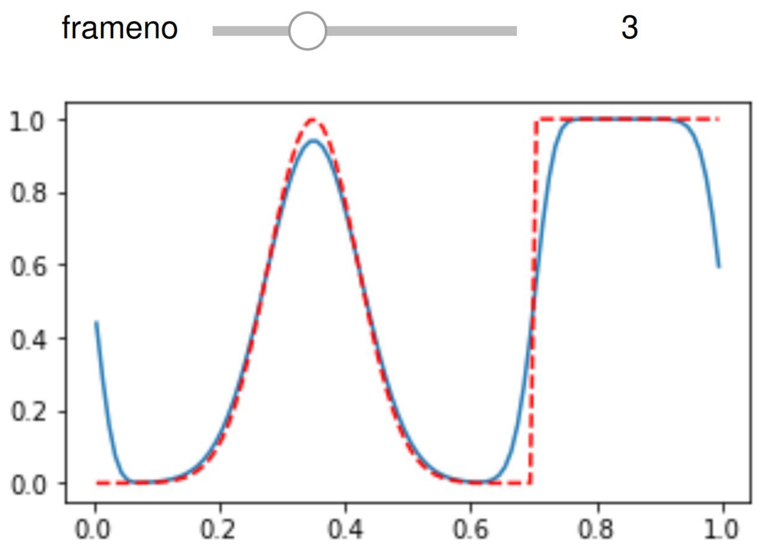
```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['text.usetex'] = True
frame = claw.frames[5]
dt = claw.tfinal/claw.num_output_times
t = dt*5
true = qtrue(x,t)
fig, ax = plt.subplots(figsize=(5, 2.7))
w = frame.q[0,:]
x = frame.state.grid.c_centers
x = x[0]
```

```python
ax.plot(x, w, label='pyclaw')
ax.plot(x, true, ':',label='exact solution')
ax.legend(loc='right')
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$q$')
plt.savefig('frameAdvection5.pdf')
```



We can also plot an animation:

```python
nsimul=np.size(claw.frames)
figs = []
for i in range(nsimul):
    fig = plt.figure(figsize=(5,3))
    frame = claw.frames[i]
    w = frame.q[0,:]
    x = frame.state.grid.c_centers
    x = x[0]
    dt = claw.tfinal/claw.num_output_times
    t = dt*i
    true = qtrue(x,t)
    plt.plot(x, w)
    plt.plot(x, true, '--',color = 'r')
    figs.append(fig)
    plt.close(fig)
from clawpack.visclaw import animation_tools
animation_tools.interact_animate_figs(figs)
```



Sharpclaw solvers

We can also use Sharpclaw solvers by initiating the solver `solver␣=␣pyclaw.SharpClawSolver1D()` and specifying its type (e.g., here the custom solver `advection`). A number of additional variables must be set, in particular the WENO order (defined by `weno_order`) and the time integrator (defined by `time_integrator`). See the Clawpack documentation for further information on these variables. Here we provide an example with the third-order strong stability-preserving method SSP33 proposed by Shu and Osher (Ketcheson *et al.*, 2011).

```
 %matplotlib inline

from numpy import sqrt, exp, cos, logical_and, where
from clawpack import riemann
from clawpack import pyclaw

def advection(q_l,q_r,aux_l,aux_r,problem_data):
    r"""
    1d linear advection riemann solver
    """
    import numpy as np
    num_eqn = 1
    num_waves = 1

    # Convenience
    num_rp = q_l.shape[1]

    # Return values
    wave = np.empty( (num_eqn, num_waves, num_rp) )
    s = np.empty( (num_waves, num_rp) )
    amdq = np.empty( (num_eqn, num_rp) )
    apdq = np.empty( (num_eqn, num_rp) )

    # Local values
    delta = np.empty(np.shape(q_l))

    delta = q_r - q_l
    a =  problem_data['u']

    # Compute the wave
    # 1-Wave
    wave[0,0,:] = delta
    s[0,:] = a

    # Compute the left going and right going fluctuations
    for m in range(num_eqn):
        amdq[m,:] = min(a,0) * wave[m,0,:]
        apdq[m,:] = max(a,0) * wave[m,0,:]

    return wave, s, amdq, apdq

def setup(outdir='./_output',  output_style=1):

    solver = pyclaw.SharpClawSolver1D()
    solver.rp = advection
    solver.weno_order = 5
    solver.lim_type = 2
    solver.time_integrator = 'SSP33'
```

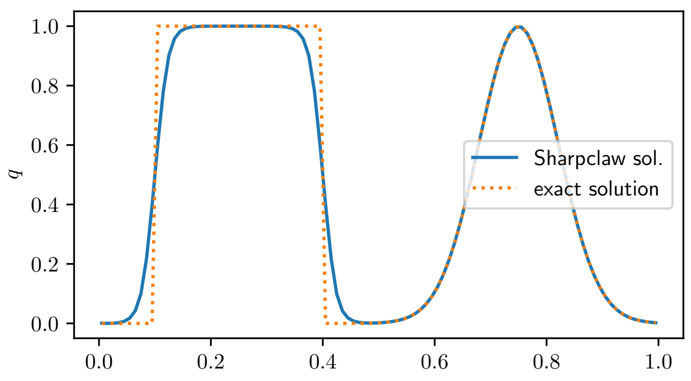
```python
    solver.cfl_max = 0.5

    solver.num_waves = 1
    solver.num_eqn = 1
    solver.kernel_language = 'Python'
    solver.limiters = pyclaw.limiters.tvd.superbee
    solver.bc_lower[0] = pyclaw.BC.periodic
    solver.bc_upper[0] = pyclaw.BC.periodic

    x = pyclaw.Dimension(0.0, 1.0, 100, name='x')
    domain = pyclaw.Domain(x)
    num_eqn = 1

    state = pyclaw.State(domain, num_eqn)

    a = 0.5  # advection velocity
    state.problem_data['u'] = a

    xc = domain.grid.x.centers
    beta = 100
    gamma = 0
    x0 = 0.75
    state.q[0, :] = exp(-beta * (xc-x0)**2) + where(logical_and(xc > 0.1,
   xc < 0.4),  1, 0)

    claw = pyclaw.Controller()
    claw.solution = pyclaw.Solution(state, domain)
    claw.solver = solver
    claw.outdir = outdir
    claw.output_style = output_style
    claw.tfinal = 4.0
    claw.num_output_times = 10
    claw.keep_copy = True
    #claw.setplot = setplot

    return claw


def setplot(plotdata):
    """
    Plot solution using VisClaw.
    """
    plotdata.clearfigures()  # clear any old figures,axes,items data

    plotfigure = plotdata.new_plotfigure(name='q', figno=1)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.ylimits = [-.2,1.0]
    plotaxes.title = 'q'

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
    plotitem.plot_var = 0
    plotitem.plotstyle = '-o'
    plotitem.color = 'b'
    plotitem.kwargs = {'linewidth':2,'markersize':5}

    return plotdata

```

We run the code.

```
claw = setup()
claw.run()

from clawpack.visclaw import data
from clawpack.visclaw import frametools
plotdata = data.ClawPlotData()
plotdata.setplot = setplot
claw.plotdata = frametools.call_setplot(setplot,plotdata)

frame = claw.load_frame(10)
f=claw.plot_frame(frame)
```

We can eventually plot the solution at time $t = 4$ s and compare with the exact solution.

```
def qtrue(x,t):
    """
    The true solution, for comparison.
    """
    from numpy import mod, exp, where, logical_and
    beta = 100
    gamma = 0
    x0 = 0.75
    u = claw.solution.state.problem_data['u']
    xm = x - u*t
    xm = mod(xm, 1.)   # because of periodic boundary conditions
    q = exp(-beta * (xm-x0)**2) + where(logical_and(xm > 0.1, xm < 0.4),
   1, 0)
    return q

%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['text.usetex'] = True
frame = claw.frames[5]
dt = claw.tfinal/claw.num_output_times
t = dt*5
x = frame.state.grid.c_centers
x = x[0]
true = qtrue(x,t)
fig, ax = plt.subplots(figsize=(5, 2.7))
w = frame.q[0,:]

ax.plot(x, w, label='Sharpclaw sol.')
ax.plot(x, true, ':',label='exact solution')
ax.legend(loc='right')
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$q$')
plt.savefig("frameAdvectionSharpClaw.pdf")

```



1.0
0.8
0.6
0.4
0.2
0.0
q
0.0
0.2
0.4
0.6
0.8
1.0
Sharpclaw sol.
exact solution

3.2 Acoustic waves

3.2.1 Governing equation

When linearised, the acoustic wave equation takes the form

$$\frac{\partial p}{\partial t} + K\frac{\partial u}{\partial x} = 0,$$
$$\frac{\partial u}{\partial t} + \frac{1}{\varrho}\frac{\partial p}{\partial x} = 0,$$

where K is the bulk modulus, ϱ the density, $u(x,\ t)$ and $p(x,\ t)$ the velocity and pressure. We define the speed of sound as $c = \sqrt{K/\varrho}$ and impedance $Z = \sqrt{K\varrho}$. In tensorial form, the acoustic wave equation is:

$$\frac{\partial \boldsymbol{q}}{\partial t} + \boldsymbol{A}\cdot\frac{\partial \boldsymbol{q}}{\partial x}, \text{ with } \boldsymbol{q} = \left(\begin{array}{c} p \\ u \end{array}\right) \text{ and } \boldsymbol{A} = \left(\begin{array}{cc} O & K \\ \varrho^{-1} & 0 \end{array}\right).$$

We define the right and left eigenvector matrices $\boldsymbol{R}$ and $\boldsymbol{L}$

$$\boldsymbol{R} = \left(\begin{array}{cc} -Z & Z \\ 1 & 1 \end{array}\right) \text{ et } \boldsymbol{L} = \frac{1}{2}\left(\begin{array}{cc} Z^{-1} & 1 \\ Z & 1 \end{array}\right)$$

and the eigenvalue matrix $\boldsymbol{\Lambda}$

$$\boldsymbol{\Lambda} = \left(\begin{array}{cc} \lambda_1 & 0 \\ 0 & \lambda_2 \end{array}\right) \text{ avec } \lambda_1 = -c \text{ et } \lambda_2 = +c.$$

We diagonalise the matrix $\boldsymbol{A}$

$$\boldsymbol{A} = \boldsymbol{R}\cdot\boldsymbol{\Lambda}\cdot\boldsymbol{L}.$$

By introducing the Riemann variables

$$\boldsymbol{r} = \boldsymbol{L}\cdot\boldsymbol{q}$$

we want to solve

$$\frac{\partial \boldsymbol{r}}{\partial t} + \boldsymbol{\Lambda}\cdot\frac{\partial \boldsymbol{r}}{\partial x} = 0,$$

subject to the initial conditions

$$r_i = \begin{cases} r_{i,l} \text{ if } x < 0 \\ r_{i,r} \text{ if } x > 0 \end{cases}$$

In a Riemann problem, the left and right states can be connected using the right eigenvectors:

$$\boldsymbol{q}_r - \boldsymbol{q}_l = \alpha_1\boldsymbol{w}_1 + \alpha_2\boldsymbol{w}_2 = \boldsymbol{R}\cdot\boldsymbol{\alpha},$$

thus

$$\boldsymbol{\alpha} = \boldsymbol{R}^{-1}\cdot(\boldsymbol{q}_r - \boldsymbol{q}_l) = \boldsymbol{L}\cdot(\boldsymbol{q}_r - \boldsymbol{q}_l),$$

which leads to:

$$\alpha_1 = \frac{1}{2}\left(-\frac{p_r - p_l}{Z} + u_r - u_l\right),$$

$$\alpha_2 = \frac{1}{2}\left(\frac{p_r - p_l}{Z} + u_r - u_l\right),$$

The jump from $\boldsymbol{q}_l$ to $\boldsymbol{q}_*$ is $\boldsymbol{W}_1 = \alpha_1\boldsymbol{r}_1$ while the jump from $\boldsymbol{q}_*$ to $\boldsymbol{q}_r$ is $\boldsymbol{W}_2 = \alpha_2\boldsymbol{r}_2$.

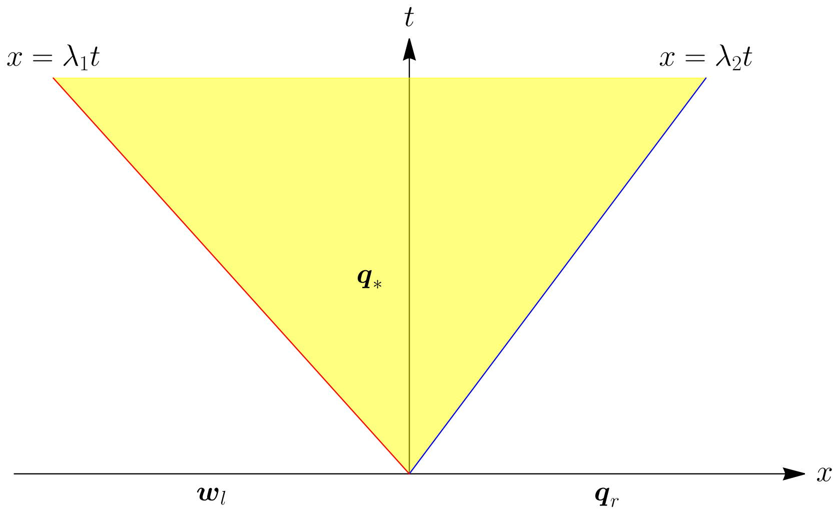


Figure 3.4 Solution to the Riemann problem.

3.2.2 *Implementation*

In classic Clawpack, the algorithm for the solver is quite simple.

```fortran
! =====================================================
subroutine rp1(maxm,meqn,mwaves,maux,mbc,mx,ql,qr,auxl,auxr,wave,s,amdq,
    apdq)
! =====================================================

    implicit double precision (a-h,o-z)

    dimension wave(meqn, mwaves, 1-mbc:maxm+mbc)
    dimension    s(mwaves,1-mbc:maxm+mbc)
    dimension   ql(meqn, 1-mbc:maxm+mbc)
    dimension   qr(meqn, 1-mbc:maxm+mbc)
    dimension apdq(meqn, 1-mbc:maxm+mbc)
    dimension amdq(meqn, 1-mbc:maxm+mbc)

!     local arrays
!     ------------
    dimension delta(2)

!     # density, bulk modulus, and sound speed, and impedance of medium:
!     # (should be set in setprob.f)
    common /cparam/ rho,bulk,cc,zz

!     # find a1 and a2, the coefficients of the 2 eigenvectors:
    do 20 i = 2-mbc, mx+mbc
        delta(1) = ql(1,i) - qr(1,i-1)
        delta(2) = ql(2,i) - qr(2,i-1)
        a1 = (-delta(1) + zz*delta(2)) / (2.d0*zz)
        a2 =  (delta(1) + zz*delta(2)) / (2.d0*zz)

    !        # Compute the waves.

        wave(1,1,i) = -a1*zz
        wave(2,1,i) = a1
        s(1,i) = -cc

        wave(1,2,i) = a2*zz
        wave(2,2,i) = a2
        s(2,i) = cc

    20 END DO


!     # compute the leftgoing and rightgoing flux differences:
!     # Note s(1,i) < 0   and   s(2,i) > 0.

    do 220 m=1,meqn
        do 220 i = 2-mbc, mx+mbc
            amdq(m,i) = s(1,i)*wave(m,1,i)
            apdq(m,i) = s(2,i)*wave(m,2,i)
    220 END DO

    return
    end subroutine rp1
```

3.2.3 Implementation in Pyclaw

Here we give an example of Pyclaw script. We start with the initialisation of the controller.

```
claw = pyclaw.Controller()  # Creation of the controller
claw.tfinal = 1.0           # Final time
claw.keep_copy = True       # Keep solution data in memory for plotting
claw.output_format = None   # Don't write solution data to file
claw.num_output_times = 50  # Write 50 output frames

# Riemann solver
# Here we use the one provided in Clawpack
# We could have also used SharpClawSolver1D
riemann_solver = riemann.acoustics_1D
claw.solver = pyclaw.ClawSolver1D(riemann_solver)
```

For the boundary conditions, we can specify all boundaries at once by selecting periodic (periodic), wall (wall), user-defined (custom), or extrapolation (extrap):

```
# Boundary conditions
claw.solver.all_bcs = pyclaw.BC.periodic
```

We can also define the boundary conditions separately

```
claw.solver.bc.lower[0] = pyclaw.BC.wall
claw.solver.bc.upper[0] = pyclaw.BC.wall
```

We will solve the governing equation over the unit line [0, 1] using 100 grid cells. The arguments of the Domain object are tuples:

```
domain = pyclaw.Domain( (0.,), (1.,), (100,))
```

Next we create a solution object whose dimension is given by `claw.solver.num_eqn` (set automatically when selecting the solver) and which belongs to the controller and extends over the domain specified above:

```
claw.solution = pyclaw.Solution(claw.solver.num_eqn,domain)
```

The initial data is specified in the array `q`: the pressure is contained in `q[0,:]` and the velocity in `q[1,:]`. We impose a wavepacket to the initial pressure and zero velocity.

```
x=domain.grid.x.centers
beta=100; gamma=5; x0=0.75
claw.solution.q[0,:] = np.exp(-beta * (x-x0)**2) * np.cos(gamma * (x - x0))
claw.solution.q[1,:] = 0.
```

The Riemann solver requires some physical parameters to be specified:

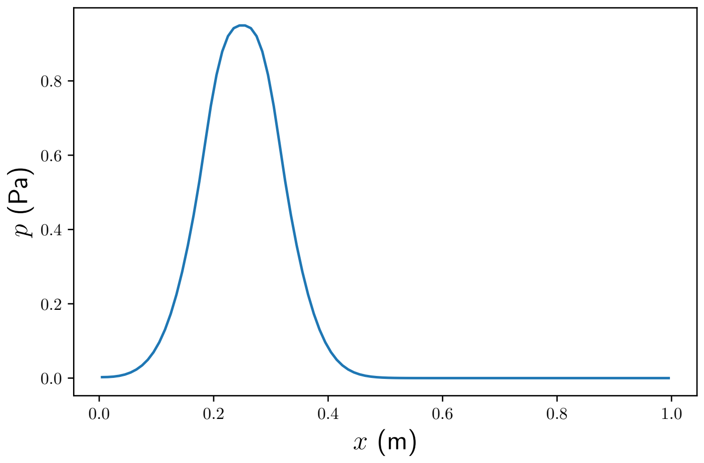
```
import numpy as np

density = 1.0
bulk_modulus = 1.0
impedance = np.sqrt(density*bulk_modulus)
sound_speed = np.sqrt(density/bulk_modulus)

claw.solution.state.problem_data = {
                              'rho' : density,
                              'bulk': bulk_modulus,
                              'zz'  : np.sqrt(density*bulk_modulus),
                              'cc'  : np.sqrt(bulk_modulus/density)
                              }
```

We can now run Pyclaw:

```
claw.solver.dt_initial = 1.e-99
status = claw.run()
```

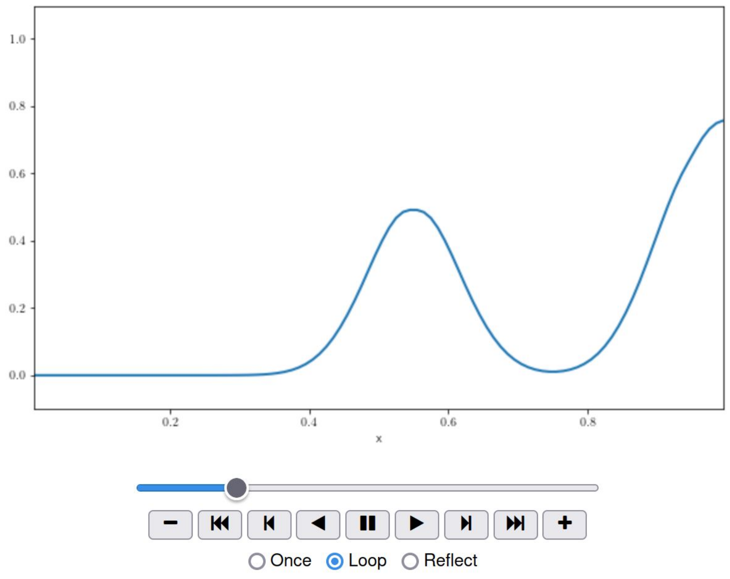
The results are contained in `claw.frames[:]`. We can plot a single frame using matplotlib:

```
pressure = claw.frames[50].q[0,:]
plt.rcParams['text.usetex'] = True
#plt.plot(x,pressure,'-')
fig, ax = plt.subplots(figsize=(6, 4), tight_layout=True)
ax.plot(x, pressure)

ax.set_xlabel(r'$x$ (m)', fontsize=16)
ax.set_ylabel(r'$p$ (Pa)', fontsize=16)
plt.savefig('solution50.pdf')
```

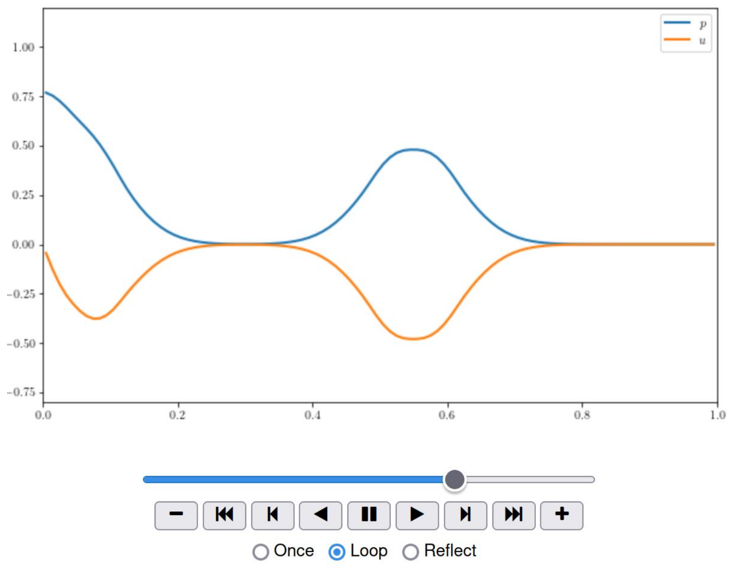


We can also see animations importing `ianimate` from the `visclaw` library (Clawpack visualisation tools):

```python
from clawpack.visclaw import ianimate
ianimate.ianimate(claw)
```



```python
from matplotlib import animation
import matplotlib.pyplot as plt
from IPython.display import HTML
import numpy as np

fig = plt.figure(figsize=(10,6))
ax = plt.axes(xlim=(0, 1), ylim=(-0.8, 1.2))

frame = claw.frames[0]
pressure = frame.q[0,:]
line1, = ax.plot([], [], lw=2)
line2, = ax.plot([], [], lw=2)
plt.legend([r'$p$',r'$u$'])

def fplot(frame_number):
    frame = claw.frames[frame_number]
    pressure = frame.q[0,:]
    velocity = frame.q[1,:]
    line1.set_data(x,pressure)
    line2.set_data(x,velocity)
    return line1,

anim = animation.FuncAnimation(fig, fplot, frames=len(claw.frames),
    interval=30, repeat=False)
plt.close()
HTML(anim.to_jshtml())
```



Here is how Pyclaw has encoded the solver of the Riemann problem.

```python
def acoustics_1D(q_l,q_r,aux_l,aux_r,problem_data):
    r"""
    Basic 1d acoustics riemann solver, with interleaved arrays

    *problem_data* is expected to contain -
     - *zz* - (float) Impedance
     - *cc* - (float) Speed of sound

    See :ref:`pyclaw_rp` for more details.

    :Version: 1.0 (2009-02-03)
    """
    import numpy as np

    # Convenience
    num_rp = np.size(q_l,1)
    num_eqn = 2
    num_waves = 2

    # Return values
    wave = np.empty( (num_eqn, num_waves, num_rp) )
    s = np.empty( (num_waves, num_rp) )
    amdq = np.empty( (num_eqn, num_rp) )
    apdq = np.empty( (num_eqn, num_rp) )

    # Local values
    delta = np.empty(np.shape(q_l))

    delta = q_r - q_l
    a1 = (-delta[0,:] + problem_data['zz']*delta[1,:]) / (2.0 *
   problem_data['zz'])
    a2 = (delta[0,:] + problem_data['zz']*delta[1,:]) / (2.0 * problem_data
   ['zz'])

    # Compute the waves
    # 1-Wave
    wave[0,0,:] = -a1 * problem_data['zz']
    wave[1,0,:] = a1
    s[0,:] = -problem_data['cc']

    # 2-Wave
    wave[0,1,:] = a2 * problem_data['zz']
    wave[1,1,:] = a2
    s[1,:] = problem_data['cc']

    # Compute the left going and right going fluctuations
    for m in range(num_eqn):
        amdq[m,:] = s[0,:] * wave[m,0,:]
        apdq[m,:] = s[1,:] * wave[m,1,:]

    return wave, s, amdq, apdq
```

Here is another example of notebook setting up a solver for the acoustic wave equations, where a user-defined solver is employed.

```
%matplotlib inline

from numpy import sqrt, exp, cos
from clawpack import riemann
from clawpack import pyclaw
def acoustics(q_l,q_r,aux_l,aux_r,problem_data):
    r"""
    Basic 1d acoustics riemann solver, with interleaved arrays

    *problem_data* is expected to contain -
     - *zz* - (float) Impedance
     - *cc* - (float) Speed of sound

    See :ref:`pyclaw_rp` for more details.

    :Version: 1.0 (2009-02-03)
    """
    import numpy as np
    num_eqn = 2
    num_waves = 2

    # Convenience
    num_rp = np.size(q_l,1)

    # Return values
    wave = np.empty( (num_eqn, num_waves, num_rp) )
    s = np.empty( (num_waves, num_rp) )
    amdq = np.empty( (num_eqn, num_rp) )
    apdq = np.empty( (num_eqn, num_rp) )

    # Local values
    delta = np.empty(np.shape(q_l))

    delta = q_r - q_l
    a1 = (-delta[0,:] + problem_data['zz']*delta[1,:]) / (2.0 *
   problem_data['zz'])
    a2 = (delta[0,:] + problem_data['zz']*delta[1,:]) / (2.0 * problem_data
   ['zz'])

```

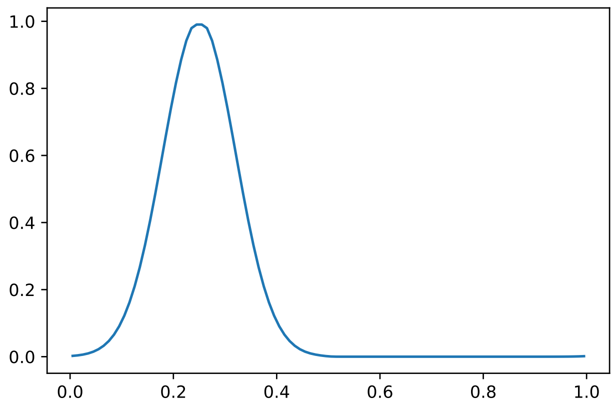
```
    # Compute the waves
    # 1-Wave
    wave[0,0,:] = -a1 * problem_data['zz']
    wave[1,0,:] = a1
    s[0,:] = -problem_data['cc']

    # 2-Wave
    wave[0,1,:] = a2 * problem_data['zz']
    wave[1,1,:] = a2
    s[1,:] = problem_data['cc']

    # Compute the left going and right going fluctuations
    for m in range(num_eqn):
        amdq[m,:] = s[0,:] * wave[m,0,:]
        apdq[m,:] = s[1,:] * wave[m,1,:]

    return wave, s, amdq, apdq

def setup(outdir='./_output',  output_style=1):

    riemann_solver = acoustics
    solver = pyclaw.ClawSolver1D(riemann_solver)
    solver.limiters = pyclaw.limiters.tvd.MC
    solver.kernel_language = 'Python'
    solver.num_waves = 2
    solver.num_eqn = 2
    x = pyclaw.Dimension(0.0, 1.0, 100, name='x')
    domain = pyclaw.Domain(x)
    num_eqn = 2
    state = pyclaw.State(domain, num_eqn)

    solver.bc_lower[0] = pyclaw.BC.periodic
    solver.bc_upper[0] = pyclaw.BC.periodic

    rho = 1.0   # Material density
    bulk = 1.0  # Material bulk modulus

    state.problem_data['rho'] = rho
    state.problem_data['bulk'] = bulk
    state.problem_data['zz'] = sqrt(rho*bulk)   # Impedance
    state.problem_data['cc'] = sqrt(bulk/rho)   # Sound speed

    xc = domain.grid.x.centers
    beta = 100
    gamma = 0
    x0 = 0.75
    state.q[0, :] = exp(-beta * (xc-x0)**2) * cos(gamma * (xc - x0))
    state.q[1, :] = 0.0

    solver.dt_initial = domain.grid.delta[0] / state.problem_data['cc'] *
   0.1

    claw = pyclaw.Controller()
    claw.solution = pyclaw.Solution(state, domain)
    claw.solver = solver
    claw.outdir = outdir
    claw.output_style = output_style
    output_style = 1
    claw.tfinal = 1.0
    claw.num_output_times = 10
    claw.keep_copy = True

    return claw
```

To run the script and plot one result here (frame 10), the following can be done:

```
claw = setup()
claw.run()

from clawpack.visclaw import data
from clawpack.visclaw import frametools
plotdata = data.ClawPlotData()
plotdata.setplot = setplot
claw.plotdata = frametools.call_setplot(setplot,plotdata)

frame = claw.load_frame(10)
f=claw.plot_frame(frame)
```

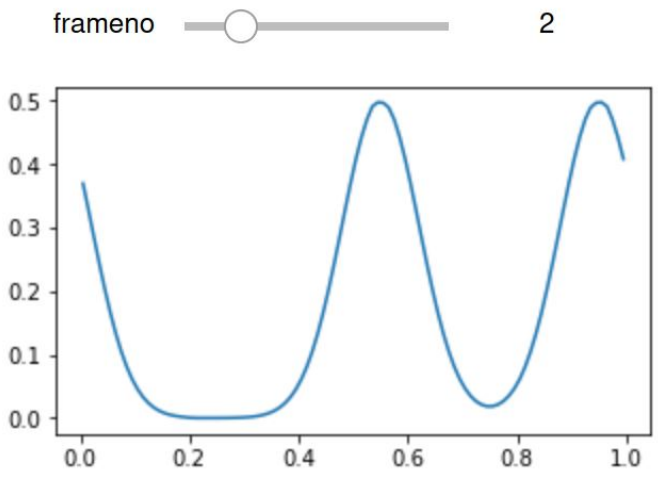
We can also plot a frame directly:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

frame = claw.frames[5]
w = frame.q[0,:]
x = frame.state.grid.c_centers
x = x[0]

plt.plot(x, w)
```



We can also plot an animation:

```
nsimul=np.size(claw.frames)
figs = []
for i in range(nsimul):
    fig = plt.figure(figsize=(5,3))
    frame = claw.frames[i]
    w = frame.q[0,:]
    x = frame.state.grid.c_centers
    x = x[0]
    plt.plot(x, w)
    figs.append(fig)
    plt.close(fig)
    #plt.plot(x, w)
from clawpack.visclaw import animation_tools
animation_tools.interact_animate_figs(figs)
```



3.3 Spatially-varying advection equation

3.3.1 Theory

Let us consider the variable-coefficient linear advection equation:

$$\frac{\partial}{\partial t}\boldsymbol{q} + \boldsymbol{A}(x) \cdot \frac{\partial}{\partial x}\boldsymbol{q} = 0, \tag{3.6}$$

where $\boldsymbol{A}(x)$ is a non-constant diagonisable matrix with real and distinct eigenvalues at any position x of a given domain. In this form, Eq. (3.6) is not in conservative form. Of course, we could make it consistent with the conservative form:

$$\frac{\partial}{\partial t}\boldsymbol{A} + \frac{\partial}{\partial x}(\boldsymbol{A} \cdot \boldsymbol{q}) = (\boldsymbol{A}' \cdot \boldsymbol{q}), \tag{3.7}$$

but when doing so, we transform a homogeneous equation into a non-homogeneous equation, and from the numerical standpoint, handling the resulting source term may be problematic. It can be shown that the overall approach implemented in Clawpack can be applied to equations like Eq. (3.6) (see Chap. 9.12 LeVeque, 2002):

$$\boldsymbol{Q}_i^{n+1} = \boldsymbol{Q}_i^n - \frac{\Delta t}{\Delta x}\left(\boldsymbol{\mathcal{A}}^-\Delta \boldsymbol{Q}_{i+1/2}^n + \boldsymbol{\mathcal{A}}^+\Delta \boldsymbol{Q}_{i-1/2}^n\right) - \frac{\Delta t}{\Delta x}\left(\hat{\boldsymbol{F}}_{i+1/2}^n - \hat{\boldsymbol{F}}_{i-1/2}^n\right), \tag{3.8}$$

where the operators $\boldsymbol{\mathcal{A}}^{\pm}$ are defined by Eqs. (2.46) and (2.46), and the flux correction $\hat{\boldsymbol{F}}_{i+1/2}^n$ is:

$$\hat{\boldsymbol{F}}_{i+1/2}^n = \frac{1}{2}\sum_{k=1}^{M_w} |s_{i-1/2}^k| \left(1 - \frac{\Delta t}{\Delta x}|s_{i-1/2}^k|\right) \tilde{\boldsymbol{W}}_{i-1/2}^k \tag{3.9}$$

The limiters $\tilde{\boldsymbol{W}}_{i-1/2}^k$ require more work (see Chap. 9.13 LeVeque, 2002).

The Clawpack approach can be extended to spatially-varying advection equations in a simple way. Let us consider the one-dimensional advection equation

$$\frac{\partial}{\partial t}q + u(x)\frac{\partial}{\partial x}q = 0, \tag{3.10}$$

where $u(x)$ is a function of x and the initial condition is

$$q(x,0) = q_0(x). \tag{3.11}$$

Contrary to the linear case, we cannot express the cell average Q_i^{n+1} as a function of the flux difference $F_{i+1/2} - F_{i-1/2}$ since Eq. (3.10) is not in a conservative form, but we can still express the change in Q_i^{n+1} from the wave

$$W_{i-1/2} = Q_i - Q_{i-1}$$

and speed

$$s_{i-1/2} = \begin{cases} u_i & \text{if } u_i > 0, \\ u_{i-1} & \text{if } u_i < 0. \end{cases}$$

. Note that in this discretisation, the velocity u is evaluated at the cell center, but we could have defined at the cell interface (which would make more sense in multidimensional problems). We define the fluctuations

$$\mathcal{A}^+\Delta Q_{i-1/2} = s_{i-1/2}^+ W_{i-1/2},$$
$$\mathcal{A}^-\Delta Q_{i-1/2} = s_{i-1/2}^- W_{i-1/2}.$$

3.3.2 Implementation in Pyclaw

The solver is a simple adaptation of the solver for linear advection.

```python
def SpatialAdvection(q_l,q_r,aux_l,aux_r,problem_data):
    r"""Basic 1d advection riemann solver
    *aux(i)* should contain -
     - *u(x_i)* - (float) advection speed
    """
    num_eqn = 1
    num_waves = 1
    # Number of Riemann problems we are solving
    num_rp = q_l.shape[1]

    # Return values
    wave = np.empty( (num_eqn,num_waves,num_rp) )
    s = np.empty( (num_waves,num_rp) )
    amdq = np.zeros( (num_eqn,num_rp) )
    apdq = np.zeros( (num_eqn,num_rp) )

    wave[0,0,:] = q_r[0,:] - q_l[0,:]

    s[0,:] = aux_l[0,:]

    apdq[0,:] = (aux_l[0,:]>0)*s[0,:] * wave[0,0,:]
    amdq[0,:] = (aux_l[0,:]<0)*s[0,:] * wave[0,0,:]

    return wave, s, amdq, apdq
```

3.3.3 Application

Let us consider the case $a = 2\sqrt{x}$. In this case, Eq. (3.10) can be cast in the characteristic form

$$\frac{\mathrm{d}q}{\mathrm{d}t} = 0 \text{ along } \frac{\mathrm{d}x}{\mathrm{d}t} = 2\sqrt{x}.$$

This shows that the characteristics are curves

$$\frac{\mathrm{d}x}{\mathrm{d}t} = 2\sqrt{x} \Rightarrow \frac{\mathrm{d}x}{2\sqrt{x}} = \mathrm{d}t \Rightarrow \sqrt{x} - \sqrt{x_0} = t.$$

The characteristic form implies that

$$q(x,t) = q_0(x_0) = q_0\left((\sqrt{x} - t)^2\right).$$

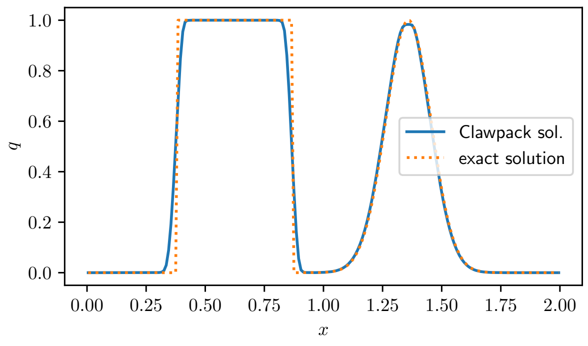
As initial condition, we take the same condition as in § 3.1.3. We need define the auxiliary variable `aux` in the `auxinit` function to define the advection velocity.

```python
%matplotlib inline

import numpy as np
from clawpack import riemann
from clawpack import pyclaw
import matplotlib.pyplot as plt

def qtrue(x,t):
    """
    The true solution, for comparison.
    """
    from numpy import mod, exp, where, logical_and
    beta = 100
    gamma = 0
    x0 = 0.75

    xm = (np.sqrt(x)-t)**2
    q = exp(-beta * (xm-x0)**2) + where(logical_and(xm > 0.1, xm < 0.4),
   1, 0)
    return q

# Advection speed
def auxinit(state):
    # Initialize aux
    xc = state.grid.x.centers
    state.aux[0,:] =  2*np.sqrt(xc)


def setup(outdir='./_output'):
    from clawpack import riemann

    solver = pyclaw.ClawSolver1D(SpatialAdvection)
    solver.num_waves = 1
    solver.num_eqn = 1
    solver.kernel_language = 'Python'
    solver.limiters = pyclaw.limiters.tvd.MC
    solver.bc_lower[0] = pyclaw.BC.extrap
    solver.bc_upper[0] = pyclaw.BC.extrap
    solver.aux_bc_lower[0] = pyclaw.BC.extrap
    solver.aux_bc_upper[0] = pyclaw.BC.extrap

    xlower=0.0; xupper=2.0; mx=200
    x = pyclaw.Dimension(xlower,xupper,mx,name='x')
    domain = pyclaw.Domain(x)
    num_aux, num_eqn = 1, 1
    state = pyclaw.State(domain,num_eqn,num_aux)

    xc = domain.grid.x.centers
    state.q[0,:] = qtrue(xc,0)
    auxinit(state)

    claw = pyclaw.Controller()
    claw.outdir = outdir
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver

    claw.tfinal = .3
    claw.num_output_times = 10
    claw.keep_copy = True

    return claw
```

We run the code and plit the solution for $t = 0.3$ s.

```
claw = setup()
claw.run()
plt.rcParams['text.usetex'] = True
index = 10
```

```python
frame = claw.frames[index]
dt = claw.tfinal/claw.num_output_times
t = dt*index
x = frame.state.grid.c_centers
x = x[0]

true = qtrue(x,t)
fig, ax = plt.subplots(figsize=(5, 2.7))
w = frame.q[0,:]

ax.plot(x, w, label='Clawpack sol.')
ax.plot(x, true, ':',label='exact solution')
ax.legend(loc='right')
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$q$')
plt.savefig("SpatiallyVaryingAdvection.pdf",bbox_inches='tight')
```



3.4 Conservative spatially-varying advection equation

3.4.1 Theory

Let us consider the one-dimensional advection equation, which has a conservative form but also exhibits a space-dependent flux function

$$\frac{\partial}{\partial t}q + \frac{\partial}{\partial x}(u(x)q) = 0, \tag{3.12}$$

where $u(x) > 0$ is a function of x and the initial condition is

$$q(x, 0) = q_0(x). \tag{3.13}$$

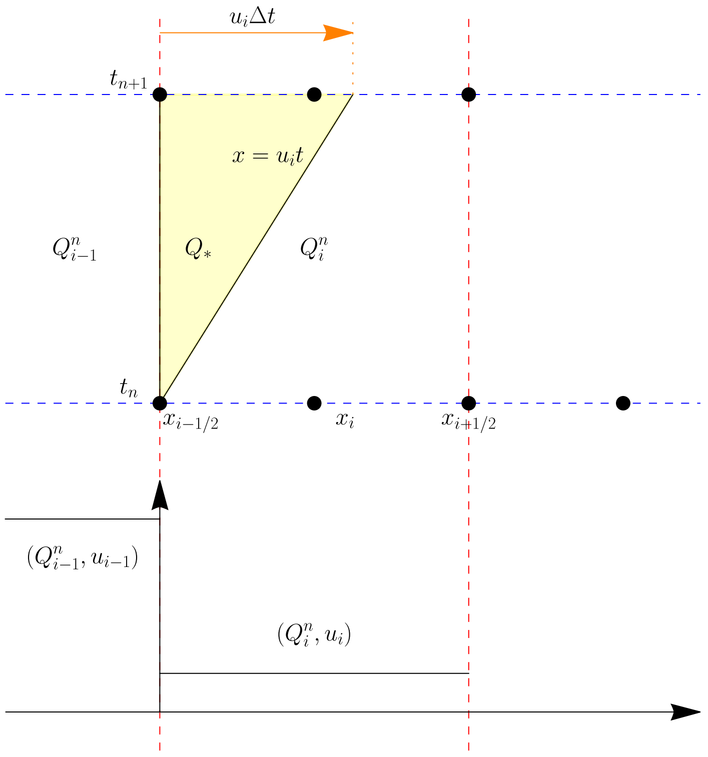


Figure 3.5 Notation for the Riemann problem in Clawpack.

When discretising the equation and solving the associated Riemann problem, there are in fact two Riemann subproblems (see Fig. 3.5):

$$\begin{cases} \partial_t q + u_{i-1}\partial_x q = 0 & \text{subject to } q(x, t_n) = Q_{i-1}^n \quad \text{if } x < x_{i-1/2}, \\ \partial_t q + u_i \partial_x q = 0 & \text{subject to } q(x, t_n) = Q_i^n \quad \text{if } x > x_{i-1/2}. \end{cases} \tag{3.14}$$

These Riemann subproblems are the following:

- At the interface $x_{i-1/2}$, there is a jump from u_{i-1} to u_i. As this jump occurs at the interface $x_{i-1/2}$, its velocity is zero. This jump is associated with a change in q since the flux $f(x, q) = u(x)q$ remains constant. We assume that q jumps from Q_{i-1}^n to an intermediate state Q_*.

- In the cell C_i, the second Riemann involves the left state Q_* and the right state Q_i^n. The shock moves with the velocity $s = u_i$.

The intermediate state is computed by considering the flux conservation

$$u_{i-1}Q_{i-1}^n = u_i Q_* \Rightarrow Q_* = \frac{u_{i-1}Q_{i-1}^n}{u_i}.$$

The first discontinuity has the strength

$$W_{i-1/2}^1 = Q_* - Q_{i-1}$$

and is stationary ($s_{i-1/2}^1 = 0$). The second wave has the strength and velocity

$$W_{i-1/2}^2 = Q_i^n - Q_* \text{ and } s_{i-1/2}^2 = u_i$$

We define the fluctuations

$$\begin{aligned}
\mathcal{A}^+\Delta Q_{i-1/2} &= \sum_{k=1,2} s_{i-1/2}^k W_{i-1/2}^k = u_i\left(Q_i^n - \frac{u_{i-1}Q_{i-1}^n}{u_i}\right) = u_i Q_i^n - u_{i-1}Q_{i-1}^n,\\
\mathcal{A}^-\Delta Q_{i-1/2} &= 0.
\end{aligned}$$

Similarly, when $u(x) < 0$, we have

$$\begin{aligned}
Q_* &= \frac{u_i Q_i^n}{u_{i-1}},\\
W_{i-1/2}^2 &= Q_* - Q_{i-1}^n,\\
s_{i-1/2}^2 &= u_{i-1},\\
\mathcal{A}^+\Delta Q_{i-1/2} &= 0,\\
\mathcal{A}^-\Delta Q_{i-1/2} &= \sum_{k=1,2} s_{i-1/2}^k W_{i-1/2}^k = u_{i-1}\left(\frac{u_i Q_i^n}{u_{i-1}} - Q_{i-1}^n\right) = u_i Q_i^n - u_{i-1}Q_{i-1}^n.
\end{aligned}$$

The general case is thus

$$\begin{aligned}
Q_* &= \begin{cases} \dfrac{u_{i-1}Q_{i-1}^n}{u_i} & \text{if } u_i > 0,\\ \dfrac{u_i Q_i^n}{u_{i-1}} & \text{if } u_i < 0 \end{cases},\\
W_{i-1/2}^2 &= \begin{cases} Q_i^n - Q_* & \text{if } u_i > 0,\\ Q_* - Q_{i-1}^n & \text{if } u_i < 0 \end{cases},\\
s_{i-1/2}^2 &= \begin{cases} u_i & \text{if } u_i > 0,\\ u_{i-1} & \text{if } u_i < 0 \end{cases},\\
\mathcal{A}^+\Delta Q_{i-1/2} &= s_{i-1/2}^- W_{i-1/2}^2 = u_i Q_i^n - u_{i-1}Q_{i-1}^n,\\
\mathcal{A}^-\Delta Q_{i-1/2} &= s_{i-1/2}^+ W_{i-1/2}^2 = u_i Q_i^n - u_{i-1}Q_{i-1}^n.
\end{aligned}$$

Although the overall problem involves two waves, only one affects the evolution of Q_i^{n+1} since the other wave has a zero velocity, and thus its contribution to $\mathcal{A}^\pm\Delta Q_{i-1/2}$ is zero.

3.4.2 Implementation in Clawpack

The Clawpack solver encodes the general case.

```
! =========================================================
subroutine rp1(maxmx,meqn,mwaves,maux,mbc,mx,ql,qr,auxl,auxr,wave,s,amdq,
   apdq)
! =========================================================

    implicit double precision (a-h,o-z)
    dimension   ql(meqn,1-mbc:maxmx+mbc)
    dimension   qr(meqn,1-mbc:maxmx+mbc)
    dimension   qs(meqn,1-mbc:maxmx+mbc)
    dimension  auxl(maux,1-mbc:maxmx+mbc)
    dimension  auxr(maux,1-mbc:maxmx+mbc)
    dimension    s(mwaves,1-mbc:maxmx+mbc)
    dimension wave(meqn, mwaves,1-mbc:maxmx+mbc)
    dimension amdq(meqn,1-mbc:maxmx+mbc)
    dimension apdq(meqn,1-mbc:maxmx+mbc)
    common /comrp/ u, p

    do 30 i=1-mbc,mx+mbc

        u = auxl(1,i)
        p = auxr(1,i-1)

        if (u > 0.d0) then
      qs(1,i) = qr(1,i-1) * p / u
      wave(1,1,i) = ql(1,i) - qs(1,i)
      s(1,i) = u
      amdq(1,i) = 0.d0
      apdq(1,i) = u * wave(1,1,i)
  endif
  if (u < 0.d0) then
      qs(1,i) = qr(1,i) * u / p
      wave(1,1,i) = qs(1,i) - ql(1,i-1)
      s(1,i) = p
      amdq(1,i) = p * wave(1,1,i)
      apdq(1,i) = 0.d0
  endif
  if (u == 0.d0) then
      wave(1,1,i) = 0.d0
      s(1,i) = 0.d0
      amdq(1,i) = 0.d0
      apdq(1,i) = 0.d0
  endif

    30 end do

    return
    end subroutine rp1
```

3.4.3 Implementation in Pyclaw

Here is the Pyclaw version of the solver. In the function `setup()`, we give the choice between the Pyclaw and Fortran versions of the solver. The function u is defined by the function. `auxinit(state)`

```python
def SpatialAdvection(q_l,q_r,aux_l,aux_r,problem_data):
    r"""Basic 1d advection riemann solver
    *aux(i)* should contain -
     - *u(x_i)* - (float) advection speed
    """
    num_eqn = 1
    num_waves = 1
    # Number of Riemann problems we are solving
    num_rp = q_l.shape[1]

    # Return values
    wave = np.empty( (num_eqn,num_waves,num_rp) )
    s = np.empty( (num_waves,num_rp) )
    amdq = np.zeros( (num_eqn,num_rp) )
    apdq = np.zeros( (num_eqn,num_rp) )
    q_sp = np.zeros( (num_eqn,num_rp) )
    q_sm = np.zeros( (num_eqn,num_rp) )
    eps = 1.e-12

    q_sp[0,:] = q_l[0,:]*aux_l[0,:]/ (aux_r[0,:]+eps)
    q_sm[0,:] = q_r[0,:]*aux_r[0,:]/ (aux_l[0,:]+eps)
    wave[0,0,:] = (aux_l[0,:]<0)*(q_sm[0,:] - q_l[0,:])+(aux_l[0,:]>0)*(q_r
    [0,:] - q_sp[0,:])

    s[0,:] = (aux_l[0,:]>0)* aux_r[0,:] +(aux_l[0,:]<0)* aux_l[0,:]

    apdq[0,:] =  (aux_l[0,:]>0)*s[0,:] * wave[0,0,:]
    amdq[0,:] =  (aux_l[0,:]<0)*s[0,:] * wave[0,0,:]

    return wave, s, amdq, apdq

# Advection speed
def auxinit(state):
    xc = state.grid.x.centers
    state.aux[0,:] =  2*np.sqrt(xc)

def setup(typeSolver=1,outdir='./_output'):
    from clawpack import riemann

    if typeSolver == 2:
        solver = pyclaw.ClawSolver1D(SpatialAdvection)
        solver.kernel_language = 'Python'
    elif typeSolver == 1:
        solver = pyclaw.ClawSolver1D(advectionVariableBis)
        solver.kernel_language = 'Fortran'
    solver.num_waves = 1
    solver.num_eqn = 1
    solver.limiters = pyclaw.limiters.tvd.superbee
    solver.order = 2 #1: Godunov, 2: Lax-Wendroff-LeVeque

    solver.limiters = pyclaw.limiters.tvd.MC
    solver.bc_lower[0] = pyclaw.BC.extrap
    solver.bc_upper[0] = pyclaw.BC.extrap
    solver.aux_bc_lower[0] = pyclaw.BC.extrap
    solver.aux_bc_upper[0] = pyclaw.BC.extrap

    xlower = 0.00; xupper = 2.0; mx = 200
    x      = pyclaw.Dimension(xlower,xupper,mx,name='x')
    domain = pyclaw.Domain(x)
    num_aux, num_eqn = 1, 1
    state  = pyclaw.State(domain,num_eqn,num_aux)

    xc = domain.grid.x.centers
    state.q[0,:] = qinit(xc)
    auxinit(state)

    claw = pyclaw.Controller()
    claw.outdir = outdir
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver

    claw.tfinal = 1
    claw.num_output_times = 20
    claw.keep_copy = True

    return claw
```

3.4.4 Application

Let us consider Eq. (3.12) with

$$u(x) = 2\sqrt{x}.$$

Eq. (3.12) can be recast

$$\frac{\partial}{\partial t}q + u(x)\frac{\partial}{\partial x}(q) = -q(x,t)\frac{\mathrm{d}}{\mathrm{d}x}(u),$$

and in characteristic form:

$$\frac{\mathrm{d}}{\mathrm{d}t}q = -\frac{q}{\sqrt{x}} \text{ along } \frac{\mathrm{d}x}{\mathrm{d}t} = 2\sqrt{x}.$$

The characteristic curve's equation is thus

$$\sqrt{x} - \sqrt{x_0} = t.$$

The q time derivative becomes

$$\frac{\mathrm{d}q}{\mathrm{d}t} = -\frac{q}{\sqrt{x}} = -\frac{q}{t+\sqrt{x_0}} \Rightarrow \frac{\mathrm{d}q}{q} = -\frac{\mathrm{d}t}{t+\sqrt{x_0}}.$$

The solution is obtained by taking the initial condition (3.13)

$$\ln\frac{q}{q_0} = -\ln\frac{t+\sqrt{x_0}}{\sqrt{x_0}}.$$

We eventually find

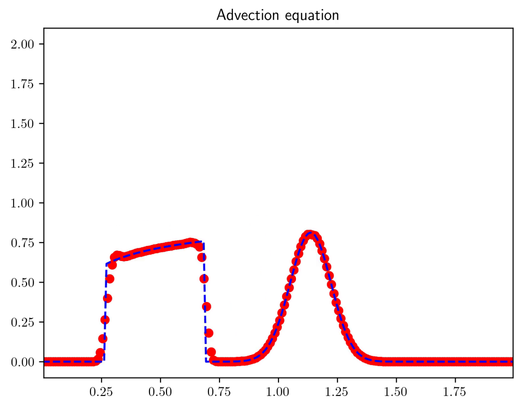
$$\begin{aligned} q(x,t) &= q_0(x_0)\frac{\sqrt{x_0}}{t+\sqrt{x_0}}, \\ &= q_0\left((\sqrt{x}-t)^2\right)\frac{|\sqrt{x}-t|}{\sqrt{x}}. \end{aligned}$$

We encode the exact solution.

```python
def qtrue(x,t):
    """
    The true solution, for comparison.
    """
    from numpy import mod, exp, where, logical_and
    beta = 100
    gamma = 0
    x0 = 0.75
    if isinstance(x,(np.ndarray,list)):
        m = np.size(x)
        q = np.empty((m))
        for i in range(m):
            xm = (np.sqrt(x[i])-t)**2
            if x[i] != 0.:
                q[i] = qinit(xm)/np.sqrt(x[i])*np.sqrt(xm)
            else:
                q[i] = 0.
    if isinstance(x,(float,int)):
        if x != 0:
            xm = (np.sqrt(x)-t)**2
            q = qinit(xm)/np.sqrt(x)*np.sqrt(xm)
        else:
            q = 0
    return q

def qinit(x):
    """
    The true solution, for comparison.
    """
    from numpy import mod, exp, where, logical_and
    beta = 100
    gamma = 0
    x0 = 0.75
    q = exp(-beta * (x-x0)**2) + where(logical_and(x > 0.1, x < 0.4),  1,
   0)
    return q
```

We can use the matplotlib animation toolbox to plot the solution at different times.

```python
def advection_animation():
    import matplotlib.animation
    import numpy
    # compute the solution with the method defined above:
    claw = setup(typeSolver=2)
    claw.keep_copy = True
    claw.run()
    x = claw.frames[0].grid.dimensions[0].centers
    x_true = numpy.linspace(0.00, 2.0, 200)

    fig = plt.figure()
    axes = plt.subplot(1, 1, 1)
    axes.set_xlim((x[0], x[-1]))
    axes.set_ylim((-0.1, 2))
    axes.set_title("Advection equation")

    def init():
        axes.set_xlim((x[0], x[-1]))
        axes.set_ylim((-0.1,2.1))
        computed_line, = axes.plot(x[0], claw.frames[0].q[0, :][0], 'ro')
        exact_line, = axes.plot(x_true, qtrue(x_true,0.0), 'b--')
        return (computed_line, exact_line)

    computed_line, exact_line = init()

    def fplot(n):
        computed_line.set_data([x,], [claw.frames[n].q[0, :]])
        exact_line.set_data([x_true], [qtrue(x_true,claw.frames[n].t)])
        return (computed_line, exact_line)

    frames_to_plot = range(0, len(claw.frames))
    plt.close(fig)
    return matplotlib.animation.FuncAnimation(fig, fplot, frames=
   frames_to_plot, interval=100,
                                       blit=True, init_func=None, repeat=False)

from IPython.display import HTML
anim = advection_animation()
HTML(anim.to_jshtml())
```



We can also plot an animation using the Clawpack animation tools.

```
nsimul=np.size(claw.frames)
figs = []
for i in range(nsimul):
    fig, ax = plt.subplots(figsize=(5, 3))

    frame = claw.frames[i]
    w = frame.q[0,:]
    x = frame.state.grid.c_centers
    x = x[0]
    dt = claw.tfinal/claw.num_output_times
    t = dt*i
    true = qtrue(x,t)
    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'$q$')
    plt.plot(x, w)
    plt.plot(x, true, '--',color = 'r')
    figs.append(fig)
    plt.close(fig)

from clawpack.visclaw import animation_tools
```

```
animation_tools.interact_animate_figs(figs)
```

3.4.5 Alternative

We can also handle Eq. (3.12) as a non-homogeneous advection equation

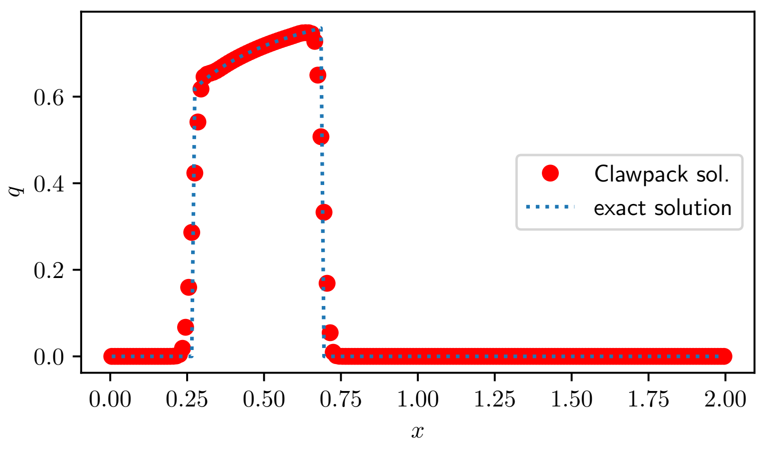
$$\frac{\partial q}{\partial t} + u(x)\frac{\partial q}{\partial x} = -q\frac{\mathrm{d}u}{\mathrm{d}x}.$$

```
%matplotlib inline

import numpy as np
from clawpack import riemann
from clawpack import pyclaw

def qtrue(x,t):
    """
    The true solution, for comparison.
    """
    from numpy import mod, exp, where, logical_and
    beta = 100
    gamma = 0
    x0 = 0.75
    if isinstance(x,(np.ndarray,list)):
        m = np.size(x)
        q = np.empty((m))
        for i in range(m):
            xm = (np.sqrt(x[i])-t)**2
            if x[i] != 0.:
                q[i] = qinit(xm)/np.sqrt(x[i])*np.sqrt(xm)
            else:
                q[i] = 0.
    if isinstance(x,(float,int)):
        if x != 0:
            xm = (np.sqrt(x)-t)**2
            q = qinit(xm)/np.sqrt(x)*np.sqrt(xm)
        else:
            q = 0
    return q

def qinit(x):
    """
    The true solution, for comparison.
    """
    from numpy import mod, exp, where, logical_and
    beta = 100
    gamma = 0
    x0 = 0.75
    q = exp(-beta * (x-x0)**2) + where(logical_and(x > 0.1, x < 0.4),  1,
   0)
    q = where(logical_and(x > 0.1, x < 0.4),  1, 0)
    return q


def advection(q_l,q_r,aux_l,aux_r,problem_data):
    r"""
    1d linear advection riemann solver
    """
    import numpy as np
    num_eqn = 1
    num_waves = 1

    # Number of Riemann problems
    num_rp = q_l.shape[1]

    # Return values
    wave = np.empty( (num_eqn, num_waves, num_rp) )
    s = np.empty( (num_waves, num_rp) )
    amdq = np.empty( (num_eqn, num_rp) )
    apdq = np.empty( (num_eqn, num_rp) )

    # Local values
    delta = np.empty(np.shape(q_l))
    delta = q_r - q_l
    # Compute the wave
    # 1-Wave
    wave[0,0,:] = delta
    s[0,:] = (aux_l[0,:]+aux_r[0,:])/2.
    s_index = np.zeros((2,num_rp))
    s_index[0,:] = s[0,:]
    amdq[0,:] = np.min(s_index,axis=0) * wave[0,0,:]
    apdq[0,:] = np.max(s_index,axis=0) * wave[0,0,:]

    return wave, s, amdq, apdq

def source_term(solver, state, dt):
    from scipy.linalg import solve_banded
    import numpy as np
    qs = state.q[0,:]
    xc = state.grid.c_centers[0]
    dq = -qs/np.sqrt(xc) *dt
    state.q[0,:] = qs+dq

# Advection speed
def auxinit(state):
    xc = state.grid.x.centers
    state.aux[0,:] =  2*np.sqrt(xc)

def setup(outdir='./_output'):
    from clawpack import riemann

    solver = pyclaw.ClawSolver1D(advection)
    solver.kernel_language = 'Python'
    solver.num_waves = 1
    solver.num_eqn = 1

    solver.limiters = pyclaw.limiters.tvd.MC
    solver.bc_lower[0] = pyclaw.BC.extrap
    solver.bc_upper[0] = pyclaw.BC.extrap
    solver.aux_bc_lower[0] = pyclaw.BC.extrap
    solver.aux_bc_upper[0] = pyclaw.BC.extrap
    solver.step_source = source_term

    xlower = 0.0; xupper = 2.0; mx = 200
```

```python
    x      = pyclaw.Dimension(xlower,xupper,mx,name='x')
    domain = pyclaw.Domain(x)
    num_aux, num_eqn = 1, 1
    state  = pyclaw.State(domain,num_eqn,num_aux)

    xc = domain.grid.x.centers
    state.q[0,:] = qinit(xc)
    auxinit(state)

    claw = pyclaw.Controller()
    claw.outdir = outdir
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver

    claw.tfinal = 1
    claw.num_output_times = 20
    claw.keep_copy = True

    return claw

```

We run the code and plot the solution at time $t = 0.2$ s.

```python
import matplotlib.pyplot as plt
claw = setup()
claw.run()
plt.rcParams['text.usetex'] = True
index = 4
frame = claw.frames[index]
dt = claw.tfinal/claw.num_output_times
t = dt*index
x = frame.state.grid.c_centers
x = x[0]

true = qtrue(x,t)
fig, ax = plt.subplots(figsize=(5, 2.7))
w = frame.q[0,:]

ax.plot(x, w, label='Clawpack sol.')
ax.plot(x, true, ':',label='exact solution')
ax.legend(loc='right')
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$q$')
plt.savefig("SpatiallyVaryingAdvectionWithSource.pdf",bbox_inches='tight')
```



0.0
0.2
0.4
0.6
q
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
x
● Clawpack sol.
exact solution

CHAPTER 4 Burgers' equations

4.1 Theory

Let us consider Burgers' equation, which is a nonlinear advection equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0, \tag{4.1}$$

or in conservative form

$$\frac{\partial u}{\partial t} + \frac{\partial f(u)}{\partial x} = 0 \text{ where } f(u) = \frac{u^2}{2}.$$

The solution to the Riemann problem has two types of solution:

- *Rarefaction wave*: $u = U(\xi)$ with $\xi = x/t$. Substituting this form into Eq. (4.1) gives

  $$\frac{\mathrm{d}U}{\partial \xi}\left(-\xi + \frac{\mathrm{d}f}{\mathrm{d}u}(U)\right) = 0 \text{ where } f'(U) = U$$

  whose solution is

  $$U(\xi) = \xi.$$

- *Shock wave*: The Rankine-Hugoniot equation tells us that the shock moves at speed:

  $$\dot{\sigma} = \frac{[\![f(u)]\!]}{[\![u]\!]} = \frac{u^+ + u^-}{2},$$

where u^+ and u^- are the u value on the shock wave's right and left sides.

The solution to the Riemann problem

$$u(x,0) = \begin{cases} u_l \text{ if } x < 0, \\ u_r \text{ if } x > 0. \end{cases} \tag{4.2}$$

depends on the sign of $u_r - u_l$, where u^r and u^l are the right and left initial states:

- If $u_r > u_l$, we have a rarefaction wave separating the two initial states. The characteristic curves separating $U(\xi)$ from u_l and u_r are, respectively, $x = u_l t$ and $x = u_r t$.
- If $u_r < u_l$, we have a shock wave moving at the speed:

  $$\dot{\sigma} = \frac{1}{2}(u_r + u_l).$$

4.2 Approximate solvers

We have seen in § 2.4.1 Clawpack, the Riemann solver can expressed in terms of fluctuations:

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

where the fluctuations are

$$\mathcal{A}^+\Delta Q_{i-1/2} = f(Q_i) - f(Q^{\downarrow}_{i-1/2}),$$
$$\mathcal{A}^-\Delta Q_{i+1/2} = f(Q^{\downarrow}_{i+1/2}) - f(Q_i),$$

where $Q_{i\pm1/2}$ represents the value advected along the characteristic coming from the $x_{i\pm1/2}$ interface and $Q^{\downarrow}_{i+1/2} = q^{\downarrow}(Q_i, Q_{i-1}$ is the q value along the ray $x = x_{i-1/2}$.

There are five possible wave configurations (see Fig. 2.6): left- and right-going shock waves, left- and right-going rarefaction waves, and transonic rarefaction wave. When the solution to the Riemann problem is not a transonic wave, the idea is to approximate this solution as a shock wave even though it is a rarefaction wave. The shock $\mathcal{W}$ propagates at a speed s:

$$\mathcal{W}_{i-1/2} = Q_i - Q_{i-1},$$
$$s_{i-1/2} = \frac{f(Q_i) - f(Q_{i-1})}{Q_i - Q_{i-1}},$$

for $Q_i \neq Q_{i-1}$. We then deduce

$$\mathcal{A}^+\Delta Q_{i-1/2} = s_{i-1/2}\mathcal{W}_{i-1/2},$$
$$\mathcal{A}^-\Delta Q_{i-1/2} = s_{i-1/2}\mathcal{W}_{i-1/2},$$

When the solution to the Riemann problem is a transonic wave, we use the definition of the fluctuations

$$\mathcal{A}^+\Delta Q_{i-1/2} = f(Q_i) - f(q_s),$$
$$\mathcal{A}^-\Delta Q_{i-1/2} = f(q_s) - f(Q_{i-1}),$$

where q_s is the value such as $f'(q_s) = 0$ (vertical characteristic corresponding to $x - x_{i-1/2} = 0 \cdot t$). For the Burgers equation we have $q_s = 0$.

To summarize, we express the functions `amdp`, `apdp`, `s`, and `W` in Clawpack:

$$\mathcal{A}^+\Delta U_{i-1/2} = s_{i-1/2}\mathcal{W}_{i-1/2}, \tag{4.3}$$
$$\mathcal{A}^-\Delta U_{i-1/2} = s_{i-1/2}\mathcal{W}_{i-1/2}, \tag{4.4}$$

with

$$\mathcal{W}_{i-1/2} = U_i - U_{i-1}, \tag{4.5}$$
$$s_{i-1/2} = \frac{1}{2}(U_i + U_{i-1}), \tag{4.6}$$

but if $U_{i-1} < 0$ and $U_i > 0$ then

$$\mathcal{A}^+\Delta U_{i-1/2} = \frac{1}{2}U_i^2, \tag{4.7}$$
$$\mathcal{A}^-\Delta U_{i-1/2} = -\frac{1}{2}U_{i-1}^2, \tag{4.8}$$

In Clawpack, the treatment of the transonic wave is called an *entropy fix*, and its use in the Riemann solver is indicated through the Boolean variable `efix`. Note that in the present case, the Roe and HLL solvers provide the same approximate solution as above. Indeed, linearising the Burgers equation (4.1) yields:

$$\frac{\partial q}{\partial t} + \hat{q}\frac{\partial q}{\partial x} = 0, \tag{4.9}$$

where the advection velocity $\hat{q}$ is given by

$$\hat{q} = \frac{1}{2}(q_r + q_l)$$

so that the linearised Burgers solution provides the exact solution when the initial states u_r and u_l lie on the Hugoniot locus. For the HLL solver, we need to determine two waves, but the solution to one-dimensional equations such as the Burgers equation involves a single wave. We thus assume that one of the waves vanishes, while the other moves at speed s given by

$$f(q_r) - f(q_l) = s(q_r - q_l) \Rightarrow s = \frac{1}{2}(q_r + q_l).$$

For the same reason, the f-wave solver will provide the same result.

4.2.1 Implementation in Clawpack

The Riemann solver is just the transcript of Eqs. (4.3) to (4.8).

```fortran
c
c
      efix = .true.    !# Compute correct flux for transonic rarefactions
c
      do 30 i=2-mbc,mx+mbc
c
c        # Compute the wave and speed
c
         wave(i,1,1) = ql(i,1) - qr(i-1,1)
         s(i,1) = 0.5d0 * (qr(i-1,1) + ql(i,1))
c
c
c        # compute left-going and right-going flux differences:
c        ------------------------------------------------------
c
         amdq(i,1) = dmin1(s(i,1), 0.d0) * wave(i,1,1)
         apdq(i,1) = dmax1(s(i,1), 0.d0) * wave(i,1,1)
c
         if (efix) then
c           # entropy fix for transonic rarefactions:
            if (qr(i-1,1).lt.0.d0 .and. ql(i,1).gt.0.d0) then
               amdq(i,1) = - 0.5d0 * qr(i-1,1)**2
               apdq(i,1) =   0.5d0 * ql(i,1)**2
               endif
            endif
   30   continue
```

4.2.2 Implementation in Pyclaw

The Riemann solver is just the transcript of Eqs. (4.3) to (4.8).

```
def burgers_1D(q_l,q_r,aux_l,aux_r,problem_data):
    r"""
    Riemann solver for Burgers equation in 1d
    *problem_data* should contain -
     - *efix* - (bool) Whether a entropy fix should be used, if not present
   ,
       false is assumed
    """

    num_rp = q_l.shape[1]
    # Output arrays
    wave = np.empty( (num_eqn, num_waves, num_rp) )
    s = np.empty( (num_waves, num_rp) )
    amdq = np.empty( (num_eqn, num_rp) )
    apdq = np.empty( (num_eqn, num_rp) )

    # Basic solve
    wave[0,:,:] = q_r - q_l
    s[0,:] = 0.5 * (q_r[0,:] + q_l[0,:])

    s_index = np.zeros((2,num_rp))
    s_index[0,:] = s[0,:]
    amdq[0,:] = np.min(s_index,axis=0) * wave[0,0,:]
    apdq[0,:] = np.max(s_index,axis=0) * wave[0,0,:]

    # Compute entropy fix
    if problem_data['efix']:
        transonic = (q_l[0,:] < 0.0) * (q_r[0,:] > 0.0)
        amdq[0,transonic] = -0.5 * q_l[0,transonic]**2
        apdq[0,transonic] = 0.5 * q_r[0,transonic]**2

    return wave, s, amdq, apdq
```

In this routine, `ql` is the left initial condition. It is a $m \times N$ array ($m = 1$ the problem dimension, and N the number of cells). So, `num_rp␣=␣q_l.shape[1]` gives N. First, the `amdp`, `apdp`, `s`, and `W` are initialised, then `s` and `W` are defined. Finally, the fluctuations are defined using the numpy function `numpy.min`, which provides the minimum value: absolute, for each column (with the `axis=0` option), or for each row (with the `axis=1` option). For instance, the lines

```
import numpy as np
x=np.array([[1,4],[2,5],[3,-2]])
np.min(x,axis=None)
np.min(x,axis=0)
np.min(x,axis=1)
```

provide the values: -2, `array([␣1,␣-2])` and `array([␣1,␣2,␣-2])`, respectively.

4.3 Examples

4.3.1 *Solution to the Riemann problem*

Here we solve the Riemann problem and compare the numerical and exact solutions (the exact solution is outlined in § 4.1). The code is initialised as follows:

```
%matplotlib inline
```

```python
import numpy as np
import matplotlib.pyplot as plt

from numpy import sqrt, log
from clawpack import riemann
from clawpack import pyclaw

def qsol(x,t,ql,qr):
    """
    The initial and true solution.
    """
    import numpy as np
    dim = x.shape[0]
    q = np.empty(dim)

    if qr < ql:
        s = (qr+ql)/2
        for i in range(dim):
            if x[i]>=s*t:
                q[i] = qr
            else:
                q[i] = ql
    else:
        if t > 0:
            xr = qr*t
            xl = ql*t
            for i in range(dim):
                if x[i]>=xl and x[i]<=xr:
                    q[i] = x[i]/t
                elif x[i]>xr:
                    q[i] = qr
                elif x[i]<xl:
                    q[i] = ql
        else:
            for i in range(dim):
                if x[i]>=0:
                    q[i] = qr
                else:
                    q[i] = ql
    return q

def setup(ql,qr):

    solver = pyclaw.ClawSolver1D()
    solver.rp = riemann.burgers_1D_py.burgers_1D
    solver.num_waves = 1
    solver.num_eqn = 1
    solver.kernel_language = 'Python'
    solver.limiters = pyclaw.limiters.tvd.superbee
    solver.bc_lower[0] = pyclaw.BC.extrap
    solver.bc_upper[0] = pyclaw.BC.extrap
    solver.order = 2 #1: Godunov, 2: Lax-Wendroff-LeVeque

    x = pyclaw.Dimension(-1, 1., 200, name='x')
    domain = pyclaw.Domain(x)
    num_eqn = 1

    state = pyclaw.State(domain, num_eqn)
    state.problem_data['efix'] = True

    xc = domain.grid.x.centers

    state.q[0, :] = qsol(xc,0,ql,qr)

    claw = pyclaw.Controller()
    claw.solution = pyclaw.Solution(state, domain)
    claw.solver = solver
    claw.outdir = './_output'
    claw.output_style = 1
    claw.tfinal = 2.0
    claw.num_output_times = 20
    claw.keep_copy = True
    return claw
```

We run the code for different initial conditions.

```
claw = setup(2,0)
claw.run()

plt.rcParams['text.usetex'] = True
index = 3
frame = claw.frames[index]
dt = claw.tfinal/claw.num_output_times
t = dt*index
x = frame.state.grid.c_centers
x = x[0]
true = qsol(x,t,2,0)
fig, ax = plt.subplots(figsize=(3,3))
w = frame.q[0,:]

ax.plot(x, w, label='Clawpack sol.')
ax.plot(x, true, ':',label='exact solution')
ax.legend(loc='lower left')
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$q$')
plt.text(-1,1.5,'(a)' )
plt.savefig("ShockBurgerClaw.pdf",bbox_inches='tight')
```

Figure 4.1 shows the numerical and solutions at time $t = 0.3$ s. We considered the case: (a) shock wave with $(u_l = 2, u_r = 0)$; (a) rarefaction wave with $(u_l = 0, u_r = 2)$; (a) transonic wave with $(u_l = -1, u_r = 1)$.

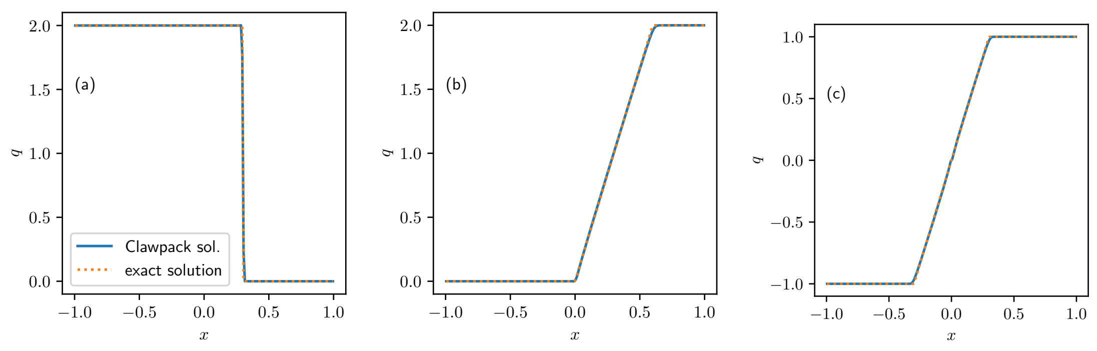


Figure 4.1 Solutions to the Riemann problem: (a) shock wave; rarefaction wave (b) ; (c) transonic rarefaction wave. Lax–Wendroff–LeVeque solver with $\Delta x = 10^{-2}$ m.

4.3.2 *Solution to an initial value problem*

Let us now consider a more complicated initial-value problem:

$$u(x,0)=u_0(x)=\begin{cases}0 & \text{if } x<0 \text{ or } x>b,\\ ax & \text{if } 0\le x\le b,\end{cases}$$

where a and b are two constants. Using the method of characteristics, we can cast Eq. (4.1) in the form:

$$\frac{\mathrm{d}u}{\mathrm{d}t}=0 \text{ along } \frac{\mathrm{d}x}{\mathrm{d}t}=u, \tag{4.10}$$

which shows that the characteristics are straight lines in the form

$$x=x_0+ut,$$

and since $u(x,t)$ is constant along this curve and equal to $u(x,0)=u_0(x_0)$, we deduce that $u(x,t)$ is the solution to

$$u_0(x_0)=a(x-ut)=u\Rightarrow u=\frac{ax}{at+1}.$$

The initial condition has a finite support with an initial discontinuity at $x=b$. Since the right state is lower than the left state, this initial discontinuity will propagate as a shock wave $x=s(t)$, whose velocity is given by the Rankine–Hugoniot equation:

$$\dot{s}=\frac{[\![f(u)]\!]}{[\![u]\!]}=\frac{1}{2}\frac{u^2(s(t),t)}{u(s(t),t)}=\frac{1}{2}u(s(t),t),$$

subject to $s(0)=b$. The solution is

$$s(t)=b\sqrt{1+at}.$$

The exact solution to Eq. (4.1) is thus

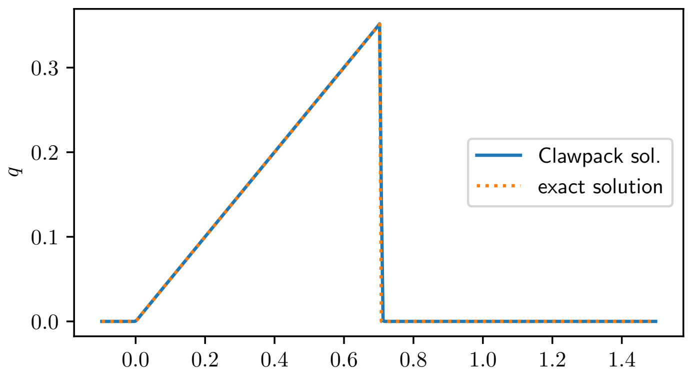
$$u(x,t)=\begin{cases}0 & \text{if } x<0 \text{ or } x>s(t),\\ \dfrac{ax}{at+1} & \text{if } 0\le x\le s(t)\end{cases} \tag{4.11}$$

We initialise the code with $a=1$ and $b=1/2$.

```python
%matplotlib inline

from numpy import sqrt, log
from clawpack import riemann
from clawpack import pyclaw

def xfr(t): return sqrt(1+t)/2

def qtrue(x,t):
    """
    The true solution, for comparison.
    """
    import numpy as np
    dim = x.shape[0]
    q = np.empty(dim)

    xf = xfr(t)
    for i in range(dim):
        if x[i]>=0.0 and x[i]<=xf:
            q[i] = (x[i]) /(t+1)
        else:
                q[i] = 0
    return q

def burgers(q_l,q_r,aux_l,aux_r,problem_data):
    r"""
    1d burgers riemann solver
    """
    import numpy as np
    num_eqn = 1
    num_waves = 1

    # Convenience
    num_rp = q_l.shape[1]

    # Return values
    wave = np.empty( (num_eqn, num_waves, num_rp) )
    s = np.empty( (num_waves, num_rp) )
    amdq = np.empty( (num_eqn, num_rp) )
    apdq = np.empty( (num_eqn, num_rp) )

    # Local values
    delta = np.empty(np.shape(q_l))
    delta = q_r - q_l

    # Compute the wave
    # 1-Wave
    wave[0,0,:] = delta
    s[0,:] = 0.5 * (q_r[0,:] + q_l[0,:])

    # Compute the left going and right going fluctuations
    s_index = np.zeros((2,num_rp))
    s_index[0,:] = s[0,:]
    amdq[0,:] = np.min(s_index,axis=0) * wave[0,0,:]
    apdq[0,:] = np.max(s_index,axis=0) * wave[0,0,:]

    # Compute entropy fix
    if problem_data['efix']:
        transonic = (q_l[0,:] < 0.0) * (q_r[0,:] > 0.0)
        amdq[0,transonic] = -0.5 * q_l[0,transonic]**2
        apdq[0,transonic] = 0.5 * q_r[0,transonic]**2

    return wave, s, amdq, apdq


def setup(outdir='./_output',  output_style=1):

    solver = pyclaw.ClawSolver1D()
    solver.rp = burgers

    solver.num_waves = 1
    solver.num_eqn = 1
    solver.kernel_language = 'Python'
    solver.limiters = pyclaw.limiters.tvd.superbee
    solver.bc_lower[0] = pyclaw.BC.extrap
    solver.bc_upper[0] = pyclaw.BC.extrap
    solver.order = 2 #1: Godunov, 2: Lax-Wendroff-LeVeque

    x = pyclaw.Dimension(-0.10, 1.5, 320, name='x')
    domain = pyclaw.Domain(x)
    xc = domain.grid.x.centers
    num_eqn = 1

    state = pyclaw.State(domain, num_eqn)
    state.problem_data['efix'] = True
    state.q[0, :] = qtrue(xc, 0)

    claw = pyclaw.Controller()
    claw.solution = pyclaw.Solution(state, domain)
    claw.solver = solver
    claw.outdir = outdir
    claw.output_style = output_style
    claw.tfinal = 2.0
    claw.num_output_times = 20
    claw.keep_copy = True

    return claw
```

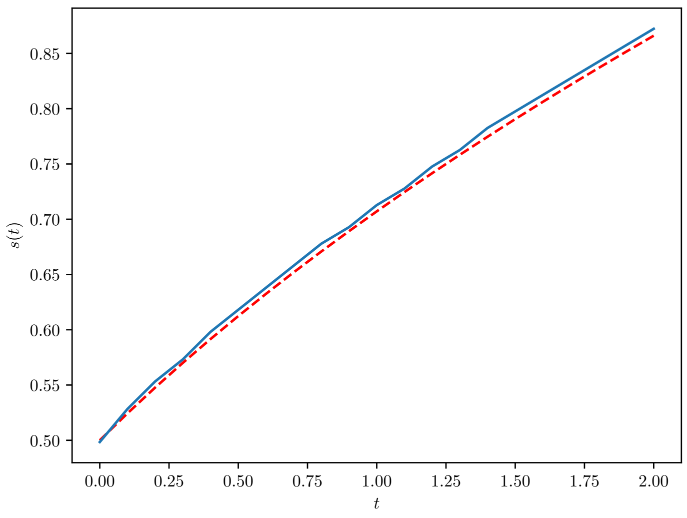
We plot the numerical solution at time $t = 1$ s.

```
claw = setup()
claw.run()
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['text.usetex'] = True
index = 10
frame = claw.frames[index]
dt = claw.tfinal/claw.num_output_times
t = dt*index
x = frame.state.grid.c_centers
x = x[0]
true = qtrue(x,t)
fig, ax = plt.subplots(figsize=(5, 2.7))
w = frame.q[0,:]

ax.plot(x, w, label='Clawpack sol.')
ax.plot(x, true, ':',label='exact solution')
ax.legend(loc='right')
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$q$')
plt.savefig("InitialValueBurgerClaw.pdf")
```



We can also determine the front position $x = s(t)$ in the numerical solution and compare it with the exact solution. We compute the shock position for a set of discrete times for which the numerical solution was recorded. A more accurate representation of $s(t)$ could have been obtained by using the function `b4step` (see the example of § 4.5).

```python
nsimul=np.size(claw.frames)
positionFront = []
for i in range(nsimul):
    frame = claw.frames[i]
    w = frame.q[0,:]
    xc=frame.state.grid.c_centers[0]
    nx=xc.size
    dx=(xc[-1]-xc[0])/nx
    pos= dx*w[w>1.e-5].size
    positionFront.append(pos)

t = np.linspace(0, claw.tfinal, nsimul)
xfront = xfr(t)

fig, ax = plt.subplots()
ax.set_xlabel(r'$t$')
ax.set_ylabel(r'$s(t)$')

ax.plot(t, xfront, '--',color = 'r')
ax.plot(t, positionFront)
plt.savefig("frontBurgerClaw.pdf")
```



4.4 Nonlinear advection equation with a diffusive term

4.4.1 Theoretical considerations

Let us consider the Burgers equation with a diffusive term representing viscous effect:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \tag{4.12}$$

where ν is the dynamic viscosity. This equation is subject to an initial condition:

$$u(x,0) = u_0(x), \tag{4.13}$$

and the following boundary conditions:

$$\lim_{x\to\pm\infty} \partial_x u(x,t) = 0. \tag{4.14}$$

General solutions

Exact solutions can be worked out using the Cole–Hopf transformation. To that end, let us first define the primitive of $u(x,t)$ with respect to x and denoted by $U(x,t)$. We have $u = \partial_x U$. Substituting this new variable into Eq. (4.12), integrating it and taking the boundary conditions (4.14) leads to

$$\frac{\partial U}{\partial t} + \frac{1}{2}\left(\frac{\partial U}{\partial x}\right)^2 = \nu\frac{\partial^2 U}{\partial x^2}, \tag{4.15}$$

subject to $U(x,0) = U_0(x)$ where U_0 is the primitive of U_0. We now use the new variable ϕ in the Cole–Hopf transformation:

$$U(x,t) = -2\nu \ln \phi(x,t)$$

Substituting this form into Eq. (4.15) gives the simple linear diffusion equation

$$\frac{\partial \phi}{\partial t} = \nu\frac{\partial^2 \phi}{\partial x^2}, \tag{4.16}$$

subject to

$$\phi(x,0) = \exp\left(-\frac{1}{2\nu}U_0(x)\right).$$

The general solution of Eq. (4.16) is the convolution of the initial value $\phi(x,0)$ with the kernel function $K(x,t)$

$$\phi(x,t) = \int_{-\infty}^{+\infty} K(x-y,t)\phi(y,0)\mathrm{d}y \text{ where } K(y,t) = \frac{1}{\sqrt{4\pi\nu t}}\exp\left(-\frac{y^2}{4\nu t}\right) \tag{4.17}$$

The solution to Eq. (4.15) is thus

$$U(x,t) = -2\nu\ln\int_{-\infty}^{+\infty}\frac{1}{\sqrt{4\pi\nu t}}\exp\left(-\frac{(x-y)^2}{4\nu t} - \frac{1}{2\nu}U_0(y)\right)\mathrm{d}y. \tag{4.18}$$

By differentiating this equation with to respect to x, we obtain the general solution $u(x,t)$

$$u(x,t) = \frac{\partial U}{\partial x} = \frac{\int_{-\infty}^{+\infty}\frac{x-y}{t}\exp\left(-\frac{(x-y)^2}{4\nu t} - \frac{1}{2\nu}U_0(y)\right)\mathrm{d}y}{\int_{-\infty}^{+\infty}\exp\left(-\frac{(x-y)^2}{4\nu t} - \frac{1}{2\nu}U_0(y)\right)\mathrm{d}y}. \tag{4.19}$$

It is difficult to go further with this equation, but it can be evaluated numerically and provide an approximation to the exact solution.

Travelling-wave solutions

It possible to work out exact solutions to the viscous Burgers equation (4.12) by noting that this equation is invariant to the translation group $x \to x + s\lambda$ and $t \to t + \lambda$, which indicates that there must be a travelling-wave solution in the form:

$$u(x,t) = U(\xi) \text{ where } \xi = x - st.$$

Substituting this form into the governing equation (4.12) leads to the principal differential equation

$$-s\frac{\mathrm{d}U}{\mathrm{d}\xi} + U\frac{\mathrm{d}U}{\mathrm{d}\xi} = \nu\frac{\mathrm{d}^2U}{\mathrm{d}\xi^2}, \tag{4.20}$$

which can be integrated to give

$$-sU + \frac{1}{2}U^2 = \nu\frac{\mathrm{d}U}{\mathrm{d}\xi} + c, \tag{4.21}$$

where c is constant of integration. We assume that the governing equation (4.12) is supplemented by the boundary conditions

$$\lim_{x\to-\infty} u(x,t) = u_l \text{ and } \lim_{x\to+\infty} u(x,t) = u_r$$

Using these boundary conditions shows that the constant of integration satisfies the equations

$$c - su_r + \frac{1}{2}u_r^2 = c - su_l + \frac{1}{2}u_l^2,$$

which is possible only if

$$s = \frac{u_l + u_r}{2},$$

and thus

$$c = -\frac{u_l u_r}{2}.$$

We can recast Eq. (4.21)

$$\begin{aligned}\frac{1}{2\nu} &= \frac{\mathrm{d}U}{U^2 - 2sU - c},\\ &= \frac{\mathrm{d}U}{(U-s)^2 - b^2} \text{ where } b^2 = s^2 + c = \left(\frac{u_l - u_r}{2}\right)^2,\end{aligned}$$

whose integral is

$$\operatorname{arctanh}\frac{s-u}{b} = |b|\frac{\xi}{2\nu} \Rightarrow u = s - |b|\tanh\left(b\frac{\xi}{2\nu}\right) + a,$$

where a is a constant of integration found to be zero. The travelling-wave solution is therefore:

$$u = \frac{u_l + u_r}{2} - \left|\frac{u_l - u_r}{2}\right| \tanh\left(\frac{u_l - u_r}{4\nu}\left(x - \frac{u_l + u_r}{2}t\right)\right). \tag{4.22}$$

4.4.2 Numerical implementation in Clawpack

The scripts for solving the Burgers equation (4.12) are the same as those for solving the inviscid case in § 4.2.1 except for the source term, where we implement the Crank–Nicolson method (see § 2.9.3):

$$Q_i^{n+1} = Q_i^* + \nu\frac{\Delta t}{2}\left(\frac{Q_{i-1}^{n+1} - 2Q_i^{n+1} + Q_{i+1}^{n+1}}{\Delta x^2} + \frac{Q_{i-1}^* - 2Q_i^* + Q_{i+1}^*}{\Delta x^2}\right). \tag{4.23}$$

We assume a no-flux condition at the domain boundaries, and thus $Q_1^{n+1} = Q_1^{n+1}$ and $Q_m^{n+1} = Q_{m+1}^{n+1}$. Let us define the ratio r

$$r = \nu \frac{\Delta t}{2\Delta x^2}.$$

We have to solve the linear system

$$\begin{bmatrix} 1+r & -r & 0 & \dots & 0 \\ -r & \ddots & & 0 & \\ \dots & -r & 1+2r & -r & \dots \\ & & & \ddots & \\ 0 & \dots & 0 & -r & 1+r \end{bmatrix} \cdot \begin{bmatrix} Q_1^{n+1} \\ Q_2^{n+1} \\ \vdots \\ Q_i^{n+1} \\ \vdots \\ Q_m^{n+1} \end{bmatrix} = \begin{bmatrix} (1-r)Q_1^n + rQ_2^n \\ Q_1^n + (1-2r)Q_2^n + rQ_3^n \\ \vdots \\ Q_{i-1}^n + (1-2r)Q_i^n + rQ_{i+1}^n \\ \vdots \\ (1-r)Q_m^n + rQ_{m-1}^n \end{bmatrix}$$

The file `src1.f` solves this system. It calls the LAPACK routine `dgtsv` for inverting tridiagonal matrices (included in the file `tridiag.f`). Makefile must be updated by adding `src1.f` and `tridiag.f`.

```
c ==========================================================
      subroutine src1(meqn,mbc,mx,xlower,dx,q,maux,aux,t,dt)
c ==========================================================
      implicit double precision (a-h,o-z)
      dimension q(meqn,1-mbc:mx+mbc)
c
c     # solve the diffusion equation q_t = q_{xx} using Crank-Nicolson
c     # The LAPACK tridiagonal solver dgtsv is used, which is in tridiag.f
c     # local storage:
      parameter (maxldb = 5000)
      dimension b(maxldb,1), d(maxldb), dl(maxldb), du(maxldb)
      common /comsrc/ dcoef

      if (mx .gt. maxldb) then
          write(6,*) 'ERROR:  increase maxldb in src1.f'
          endif
      ldb = maxldb
      nrhs = 1
      dtdx2 = dcoef * dt / (2.d0*dx*dx)
c     # set coefficients in tridiagonal matrix and RHS:
      do i=1,mx
        dl(i) = -dtdx2
        d(i) = 1.d0 + 2.d0*dtdx2
        du(i) = -dtdx2
        b(i,1) = q(1,i) + dtdx2 * (q(1,i-1) - 2.d0*q(1,i) + q(1,i+1))
        enddo
c     # no-flux boundary conditions for diffusion step:
c     # Adjust matrix entries to use q(1,0)=q(1,1) and q(1,mx+1)=q(1,mx)
c     # at end of time step:
      d(1) = d(1) - dtdx2
      d(mx) = d(mx) - dtdx2
c     # solve the tridiagonal system:
      call dgtsv(mx,nrhs,dl,d,du,b,ldb,info)

      if (info .ne. 0) then
         write(6,*) 'ERROR in src1 from call to dgtsv... info = ',info
         stop
         endif

      do i=1,mx
         q(1,i) = b(i,1)
    enddo
      return
      end
```

We first test the code to the unit box case. The initial condition is

$$u(x,0) = \begin{cases} 0 & \text{if } x < -\frac{1}{2}, \\ 1 & \text{if } -\frac{1}{2} \le x \le \frac{1}{2}, \\ 0 & \text{if } x > \frac{1}{2}. \end{cases}$$

The file `qinit.f` encodes this initial condition.

```
c =====================================================
       subroutine qinit(meqn,mbc,mx,xlower,dx,q,maux,aux)
c =====================================================
c     # Set initial conditions for q.
c
      implicit double precision (a-h,o-z)
      dimension q(meqn,1-mbc:mx+mbc)
      dimension aux(maux,1-mbc:mx+mbc)
      common /comsrc/ dcoef
c
      do 150 i=1,mx
   xcell = xlower + (i-0.5d0)*dx
c        # unit box
         q(1,i) = 1.d0
         if (xcell .lt. -0.5d0) q(1,i) = 0.d0
         if (xcell .gt. 0.5d0) q(1,i) = 0.d0
  150    continue
      return
      end
```

Figure 4.2 compares the numerical solution with the exact solution computed using Eq. (4.19) (evaluated numerically).

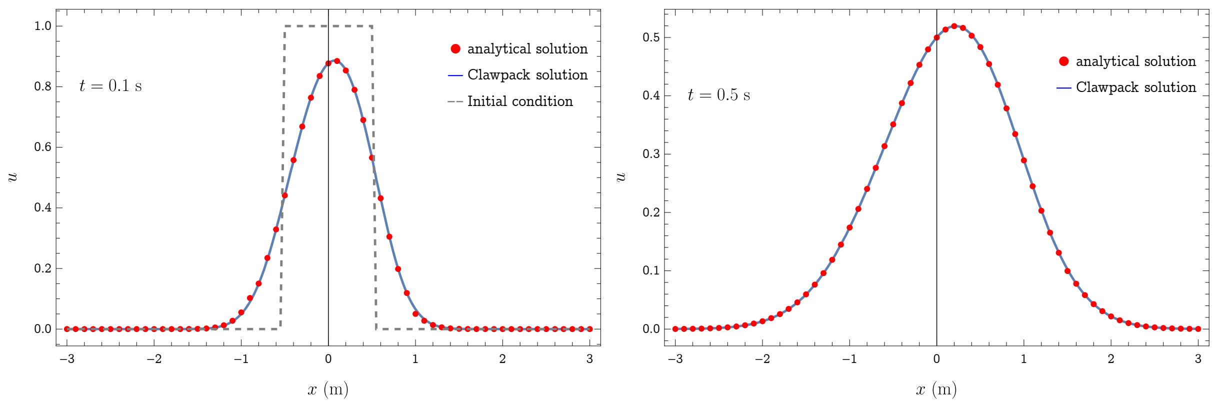


Figure 4.2 Snapshots of the solution at time $t = 0.1$ s and $t = 0.5$. Parameter $\nu = 0.5$ m^2/s.

We now consider the travelling-wave case. The initial state is calculated using Eq. (4.22) evaluated at $t = 0$:

$$u(x,0) = \frac{u_l + u_r}{2} - \left|\frac{u_l - u_r}{2}\right| \tanh\left(\frac{u_l - u_r}{4\nu} x\right).$$

The file `qinit.f` initiates the travelling wave solution by using (4.22) at time $t = 0$.

```fortran
c ==========================================================
       subroutine qinit(meqn,mbc,mx,xlower,dx,q,maux,aux)
c ==========================================================
c     # Set initial conditions for q.
c
      implicit double precision (a-h,o-z)
      dimension q(meqn,1-mbc:mx+mbc)
      dimension aux(maux,1-mbc:mx+mbc)
      common /comsrc/ dcoef

      do 150 i=1,mx
    xcell = xlower + (i-0.5d0)*dx
      ul = 0.d0
    ur = 2.d0
      q(1,i) = (ur+ul)/2-dabs((ul-ur))/2.d0*tanh(xcell*(ul-ur)/
     &                  4.d0/dcoef)
  150    continue
c
      return
      end
```

Figure 4.3 compares the numerical solution with the exact solution computed using Eq. (4.22).

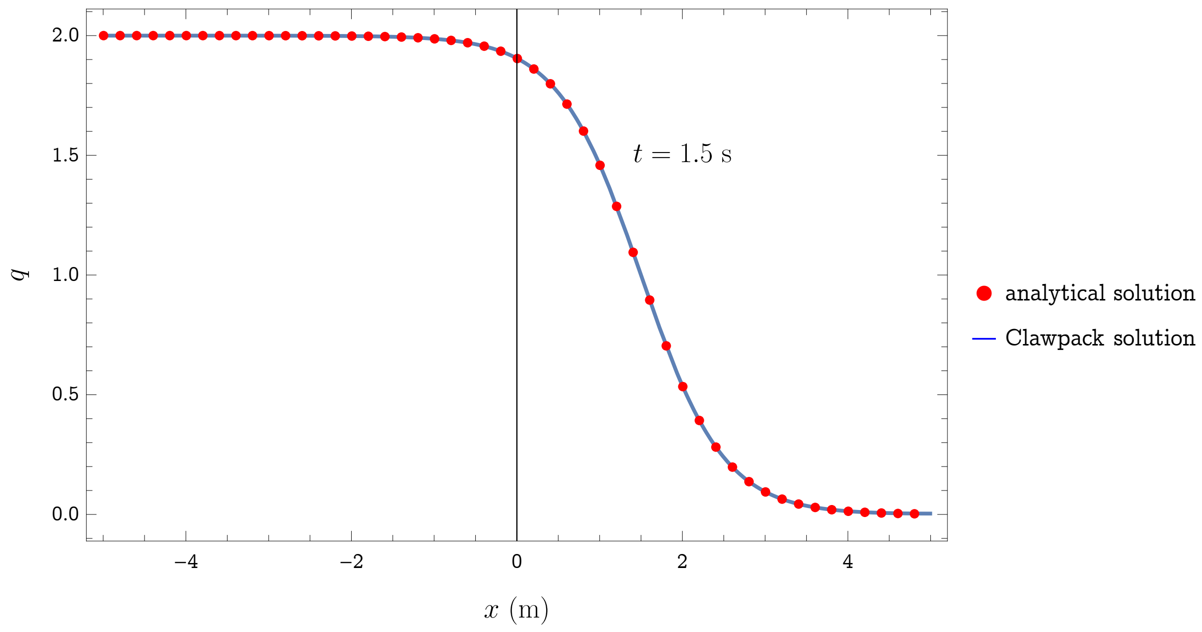


Figure 4.3 Snapshots of the solution at time $t = 0.1$ s and $t = 0.5$. Parameters $\nu = 0.5$ m^2/s, $u_l = 2$, and $u_r = 0$. Lax–Wendroff scheme (limiter: superbee) with $\Delta x = 10^{-2}$ m.

4.4.3 *Numerical implementation in Pyclaw*

We implemented the Crank–Nicolson method in the previous Pyclaw script used for solving the inviscid case. For the application, we used the same parameters as above: $\nu = 0.5$ m^2/s, $u_l = 2$, and $u_r = 0$. We use the built-in Scipy function `solve_banded`, which provides the numerical solution to $\boldsymbol{B} \cdot \boldsymbol{q}^{n+1} = \boldsymbol{F}(\boldsymbol{q}^n)$, where $\boldsymbol{B}$ is a band matrix.

```
%matplotlib inline

from numpy import sqrt, log, tanh
from clawpack import riemann
from clawpack import pyclaw

# Travelling wave solution
def travel(x,t,ul,ur,nu): return (ur+ul)/2-abs((ul-ur))/2*tanh((x-(ur+ul)
    /2*t)*(ul-ur)/4/nu )

def qtrue(x,t,ul,ur,nu):
    """
    The true solution, for comparison.
    """
    import numpy as np
    dim = x.shape[0]
    q = np.empty(dim)

    for i in range(dim):
            q[i] = travel(x[i],t,ul,ur,nu)
    return q



def source_term(solver, state, dt):
    from scipy.linalg import solve_banded
    import numpy as np
    qs = state.q[0,:]
    xc = state.grid.c_centers[0]
    nx = xc.size
    dx = (xc[-1]-xc[0])/nx
    nu = state.problem_data['nu']
    r = nu*dt/(dx**2*2.)
    m = qs.shape[0]
    b = np.empty(m)
    for i in range(m):
        if i == 0:
            b[0] = (1-r)*qs[0]+r*qs[1]
        elif i == m-1:
            b[m-1] = (1-r)*qs[m-1]+ r*qs[m-2]
        else:
            b[i] = r*qs[i-1]+(1-2*r)*qs[i]+r*qs[i+1]

    ab = np.empty((3,m))
    ab[0,1:] = [-r]*(m-1)
    ab[1,:] = [1+2*r]*m
    ab[2,:-1] = [-r]*(m-1)
    ab[1,0] = 1+r
    ab[1,-1]=1+r
    state.q[0,:] = solve_banded((1,1),ab,b)


def setup(outdir='./_output',  output_style=1):

    solver = pyclaw.ClawSolver1D()
    solver.rp = riemann.burgers_1D_py.burgers_1D

    solver.num_waves = 1
```

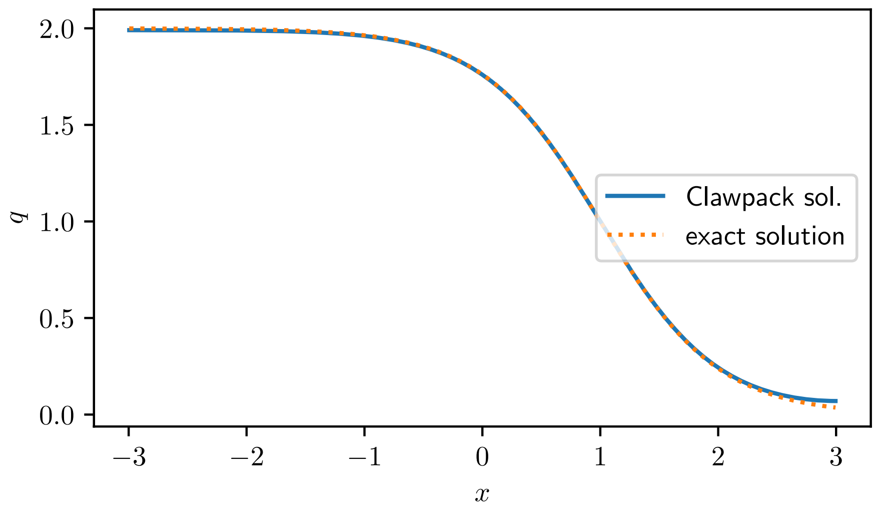
```python
    solver.num_eqn = 1
    solver.kernel_language = 'Python'
    solver.limiters = pyclaw.limiters.tvd.superbee
    solver.bc_lower[0] = pyclaw.BC.extrap
    solver.bc_upper[0] = pyclaw.BC.extrap
    solver.order = 2 #1: Godunov, 2: Lax-Wendroff-LeVeque

    solver.step_source = source_term # inclusion of source term

    mx = 600
    x = pyclaw.Dimension(-3, 3, mx, name='x')
    domain = pyclaw.Domain(x)
    num_eqn = 1

    state = pyclaw.State(domain, num_eqn)

    # Parameters
    nu = 0.5
    ul = 2
    ur = 0
    state.problem_data['efix'] = True
    state.problem_data['nu'] = nu #diffusivity
    state.problem_data['ul'] = ul #left state
    state.problem_data['ur'] = ur #right state

    xc = domain.grid.x.centers

    #for i in range(mx):
            #state.q[0,i] = travel(xc[i],0,ul,ur,nu)   #initial condition

    state.q[0,:] = travel(xc,0,ul,ur,nu)

    claw = pyclaw.Controller()
    claw.solution = pyclaw.Solution(state, domain)
    claw.solver = solver
    claw.outdir = outdir
    claw.output_style = output_style
    claw.tfinal = 2.0
    claw.num_output_times = 20
    claw.keep_copy = True

    return claw
```

We ran the code and plotted the solution for time $t = 1$ s.

```python
claw = setup()
claw.run()
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['text.usetex'] = True
index = 10
frame = claw.frames[index]
dt = claw.tfinal/claw.num_output_times
t = dt*index
x = frame.state.grid.c_centers
x = x[0]

ul = frame.state.problem_data['ul']
```

```python
ur = frame.state.problem_data['ur']
nu = frame.state.problem_data['nu']
true = qtrue(x,t,ul,ur,nu)
fig, ax = plt.subplots(figsize=(5, 2.7))
w = frame.q[0,:]

ax.plot(x, w, label='Clawpack sol.')
ax.plot(x, true, ':',label='exact solution')
ax.legend(loc='right')
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$q$')
plt.savefig("ViscousBurgerClaw.pdf")
```

Figure 4.3 compares the numerical solution with the exact solution computed using Eq. (4.22) at time $t = 1$ s.

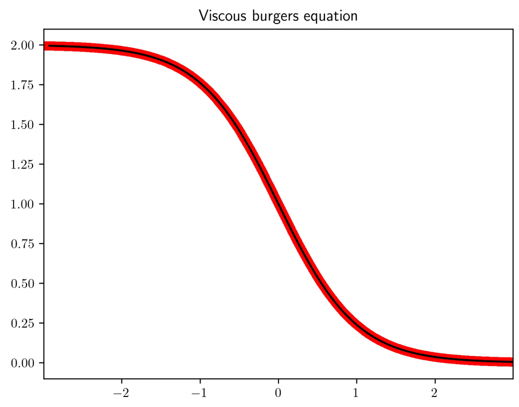


Here is an animation:

```python
nsimul=np.size(claw.frames)
figs = []
for i in range(nsimul):
    fig, ax = plt.subplots(figsize=(5, 3))


    frame = claw.frames[i]
    w = frame.q[0,:]
    x = frame.state.grid.c_centers
    x = x[0]
    dt = claw.tfinal/claw.num_output_times
    t = dt*i
    true = qtrue(x,t,ul,ur,nu)
    ax.set_xlabel(r'$x$')
    ax.set_ylabel(r'$q$')
    plt.plot(x, w)
    plt.plot(x, true, '--',color = 'r')
    figs.append(fig)
    plt.close(fig)

from clawpack.visclaw import animation_tools
animation_tools.interact_animate_figs(figs)
```

We can export an animation of the solution at different times and compare with the exact solution using `FuncAnimation` from the library `matplotlib.animation`.

```python
 def q_true(t,ul,ur,nu):
    import numpy
    # True Solution
    x_true = numpy.linspace(-3.0, 3.0, 1000)
    q_true = travel(x_true,t,ul,ur,nu)
    return q_true

def burgers_animation(ul=2, ur=0, nu=0.5):
    import matplotlib.animation
    import numpy
    # compute the solution with the method define above:
    claw = setup()
    claw.keep_copy = True
    claw.run()
    x = claw.frames[0].grid.dimensions[0].centers
    x_true = numpy.linspace(-3.0, 3.0, 1000)

    fig = plt.figure()
    axes = plt.subplot(1, 1, 1)
    axes.set_xlim((x[0], x[-1]))
    axes.set_ylim((-0.1, 2))
    axes.set_title("Viscous burgers equation")

    def init():
        axes.set_xlim((x[0], x[-1]))
        axes.set_ylim((-0.1,2.1))
        computed_line, = axes.plot(x[0], claw.frames[0].q[0, :][0], 'ro')
        exact_line, = axes.plot(x_true[0], q_true(0.0,ul,ur,nu)[0], 'k')
        return (computed_line, exact_line)

    computed_line, exact_line = init()

    def fplot(n):
        computed_line.set_data([x,], [claw.frames[n].q[0, :]])
        exact_line.set_data([x_true], [q_true(claw.frames[n].t,ul,ur,nu)])
        return (computed_line, exact_line)

    frames_to_plot = range(0, len(claw.frames))
    plt.close(fig)
    return matplotlib.animation.FuncAnimation(fig, fplot, frames=
   frames_to_plot, interval=100,
                                               blit=True, init_func=init, repeat=False)

from IPython.display import HTML
anim = burgers_animation()
HTML(anim.to_jshtml())
```



We can export the animation using `save` as a video using `save`:

```
anim.save('ViscousBurgers.mp4',fps=5,writer="ffmpeg",dpi=300)
```

It may be useful to export a series of images instead of a video. This can be achieved by different means. A possibility involves the `imagemagick` library:

```
anim.save('ViscousBurgers.png', writer="imagemagick",dpi=300)
```

In the command prompt, we can ask `imagemagick` to decompose the animation into a series of images

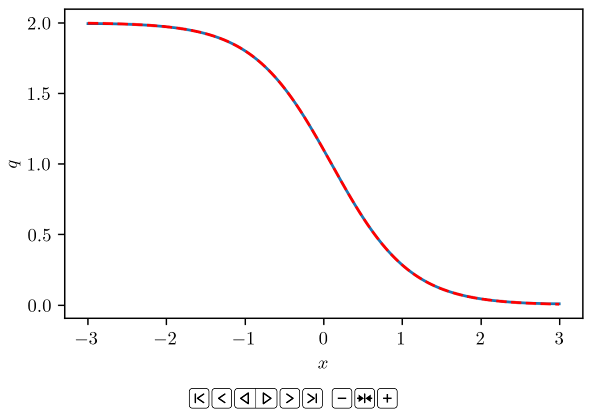
```
convert -coalesce ViscousBurgers.mp4 ViscousBurgers_.png
```

It may be more convenient to export directly the figures as pdf or png files:

```
for i in range(len(figs)):
    figs[i].savefig('ViscousBurgersFrame'+str(i)+'.png', bbox_inches='tight
   ',dpi=300)
```

This series of images can be inserted in a pdf file and produce animations, for instance by using the `animate` and `tikz` latex libraries.

```
 \animategraphics[autoplay,loop,width=8cm, controls={play, step, stop,
   speed},buttonsize=0.3cm]{5}{ViscousBurgersFrame}{1}{20}
```



4.5 Nonlinear advection equation with a source term

4.5.1 *Theoretical considerations*

Let us consider a rainfall of intensity I over a sloping bed inclined at α (see Fig. 4.4). There are two possible runoff mechanisms: superficial or hyporheic flow. For both cases, we assume that the flow depth is $h(x,t)$ and velocity $u(x,t)$ related to h: $u = ah^b$, where a and b are two coefficients: $a = C\sqrt{\alpha}$ et $b = 1/2$ if one considers runoff with a Chézy friction C.

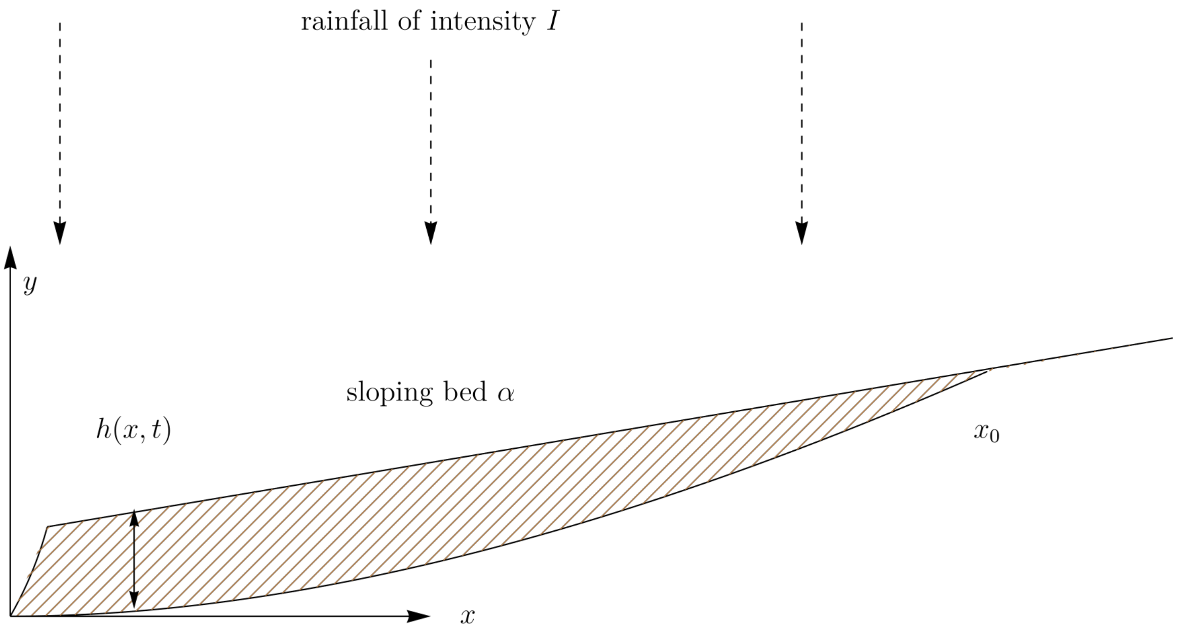


Figure 4.4 Flow generated by a rainfall.

The governing equation is given by mass conservation:

$$\frac{\partial h}{\partial t} + \frac{\partial hu}{\partial x} = I. \tag{4.24}$$

As we have $u = ah^b$, we obtain:

$$\frac{\partial h}{\partial t} + c(h)\frac{\partial h}{\partial x} = I \text{ with } c(h) = a(b+1)h^b.$$

or in a characteristic form:

$$\frac{\mathrm{d}h}{\mathrm{d}t} = I \text{ along } \frac{\mathrm{d}x}{\mathrm{d}t} = c(h) = a(b+1)h^b,$$

and we assume that initially the flow depth is zero (dry bed: $h(x,0) = 0$) and no water comes from upstream of x_0 ($h(x_0, t) = 0$). The solution to the characteristic equation is $h = It$ along the characteristic curve:

$$x = \int a(b+1)h^b \mathrm{d}t + x_1 = aI^b t^{1+b} + x_1, \tag{4.25}$$

where x_1 is constante of integration (such that at $x = x_1$, we have $h = 0$). This implies for any x ($0 \le x \le x_0$, with the frame used in Fig. 4.4, $x_0 = L_0$), we have:

- a linear growth $h(x,\, t) = It$ until time t_∞ such that $aI^b t_\infty^{1+b} = x_0 - x$;
- a stationary state for:

$$h(x,\, t) = h_\infty(x) = \left(\frac{I(x_0 - x)}{a}\right)^{1/(1+b)} .$$

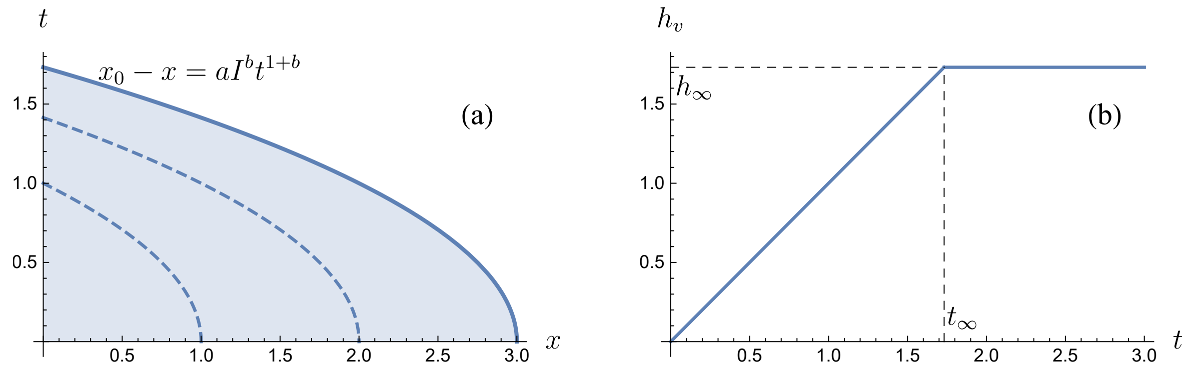


Figure 4.5 (a) characteristic curves (4.25). The thick line represents $x = aI^b t^{1+b}$, the path of a fluid parcel emitted from x_0. The coloured area represents the domain controlled by the initial condition $h = 0$ for which we observe a linear growth $h(x,\, t) = It$. Above the curve $x = aI^b t^{1+b}$, the depth is constant and equal to $h_\infty(x)$. (b) Flow depth variation at $x = 0$. Computation for arbitrary values $a = 1$ 1/s $b = 1$, $I = 1$ m/s, and $x_0 = 3$ m.

4.5.2 Numerical implementation

```python
import numpy as np
import matplotlib.pyplot as plt
import os
from clawpack import riemann
plt.ioff()

#!/usr/bin/env python
# encoding: utf-8

r"""
Burgers' equation
"""
def source_term(solver, state, dt):
    i  = state.problem_data['i']
    h  = state.q[0, :]
    # Update to momentum
    state.q[0, :] +=   dt * i

def inlet_bc(state,dim,t,qbc,auxbc,num_ghost):
    "inlet boundary conditions"
    qbc[0, :num_ghost] = 0.

def b4step(solver,state):
    h  = state.q[0,:]
    t  = state.t
    hf = h[-1]
    front.append([t,hf])


def setup(use_petsc=0,kernel_language='Fortran',outdir='./_output',
   solver_type='classic'):

    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if kernel_language == 'Python':
        riemann_solver = riemann.burgers_1D_py.burgers_1D
    elif kernel_language == 'Fortran':
        riemann_solver = riemann.burgers_1D

    if solver_type=='sharpclaw':
        solver = pyclaw.SharpClawSolver1D(riemann_solver)
    else:
        solver = pyclaw.ClawSolver1D(riemann_solver)
        solver.limiters = pyclaw.limiters.tvd.vanleer
    solver.kernel_language = kernel_language

    solver.bc_lower[0] = pyclaw.BC.custom
    solver.user_bc_lower = inlet_bc
    solver.bc_upper[0] = pyclaw.BC.extrap
    solver.step_source = source_term
    solver.before_step = b4step

    x = pyclaw.Dimension(0.0,10.0,1000,name='x')
    domain = pyclaw.Domain(x)
    num_eqn = 1
    state = pyclaw.State(domain,num_eqn)
    xc = state.grid.x.centers
    state.q[0,:] = 0.
    state.problem_data['efix']=True
    state.problem_data['i'] = 1

    claw = pyclaw.Controller()
    claw.tfinal = 10
    claw.num_output_times = 20
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver
    claw.outdir = outdir
    claw.setplot = setplot
    claw.keep_copy = True

    return claw

def setplot(plotdata):
    """
    Plot solution using VisClaw.
    """
    plotdata.clearfigures()  # clear any old figures,axes,items data
    # Figure for q[0]
    plotfigure = plotdata.new_plotfigure(name='q[0]', figno=0)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.xlimits = 'auto'
    plotaxes.ylimits = [-1., 2.]
    plotaxes.title = 'q[0]'
    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='1d')
    plotitem.plot_var = 0
    plotitem.plotstyle = '-o'
    plotitem.color = 'b'

    return plotdata
```

```
front = []
claw = setup()
claw.run()

ind=5
ind2=20
delta_t=claw.tfinal/claw.num_output_times

fig = plt.figure(figsize=(8,4))
left, bottom, width, heigth = 0.2, 0.2, 0.8, 0.8
ax = fig.add_axes((left ,bottom, width, heigth ))
ax.ylimits = [0,0.1]
frame = claw.frames[ind]
h = frame.q[0,:]
frame = claw.frames[ind2]
h2 = frame.q[0,:]

x = frame.state.grid.x.centers
ax.plot(x,h,label='t =  {:.2f}'.format(ind*delta_t))
ax.plot(x,h2, 'k-.',label='t  {:.2f}'.format(ind2*delta_t))

ax.set_xlabel(r'$x$ (m)')
ax.set_ylabel(r'$y$ (m)')
ax.legend()
plt.show()
```

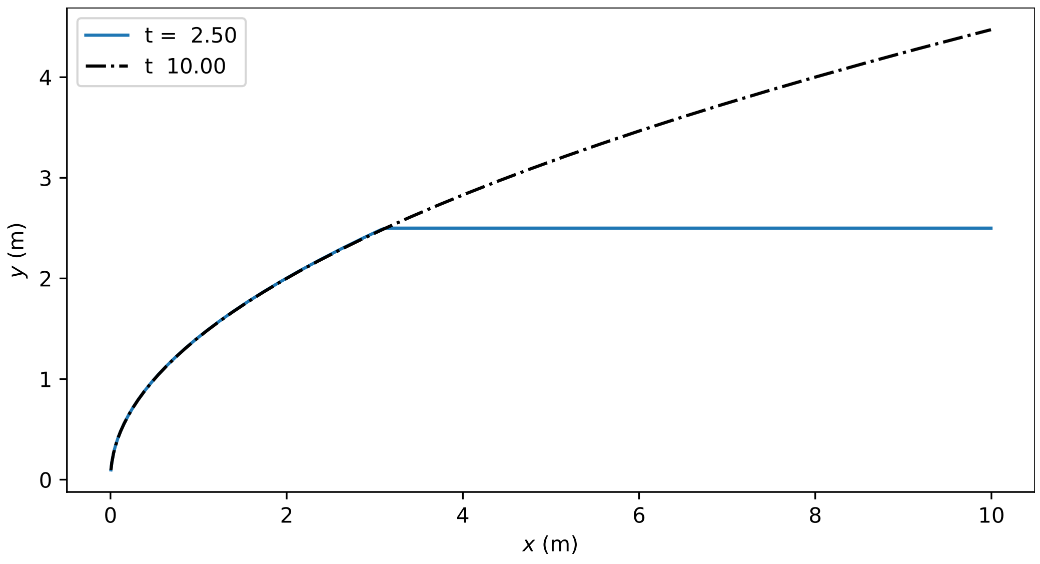


Figure 4.6 Numerical solutions at time $t = 2.5$ s and $t = 10$ s.

CHAPTER 5 Shallow water equations

5.1 Theory

The *shallow water* equations (also called the Saint-Venant) equations consist of the depth-averaged mass and momentum balance equations for a water flow along a sloping bed (Saint-Venant, 1871). In this chapter, we consider the simplest case, in which the bottom is horizonal and exerts no resistance, and the flow is one-directional. In this case, the conservative form of the governing equations comprises the mass balance equation:

$$\frac{\partial h}{\partial t} + \frac{\partial q}{\partial x} = 0, \tag{5.1}$$

the momentum balance equation

$$\frac{\partial q}{\partial t} + \frac{\partial h u^2}{\partial x} + gh\frac{\partial h}{\partial x} = 0, \tag{5.2}$$

where g is gravitational acceleration, h denotes the flow depth, $q = hu$ is the flow rate, and u the depth-averaged velocity. The unknowns are q and h. In a matrix form, Eqs. (5.1)-(5.2) takes the form:

$$\frac{\partial}{\partial t}\boldsymbol{Q} + \frac{\partial}{\partial x}\boldsymbol{f}(\boldsymbol{Q}) = 0, \tag{5.3}$$

where the flux function is:

$$\boldsymbol{f} = \left(\begin{array}{c} q \\ q^2/h + gh^2/2 \end{array}\right) \text{ and } \boldsymbol{Q} = \left(\begin{array}{c} h \\ q \end{array}\right). \tag{5.4}$$

The Jacobian is

$$\boldsymbol{f}' = \left(\begin{array}{cc} 0 & 1 \\ -q^2/h^2 + gh & 2q/h \end{array}\right), \tag{5.5}$$

whose eigenvalues are

$$\lambda_1 = u - \sqrt{gh} \text{ and } \lambda_2 = u + \sqrt{gh}, \tag{5.6}$$

associated with the right eigenvectors:

$$\boldsymbol{w}_1 = \left(\begin{array}{c} 1 \\ u - \sqrt{gh} \end{array}\right) \text{ and } \boldsymbol{w}_2 = \left(\begin{array}{c} 1 \\ u + \sqrt{gh} \end{array}\right). \tag{5.7}$$

5.1.1 *Dam break solution*

Let us consider the dam break problem on a wet bed (that is, the initial flow depth is nonzero everywhere):

$$h(x,\ 0) = \left\{ \begin{array}{l} h_l \text{ for } x < 0, \\ h_r \text{ for } x > 0, \end{array}\right. \tag{5.8}$$

and $u(x,\ 0) = 0$ everywhere, and we assume that $h_l > h_l$. The solution's structure is shown by Fig. 5.1. There is an intermediate state $\boldsymbol{Q} = (h_*,\ q_*)$ separated from the left initial state $\boldsymbol{Q}_l$ by a rarefaction wave, and from the right initial state $\boldsymbol{Q}_r$ by a shock wave.

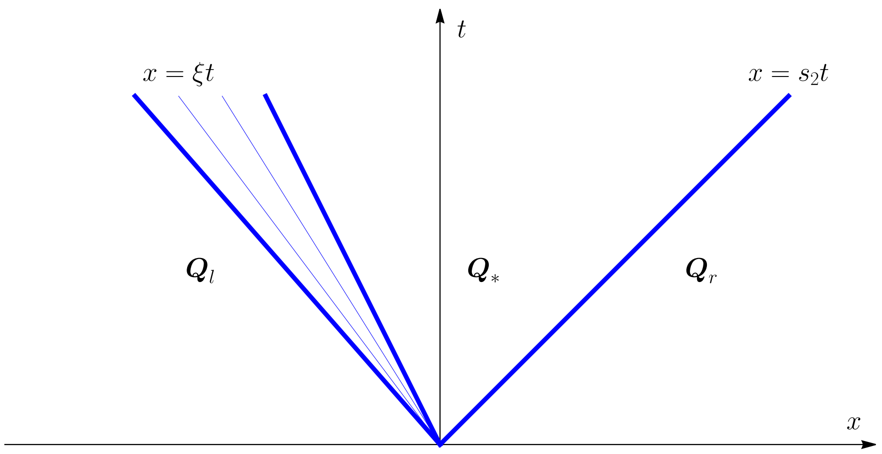


Figure 5.1 Dambreak with an intermediate state separated from the left initial state by a rarefaction wave, and from the right initial state by a shock wave.

The intermediate state satisfies the Rankine–Hugoniot condition:

$$s(\boldsymbol{Q}_* - \boldsymbol{Q}_r) = \boldsymbol{f}(\boldsymbol{Q}_*) - \boldsymbol{f}(\boldsymbol{Q}_r), \tag{5.9}$$

which implies that the shock speed is

$$s = \frac{q_* - q_r}{h_* - h_r} = u_* \mp \sqrt{g h_r \frac{h_r + h_*}{2h_*}},$$

By eliminating the shock speed in Eq. (5.9), we find that the flow rate q_* depends on the initial rate q_r (which is 0 in the example here, but we will keep it here to outline the general case) and depth h_r:

$$q_* = q_r + (h_* - h_r)\left(u_r \pm \sqrt{g h_r \left(1 + \frac{h_* - h_r}{h_r}\right)\left(1 + \frac{h_* - h_r}{2h_r}\right)}\right), \tag{5.10}$$

or in terms of the velocity u_*:

$$u_* = u_r \pm (h_* - h_r)\sqrt{\frac{g}{2}\left(\frac{1}{h_r} + \frac{1}{h_*}\right)}. \tag{5.11}$$

The intermediate state is also connected to the left state by the rarefaction wave. Let us remember that a rarefaction wave is a similarity solution $\boldsymbol{Q}(\xi)$ with $\xi = x/t$ (see § 1.4.4). Substituting this form into the hyperbolic system (5.3) gives:

$$-\xi \boldsymbol{Q}'(\xi) + \boldsymbol{f}'(\boldsymbol{Q}) \cdot \boldsymbol{Q}'(\xi) = 0,$$

which shows that $\boldsymbol{Q}'(\xi)$ is a right eigenvector of the Jacobian matrix $\boldsymbol{f}'$, and thus there exists a scalar coefficient $\alpha(\xi)$ such that

$$\boldsymbol{Q}'(\xi) = \alpha(\xi)\boldsymbol{w}_k,$$

with $k = 1, 2$. Let us assume that $\alpha = 1$ and take $k = 1$ (that is, we are looking for the 1-rarefaction wave). We have to solve:

$$\boldsymbol{Q}'(\xi) = \begin{pmatrix} h' \\ q' \end{pmatrix} = \begin{pmatrix} 1 \\ u - \sqrt{gh} \end{pmatrix}.$$

By setting the first constant of integration to zero, we find:

$$h(\xi) = \xi,$$

and

$$q' = \frac{q}{\xi} - \sqrt{g\xi} \Rightarrow q(\xi) = a\xi - 2\xi\sqrt{g\xi},$$

where a is a constant of integration. As we have $h = \xi$, this means that we also have $q(h) = ah - 2h\sqrt{gh}$. We impose that the intermediate state lies on the rarefaction wave, and thus

$$q_* = ah_* - 2h_*\sqrt{gh_*} \Rightarrow a = u_* + 2h_*\sqrt{gh_*}.$$

The 1-rarefaction is thus the curve

$$q(h) = hu_* + 2h(\sqrt{gh_*} - \sqrt{gh}). \tag{5.12}$$

In the (u, h) coordinate system, the 1-rarefaction wave satisfies:

$$u + 2\sqrt{gh} = u_* + 2\sqrt{gh_*}. \tag{5.13}$$

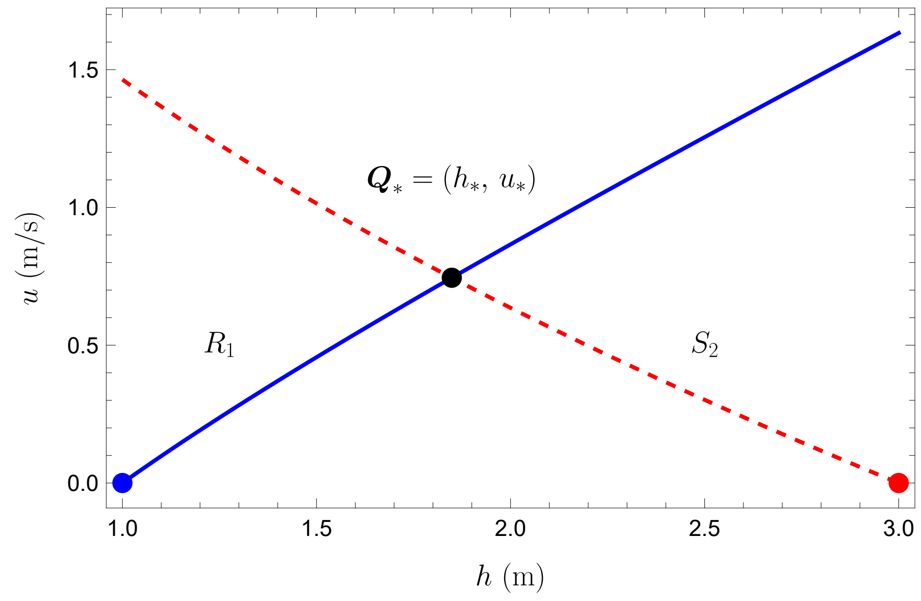


Figure 5.2 Phase plane for $h_l = 3$ and $h_r = 1$ (with $g = 1$ m/s^2).

5.2 Approximate solver: the Roe solver

5.2.1 *Derivation*

The Roe solver involves linearising the Jacobian § 2.5:

$$\hat{\boldsymbol{A}}_{i-1/2} = \int_0^1 \frac{\mathrm{d}\boldsymbol{f}(\boldsymbol{q}(\xi))}{\mathrm{d}\boldsymbol{q}}\mathrm{d}\xi, \tag{5.14}$$

where the Jacobian $\nabla \boldsymbol{f}$ has been defined by Eq. (5.5) and the integral path is the straight line:

$$\boldsymbol{q} = \boldsymbol{Q}_{i-1} + \xi(\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1}).$$

It has been shown in § 2.5 that the matrix $\hat{\boldsymbol{A}}_{i-1/2}$ satisfies Eq. (2.50)

$$\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1}) = \hat{\boldsymbol{A}}_{i-1/2} \cdot (\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1}),$$

when $\boldsymbol{Q}_{i-1}$ and $\boldsymbol{Q}_i$ are connected by the same wave (shock or rarefaction). This property is essential to ensure that the method is conservative.

The problem is that integrating Eq. (5.14) is made intricate by the coupling between variables q and h in the entries of $\nabla \boldsymbol{f}$. Roe (1981) found a change of variable $\boldsymbol{z} = \boldsymbol{z}(\boldsymbol{q})$, which allows him to solve this issue for the Euler equations. A similar trick can be applied to the shallow water equations. We make the following change of variable

$$\begin{aligned} \boldsymbol{z} &= \begin{pmatrix} z_1 \\ z_2 \end{pmatrix}, \\ &= \frac{\boldsymbol{q}}{\sqrt{h}} = \begin{pmatrix} \sqrt{h} \\ u\sqrt{h} \end{pmatrix}. \end{aligned} \tag{5.15}$$

The inverse mapping is:

$$\boldsymbol{q} = \begin{pmatrix} z_1^2 \\ z_1 z_2 \end{pmatrix} \Rightarrow \frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}\boldsymbol{z}} = \begin{pmatrix} 2z_1 & 0 \\ z_2 & z_1 \end{pmatrix}. \tag{5.16}$$

whose determinant is $2z_1^2 > 0$ wherever the flow depth is nonzero (as a consequence, $\boldsymbol{z}(\boldsymbol{q})$ is invertible everywhere). With the new variables, the flux function and its Jacobian become

$$\boldsymbol{f} = \begin{pmatrix} z_1 z_2 \\ z_2^2 + \frac{1}{2} g z_1^4 \end{pmatrix} \Rightarrow \frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{z}} = \begin{pmatrix} z_2 & z_1 \\ 2g z_1^3 & 2z_2 \end{pmatrix}.$$

Using the change of variable, we now integrate

$$\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1}) = \int_0^1 \frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\xi}(\boldsymbol{z})\mathrm{d}\xi$$

along the straight line

$$\boldsymbol{z} = \boldsymbol{Z}_{i-1} + (\boldsymbol{Z}_i - \boldsymbol{Z}_{i-1})\xi.$$

As the ξ derivative of $\boldsymbol{z}$ gives $\boldsymbol{z}' = \boldsymbol{Z}_i - \boldsymbol{Z}_{i-1}$, we obtain

$$\begin{aligned} \boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1}) &= \int_0^1 \frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{z}}(\boldsymbol{z}) \cdot \frac{\mathrm{d}\boldsymbol{z}}{\mathrm{d}\xi}\mathrm{d}\xi, \\ &= \int_0^1 \frac{\mathrm{d}\boldsymbol{f}}{\mathrm{d}\boldsymbol{z}}(\boldsymbol{z})\mathrm{d}\xi \cdot (\boldsymbol{Z}_i - \boldsymbol{Z}_{i-1}), \\ &= \begin{pmatrix} \bar{Z}^2 & \bar{Z}^1 \\ 2g\bar{Z}^1\bar{h} & 2\bar{Z}^2 \end{pmatrix}, \cdot (\boldsymbol{Z}_i - \boldsymbol{Z}_{i-1}) \end{aligned} \tag{5.17}$$

where the kth average component $\bar{Z}^k$ is defined as:

$$\bar{Z}^k = \frac{1}{2}(Z_{i-1}^k + Z_i^k) \text{ and } \bar{h} = \frac{1}{2}(h_{i-1} + h_i).$$

Proof. Integration z^k does not pose problems (where $k = 1$ or 2):

$$\begin{aligned}\int_0^1 z^k \mathrm{d}\xi &= \int_0^1 (Z_{i-1}^k + \xi(Z_i^k - Z_{i-1}^k))\mathrm{d}\xi,\\ &= \left[\xi Z_{i-1}^k + \frac{\xi}{2}(Z_i^k - Z_{i-1}^k\right]_0^1,\\ &= \frac{\xi}{2}(Z_i^k + Z_{i-1}^k),\\ &= \bar{Z}^k.\end{aligned} \tag{5.18}$$

The cubic term $(z^1)^3$ requires more work:

$$\begin{aligned}\int_0^1 (z^1)^3 \mathrm{d}\xi &= \int_0^1 (Z_{i-1}^k + \xi(Z_i^k - Z_{i-1}^k))^3\mathrm{d}\xi,\\ &= \frac{1}{Z_i^k - Z_{i-1}^k}\int_{Z_{i-1}^k}^{Z_i^k} \eta^3 \mathrm{d}\eta,\\ &= \frac{1}{4(Z_i^k - Z_{i-1}^k)}\left[\eta^4\right]_{Z_{i-1}^k}^{Z_i^k},\\ &= \frac{1}{4}\frac{(Z_i^k)^4 - (Z_{i-1}^k)^4}{Z_i^k - Z_{i-1}^k},\\ &= \frac{(Z_i^k)^2 + (Z_{i-1}^k)^2}{2}\frac{Z_i^k + Z_{i-1}^k}{2},\\ &= \bar{Z}^1\bar{h},\end{aligned}$$

since $(Z_i^k)^2 + (Z_{i-1}^k)^2 = h_i + h_{i-1}$. □

We now have Eq. (5.17), which relates the flux difference $\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1})$ to $\boldsymbol{Z}_i - \boldsymbol{Z}_{i-1}$. We need to express $\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1}$ as a function of $\boldsymbol{Z}_i - \boldsymbol{Z}_{i-1}$ so that we can directly relate the flux difference to the difference $\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1}$. To that end, we will use the following integral path $\boldsymbol{q} = \boldsymbol{Q}_{i-1} + \xi(\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1})$ and integrate d$\boldsymbol{q}$ along it:

$$\begin{aligned}\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1} &= \int_0^1 \frac{\mathrm{d}\boldsymbol{q}(\xi)}{\mathrm{d}\xi}\mathrm{d}\xi = \int_0^1 \frac{\mathrm{d}\boldsymbol{q}}{\mathrm{d}\boldsymbol{z}}\cdot\frac{\mathrm{d}\boldsymbol{z}}{\mathrm{d}\xi}\mathrm{d}\xi\\ &= \begin{pmatrix} 2\bar{Z}_1 & 0\\ \bar{Z}_2 & \bar{Z}_1\end{pmatrix}\cdot(\boldsymbol{Z}_i - \boldsymbol{Z}_{i-1}),\end{aligned}$$

where we made use of the definition (5.16) of the Jacobian $\nabla\boldsymbol{q}(\boldsymbol{z})$ together with the relation (5.18). We eventually find:

$$\hat{\boldsymbol{A}}_{i-1/2} = \begin{pmatrix} \bar{Z}_2 & \bar{Z}_1\\ 2g\bar{Z}_1\bar{h} & 2\bar{Z}_2\end{pmatrix}\cdot\begin{pmatrix} 2\bar{Z}_1 & 0\\ \bar{Z}_2 & \bar{Z}_1\end{pmatrix}^{-1} = \begin{pmatrix} 0 & 1\\ g\bar{h} - (\bar{Z}_2/\bar{Z}_1)^2 & 2\bar{Z}_2/\bar{Z}_1\end{pmatrix},$$

and after returning to the original variables

$$\hat{\boldsymbol{A}}_{i-1/2} = \begin{pmatrix} 1 & 1\\ -\hat{u} + g\bar{h} & 2\hat{u}\end{pmatrix} \text{ where } \hat{u} = \frac{\sqrt{h_{i-1}}u_{i-1} + \sqrt{h_i}u_i}{\sqrt{h_{i-1}} + \sqrt{h_i}}. \tag{5.19}$$

The Roe matrix is thus the Jacobian matrix $\boldsymbol{f}'(\boldsymbol{q})$ evaluated at the intermediate state $\boldsymbol{q} = (\bar{h},\ \bar{h}\hat{u})$. It has the eigenvalues

$$\lambda_1 = \hat{u} - \sqrt{g\bar{h}} \text{ and } \lambda_2 = \hat{u} + \sqrt{g\bar{h}},$$

associated with the right eigenvectors:

$$\boldsymbol{w}_1 = \begin{pmatrix} 1 \\ \hat{u} - \sqrt{g\bar{h}} \end{pmatrix} \text{ and } \boldsymbol{w}_2 = \begin{pmatrix} 1 \\ \hat{u} + \sqrt{g\bar{h}} \end{pmatrix}. \tag{5.20}$$

We can decompose the initial jump $\boldsymbol{Q}_i - \boldsymbol{Q}_{i-1}$ in the right eigenvector basis $(\boldsymbol{w}_k)_{k=1,2}$ as

$$\Delta\boldsymbol{Q} = \boldsymbol{Q}_i - \boldsymbol{Q}_{i-1} = \boldsymbol{R}\cdot\boldsymbol{\alpha}, \tag{5.21}$$

where $\boldsymbol{R} = [\boldsymbol{w}_1,\ \boldsymbol{w}_2]$ is the right-eigenvector matrix. We then deduce the $\boldsymbol{\alpha}$ coefficients by inverting the matrix $\boldsymbol{R}$

$$\begin{aligned}\boldsymbol{\alpha} &= \boldsymbol{R}^{-1}\cdot\Delta\boldsymbol{Q},\\ &= \frac{1}{2\hat{c}}\begin{pmatrix}\hat{u}+\hat{c} & -1\\ \hat{c}-\hat{u} & 1\end{pmatrix}\cdot\Delta\boldsymbol{Q}, && (5.22)\\ &= \frac{1}{2\hat{c}}\begin{pmatrix}(\hat{u}+\hat{c})\Delta Q_1 - \Delta Q_2\\ (-\hat{u}+\hat{c})\Delta Q_1 + \Delta Q_2\end{pmatrix}, && (5.23)\end{aligned}$$

where $\hat{c} = \sqrt{g\bar{h}}$ and $\Delta\boldsymbol{Q} = (\Delta Q_1,\ \Delta Q_2)$.

5.2.2 *Wave form*

To summarize the results, we need the following equations to write the Roe solver's algorithm:

- The velocities associated with the intermediate state

$$\hat{u} = \frac{q_{i-1}/\sqrt{h_{i-1}} + q_i/\sqrt{h_i}}{\sqrt{h_{i-1}} + \sqrt{h_i}} \text{ and } \hat{c} = \sqrt{g\bar{h}} = \sqrt{\frac{1}{2}(\sqrt{h_{i-1}} + \sqrt{h_i})}. \tag{5.24}$$

- The waves $\boldsymbol{W}_k$:

$$\boldsymbol{W}_k = \alpha_k \boldsymbol{w}_k,\ k = 1,\ 2 \tag{5.25}$$

  where α_k are the components of the $\boldsymbol{\alpha}$ vector given by Eq. (5.22) and $\boldsymbol{w}_k$ are the right eigenvectors of the Roe matrix given by Eq. (5.20).

- the characteristic speeds

$$s_1 = \hat{u} - \hat{c} \text{ and } s_2 = \hat{u} + \hat{c}. \tag{5.26}$$

- The fluctuations are

$$\boldsymbol{A}^-\cdot\Delta\boldsymbol{Q}_{i-1/2} = \sum_{k=1}^{2}\min(\lambda^k_{i-1/2}, 0)\boldsymbol{W}_{k,i-1/2},$$

$$\boldsymbol{A}^+\cdot\Delta\boldsymbol{Q}_{i+1/2} = \sum_{k=1}^{2}\max(\lambda^k_{i-1/2}, 0)\boldsymbol{W}_{k,i-1/2},$$

  which gives in the present context:

  – if $s_k < 0$, then `amdq(m,i)=s*wave`.
  – if $s_k > 0$, then `apdq(m,i)=s*wave`.

5.2.3 Sonic entropy fix

We have seen in § 2.4.2 that when the solution to the Riemann problem is a transonic wave, the Roe approximate solution may be incorrect (see also LeVeque, 2002, § 15.3.5). In that case, there is usually an intermediate state $\boldsymbol{Q}_*$ between the left and right states $\boldsymbol{Q}_{i-1}$ and $\boldsymbol{Q}_i$, and the associated speeds are

- for the first eigenvalue

$$\lambda^*_{i-1} = u_{i-1} - \sqrt{gh_{i-1}},\ \lambda^1_* = u_* - \sqrt{g\hat{h}_*}$$

- for the second eigenvalue

$$\lambda^2_* = u_* + \sqrt{g\hat{h}_*},\ \lambda^2_i = u_i + \sqrt{g\hat{h}_i}.$$

When $\lambda^1_{i-1} < 0 < \lambda^1_*$ (resp. $\lambda^2_* < 0 < \lambda^2_i$), we should consider that the 1-wave (resp. the 2-wave) is a transonic rarefaction wave rather than a 1-shock wave (see Fig. 5.3).

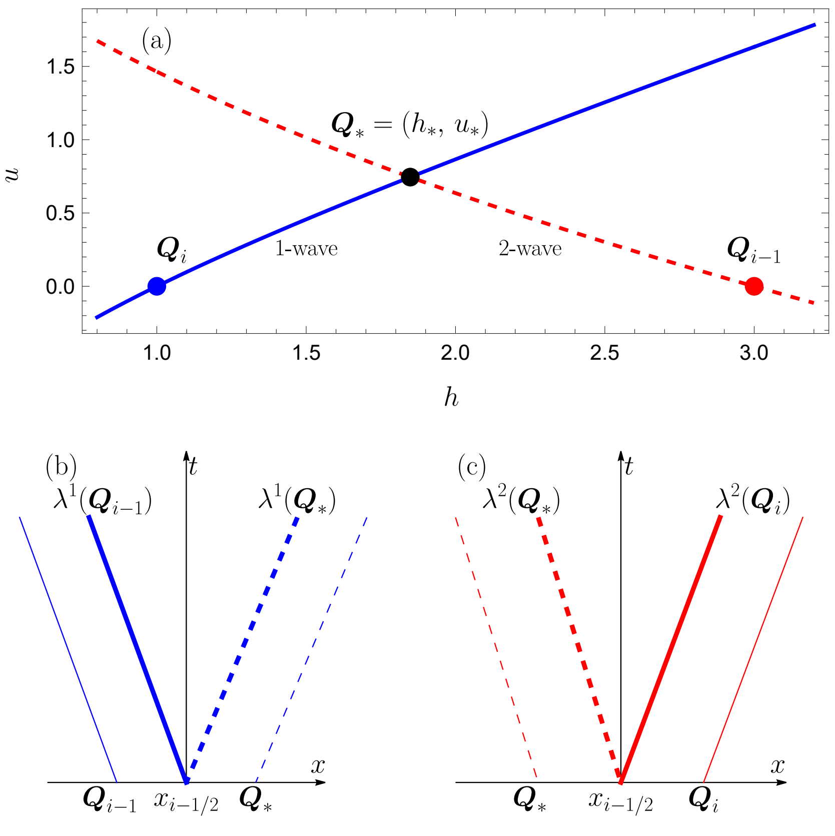


Figure 5.3 (a) In the general case, there is an intermediate state $\boldsymbol{Q}_*$ connecting the left and right states $\boldsymbol{Q}_{i-1}$ and $\boldsymbol{Q}_i$ by 1- and 2-waves of respective velocity λ^1 and λ^2. (b) If $\lambda^1_{i-1} < 0 < \lambda^1_*$, then the 1-wave is a transonic wave. (c) If $\lambda^2_* < 0 < \lambda^2_i$, then the 2-wave is a transonic wave.

By using the analytical expression (1.116) for a centred rarefaction wave (evaluated at $\xi = 0$), we can deduce the interface value

$$h_{i-1/2} = \frac{1}{9g}\left(u_{i-1} + 2\sqrt{gh_{i-1}}\right)^2, \tag{5.27}$$

while Eq. (1.115) yields:

$$u_{i-1/2} = u_{i-1} + 2\left(\sqrt{gh_{i-1}} - \sqrt{gh_{i-1/2}}\right) \tag{5.28}$$

The flux fluctuations are computed using Eqs. (2.48) and (2.49).

5.2.4 *Implementation in Clawpack*

Here is how the Roe solver is implemented in Clawpack (without and without the entropy fix to compute transonic wave).

```fortran
subroutine rp1(maxmx,num_eqn,num_waves,num_aux,num_ghost,num_cells, &
               ql,qr,auxl,auxr,wave,s,amdq,apdq)

! waves: 2
! equations: 2

    implicit none
    integer, intent(in) :: maxmx, num_eqn, num_waves, num_aux, num_ghost, &
                           num_cells
    real(kind=8), intent(in), dimension(num_eqn,1-num_ghost:maxmx+num_ghost
   ) :: ql, qr
    real(kind=8), intent(in), dimension(num_aux,1-num_ghost:maxmx+num_ghost
   ) :: auxl, auxr
    real(kind=8), intent(out) :: s(num_waves, 1-num_ghost:maxmx+num_ghost)
    real(kind=8), intent(out) :: wave(num_eqn, num_waves, 1-num_ghost:maxmx
   +num_ghost)
    real(kind=8), intent(out), dimension(num_eqn,1-num_ghost:maxmx+
   num_ghost) :: amdq,apdq

    ! local variables:
    real(kind=8) :: a1,a2,ubar,cbar,s0,s1,s2,s3,hr1,uhr1,hl2,uhl2,sfract,df
    real(kind=8) :: delta(2)
    integer :: i,m,mw

    logical :: efix

    data efix /.true./    !# Use entropy fix for transonic rarefactions

    ! Gravity constant set in setprob.f or the shallow1D.py file
    real(kind=8) :: grav
    common /cparam/ grav

    ! Main loop of the Riemann solver.
    do 30 i=2-num_ghost,num_cells+num_ghost


        ! compute  Roe-averaged quantities:
        ubar = (qr(2,i-1)/dsqrt(qr(1,i-1)) + ql(2,i)/dsqrt(ql(1,i)))/ &
               ( dsqrt(qr(1,i-1)) + dsqrt(ql(1,i)) )
        cbar=dsqrt(0.5d0*grav*(qr(1,i-1) + ql(1,i)))

        ! delta(1)=h(i)-h(i-1) and  delta(2)=hu(i)-hu(i-1)
        delta(1) = ql(1,i) - qr(1,i-1)
        delta(2) = ql(2,i) - qr(2,i-1)

        ! Compute coeffs in the evector expansion of delta(1),delta(2)
        a1 = 0.5d0*(-delta(2) + (ubar + cbar) * delta(1))/cbar
        a2 = 0.5d0*( delta(2) - (ubar - cbar) * delta(1))/cbar

        ! Finally, compute the waves.
        wave(1,1,i) = a1
        wave(2,1,i) = a1*(ubar - cbar)
        s(1,i) = ubar - cbar

        wave(1,2,i) = a2
        wave(2,2,i) = a2*(ubar + cbar)
        s(2,i) = ubar + cbar

    30 enddo

    ! Compute fluctuations amdq and apdq
    ! ------------------------------------

    if (efix) go to 110

    ! No entropy fix
    ! ----------------------------------------------
    ! amdq = SUM s*wave   over left-going waves
    ! apdq = SUM s*wave   over right-going waves

    do m=1,2
        do i=2-num_ghost, num_cells+num_ghost
            amdq(m,i) = 0.d0
            apdq(m,i) = 0.d0
            do mw=1,num_waves
                if (s(mw,i) < 0.d0) then
                    amdq(m,i) = amdq(m,i) + s(mw,i)*wave(m,mw,i)
                else
                    apdq(m,i) = apdq(m,i) + s(mw,i)*wave(m,mw,i)
                endif
            enddo
        enddo
    enddo

    ! with no entropy fix we are done...
    return


    ! -----------------------------------------------

    110 continue

    ! With entropy fix
    ! ------------------

    ! compute flux differences amdq and apdq.
    ! First compute amdq as sum of s*wave for left going waves.
    ! Incorporate entropy fix by adding a modified fraction of wave
    ! if s should change sign.

    do 200 i=2-num_ghost,num_cells+num_ghost

        ! ------------------------------------------------------
        ! check 1-wave:
        ! ---------------

        ! u-c in left state (cell i-1)
        s0 = qr(2,i-1)/qr(1,i-1) - dsqrt(grav*qr(1,i-1))

        ! check for fully supersonic case:
        if (s0 >= 0.d0 .and. s(1,i) > 0.d0)  then
            ! everything is right-going
            do m=1,2
                amdq(m,i) = 0.d0
                enddo
            go to 200
        endif

        ! u-c to right of 1-wave
        hr1  = qr(1,i-1) + wave(1,1,i)
        uhr1 = qr(2,i-1) + wave(2,1,i)
        s1 =  uhr1/hr1 - dsqrt(grav*hr1)

        if (s0 < 0.d0 .and. s1 > 0.d0) then
            ! transonic rarefaction in the 1-wave
            sfract = s0 * (s1-s(1,i)) / (s1-s0)
        else if (s(1,i) < 0.d0) then
            ! 1-wave is leftgoing
            sfract = s(1,i)
        else
            ! 1-wave is rightgoing
            sfract = 0.d0   !# this shouldn't happen since s0 < 0
        endif

        do m=1,2
            amdq(m,i) = sfract*wave(m,1,i)
            enddo

        ! -------------------------------------------------------
        ! check 2-wave:
        ! ---------------
        ! u+c in right state  (cell i)
        s3 = ql(2,i)/ql(1,i) + dsqrt(grav*ql(1,i))

        ! u+c to left of 2-wave
        hl2  = ql(1,i) - wave(1,2,i)
        uhl2 = ql(2,i) - wave(2,2,i)
        s2 = uhl2/hl2 + dsqrt(grav*hl2)

        if (s2 < 0.d0 .and. s3 > 0.d0) then
            ! transonic rarefaction in the 2-wave
            sfract = s2 * (s3-s(2,i)) / (s3-s2)
        else if (s(2,i) < 0.d0) then
            ! 2-wave is leftgoing
            sfract = s(2,i)
        else
            ! 2-wave is rightgoing
            go to 200
        endif

        do m=1,2
            amdq(m,i) = amdq(m,i) + sfract*wave(m,2,i)
            enddo
```

```

    200 enddo

    ! compute the rightgoing flux differences:
    ! df = SUM s*wave   is the total flux difference and apdq = df - amdq

    do m=1,2
        do i = 2-num_ghost, num_cells+num_ghost
            df = 0.d0
            do mw=1,num_waves
                df = df + s(mw,i)*wave(m,mw,i)
                enddo
            apdq(m,i) = df - amdq(m,i)
            enddo
        enddo

    return

    end subroutine rp1
```

5.2.5 Implementation in Pyclaw

Here is how the Roe solver is implemented in Pyclaw

```python
def shallow_roe_1D(q_l, q_r, aux_l, aux_r, problem_data):
    r"""
    Roe shallow water solver in 1d::
    """
    # Array shapes
    num_rp = q_l.shape[1]

    # Output arrays
    wave = np.empty( (num_eqn, num_waves, num_rp) )
    s = np.zeros( (num_waves, num_rp) )
    amdq = np.zeros( (num_eqn, num_rp) )
    apdq = np.zeros( (num_eqn, num_rp) )

    # Compute roe-averaged quantities
    ubar = ( (q_l[1,:]/np.sqrt(q_l[0,:]) + q_r[1,:]/np.sqrt(q_r[0,:])) /
             (np.sqrt(q_l[0,:]) + np.sqrt(q_r[0,:])) )
    cbar = np.sqrt(0.5 * problem_data['grav'] * (q_l[0,:] + q_r[0,:]))

    # Compute Flux structure
    delta = q_r - q_l
    a1 = 0.5 * (-delta[1,:] + (ubar + cbar) * delta[0,:]) / cbar
    a2 = 0.5 * ( delta[1,:] - (ubar - cbar) * delta[0,:]) / cbar

    # Compute each family of waves
    wave[0,0,:] = a1
    wave[1,0,:] = a1 * (ubar - cbar)
    s[0,:] = ubar - cbar

    wave[0,1,:] = a2
    wave[1,1,:] = a2 * (ubar + cbar)
    s[1,:] = ubar + cbar

    s_index = np.zeros((2,num_rp))
    for m in range(num_eqn):
        for mw in range(num_waves):
            s_index[0,:] = s[mw,:]
            amdq[m,:] += np.min(s_index,axis=0) * wave[m,mw,:]
            apdq[m,:] += np.max(s_index,axis=0) * wave[m,mw,:]

    return wave, s, amdq, apdq
```

5.3 HLLE solver

5.3.1 *Principle*

The HLL method is a two-wave solver that considers that the solution to the Riemann problem consists of two shock waves separating the intermediate state $\boldsymbol{Q}_*$ $\boldsymbol{Q}_i$ from the left and right initial states $\boldsymbol{Q}_{i-1}$ and $\boldsymbol{Q}_i$. In § 2.6, we have seen that this intermediate state and the associated flux can be determined by solving the Rankine–Hugoniot equations for the discontinuities across the two shock waves. Here we provide another proof based on volume integrals.

Integrating the shallow water equations (5.3) over the domain $[x_1,\ x_2] \times [0, \Delta t]$ in the $x - t$ plane (see Fig. 2.5) gives

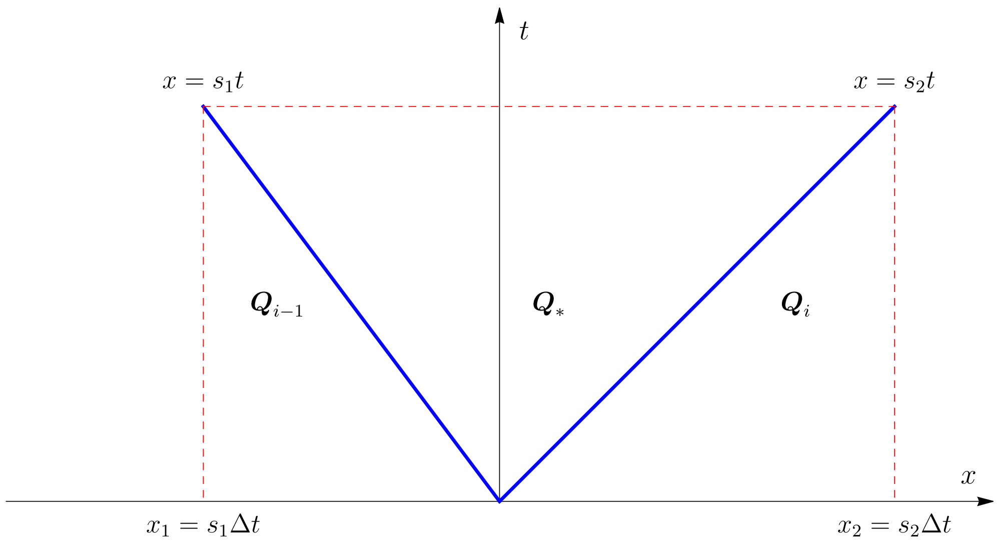
$$\int_{x_1}^{x_2} \boldsymbol{q}(x,\ \Delta t)\mathrm{d}x = \int_{x_1}^{x_2} \boldsymbol{q}(x,\ 0)\mathrm{d}x + \int_0^{\Delta t} \boldsymbol{f}(\boldsymbol{q}(x_1,\ t))\mathrm{d}t - \int_0^{\Delta t} \boldsymbol{f}(\boldsymbol{q}(x_2,\ t))\mathrm{d}t,$$

where $x_1 = s_1\Delta t$ and $x_2 = s_2\Delta t$. Since $\boldsymbol{q}(x,\ 0)$ is fixed by the initial conditions, we deduce

$$\boldsymbol{Q}_* = \frac{1}{x_2 - x_1}\int_{x_1}^{x_2} \boldsymbol{q}(x,\ \Delta t)\mathrm{d}x = \frac{\boldsymbol{Q}_i x_2 - \boldsymbol{Q}_{i-1}x_1}{x_2 - x_1} - \frac{\Delta t}{x_2 - x_1}(\boldsymbol{F}_i - \boldsymbol{F}_{i-1}),$$

where

$$\boldsymbol{F}_{i-1} = \frac{1}{\Delta t}\int_0^{\Delta t} \boldsymbol{f}(\boldsymbol{q}(x_1,\ t))\mathrm{d}t \text{ and } \boldsymbol{F}_i = \frac{1}{\Delta t}\int_0^{\Delta t} \boldsymbol{f}(\boldsymbol{Q}(x_2,\ t))\mathrm{d}t.$$



We find that the intermediate state is:

$$\boldsymbol{Q}_* = \frac{\boldsymbol{Q}_i s_2 - \boldsymbol{Q}_{i-1} s_1}{s_2 - s_1} - \frac{\boldsymbol{F}_i - \boldsymbol{F}_{i-1}}{s_2 - s_1}. \tag{5.29}$$

In § 2.3.3, we derived the general expression (2.34) for computing the interface flux from the left and right flux

$$\boldsymbol{F}_{i-1/2} = \boldsymbol{f}(\boldsymbol{Q}_{i-1}) + \frac{\delta x}{\delta t}\boldsymbol{Q}_{i-1} - \frac{1}{\delta t}\int_{-\delta x}^{0} \boldsymbol{Q}(x,\ \delta t)\mathrm{d}x.$$

which gives us when we take $\delta x = -x_2 = -s_2\Delta t$:

$$\boldsymbol{F}_* = \frac{s_2 \boldsymbol{F}_{i-1} - s_1 \boldsymbol{F}_i}{s_2 - s_1} - s_2 s_2 \frac{\boldsymbol{Q}_{i-1} - \boldsymbol{Q}_i}{s_2 - s_1}. \tag{5.30}$$

This expression of the flux holds when the two shock waves fan out on either side of $x = 0$. In that case, the interface flux is defined as $\boldsymbol{F}_{i-1/2} = \boldsymbol{F}_*$. If both shock waves go to the right (i.e., if $s_1 > 0$) then $\boldsymbol{F}_{i-1/2} = \boldsymbol{F}_{i-1}$. In the opposite case, then $\boldsymbol{F}_{i-1/2} = \boldsymbol{F}_i$:

$$\boldsymbol{F}_{i-1/2} = \begin{cases} \boldsymbol{F}_{i-1} \text{ if } s_1 > 0, \\ \boldsymbol{F}_* \text{ if } s_1 \geq 0 \geq s_2, \\ \boldsymbol{F}_i \text{ if } s_2 < 0, \end{cases} \tag{5.31}$$

The last problem to be settled is the determination of the shock speed s_1 and s_2. We use the suggest of Einfeldt, which explains why the solver is called HLLE. Let us first consider the 1-wave. If this wave is a rarefaction wave, its speeds ranges from $\lambda_1(\boldsymbol{Q}_{i-1})$ to $\lambda_1(\boldsymbol{Q}_i) = u_{i-1} - \sqrt{gh_{i-1}}$; we select the minimum value $\lambda_1(\boldsymbol{Q}_{i-1})$. If it is a shock, its speed can be estimated using the Roe matrix (5.19): $s_1 = \hat{u} - \hat{c}$. As we do not know whether the 1-wave is a shock or rarefaction wave, we take the lower bound:

$$s_1 = \min(u_{i-1} - \sqrt{gh_{i-1}},\ \hat{u} - \hat{c}). \tag{5.32}$$

The same applies for the 2-wave. We define s_2 as the upper bound

$$s_2 = \max(u_i + \sqrt{gh_i},\ \hat{u} + \hat{c}). \tag{5.33}$$

In short, we compute the Roe averages, and deduce the shock speeds (5.32) and (5.33). The intermediate state is given by Eq (5.29). The waves are

$$\boldsymbol{W}_1 = \boldsymbol{Q}_* - \boldsymbol{Q}_{i-1} \text{ and } \boldsymbol{W}_2 = \boldsymbol{Q}_i - \boldsymbol{Q}_*. \tag{5.34}$$

The fluctuations are then

$$\boldsymbol{A}^- \cdot \Delta\boldsymbol{Q}_{i-1/2} = s_1 \boldsymbol{W}_1$$
$$\boldsymbol{A}^+ \cdot \Delta\boldsymbol{Q}_{i-1/2} = s_1 \boldsymbol{W}_2.$$

5.3.2 *Implementation in Pyclaw*

```python
def shallow_hll_1D(q_l,q_r,aux_l,aux_r,problem_data):
    r"""
    HLL shallow water solver ::


        W_1 = Q_hat - Q_l    s_1 = min(u_l-c_l,u_l+c_l,lambda_roe_1,
    lambda_roe_2)
        W_2 = Q_r - Q_hat    s_2 = max(u_r-c_r,u_r+c_r,lambda_roe_1,
    lambda_roe_2)
```

```

        Q_hat = ( f(q_r) - f(q_l) - s_2 * q_r + s_1 * q_l ) / (s_1 - s_2)

    *problem_data* should contain:
     - *g* - (float) Gravitational constant

    :Version: 1.0 (2009-02-05)
    """
    # Array shapes
    num_rp = q_l.shape[1]
    num_eqn = 2
    num_waves = 2

    # Output arrays
    wave = np.empty( (num_eqn, num_waves, num_rp) )
    s = np.empty( (num_waves, num_rp) )
    amdq = np.zeros( (num_eqn, num_rp) )
    apdq = np.zeros( (num_eqn, num_rp) )

    # Compute Roe and right and left speeds
    ubar = ( (q_l[1,:]/np.sqrt(q_l[0,:]) + q_r[1,:]/np.sqrt(q_r[0,:])) /
        (np.sqrt(q_l[0,:]) + np.sqrt(q_r[0,:])) )
    cbar = np.sqrt(0.5 * problem_data['grav'] * (q_l[0,:] + q_r[0,:]))
    u_r = q_r[1,:] / q_r[0,:]
    c_r = np.sqrt(problem_data['grav'] * q_r[0,:])
    u_l = q_l[1,:] / q_l[0,:]
    c_l = np.sqrt(problem_data['grav'] * q_l[0,:])

    # Compute Einfeldt speeds
    s_index = np.empty((4,num_rp))
    s_index[0,:] = ubar+cbar
    s_index[1,:] = ubar-cbar
    s_index[2,:] = u_l + c_l
    s_index[3,:] = u_l - c_l
    s[0,:] = np.min(s_index,axis=0)
    s_index[2,:] = u_r + c_r
    s_index[3,:] = u_r - c_r
    s[1,:] = np.max(s_index,axis=0)

    # Compute middle state
    q_hat = np.empty((2,num_rp))
    q_hat[0,:] = ((q_r[1,:] - q_l[1,:] - s[1,:] * q_r[0,:]
                            + s[0,:] * q_l[0,:]) / (s[0,:] - s[1,:]))
    q_hat[1,:] = ((q_r[1,:]**2/q_r[0,:] + 0.5 * problem_data['grav'] * q_r
    [0,:]**2
                - (q_l[1,:]**2/q_l[0,:] + 0.5 * problem_data['grav'] * q_l
    [0,:]**2)
                - s[1,:] * q_r[1,:] + s[0,:] * q_l[1,:]) / (s[0,:] - s
    [1,:]))

    # Compute each family of waves
    wave[:,0,:] = q_hat - q_l
    wave[:,1,:] = q_r - q_hat

    # Compute variations
    s_index = np.zeros((2,num_rp))
    for m in range(num_eqn):
        for mw in range(num_waves):
            s_index[0,:] = s[mw,:]
            amdq[m,:] += np.min(s_index,axis=0) * wave[m,mw,:]
            apdq[m,:] += np.max(s_index,axis=0) * wave[m,mw,:]

    return wave, s, amdq, apdq
```

5.4 F-wave formulation

5.4.1 Principle

The f-wave method consists of decomposing the flux jump into f-waves

$$\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1}) = \sum_{k=1}^{m_w} \boldsymbol{Z}_{k,i-1/2},$$

where the f-wave $\boldsymbol{Z}_{k,i-1/2}$ can be related to the right eigenvector $\hat{\boldsymbol{w}}_{k,i-1/2}$ of the Roe matrix:

$$\boldsymbol{Z}_{k,i-1/2} = \beta_{k,i-1/2}\hat{\boldsymbol{w}}_{k,i-1/2}$$

where the coefficient $\beta_{k,i-1/2}$ is the linear solution (see § 2.7):

$$\boldsymbol{\beta}_{i-1/2} = \boldsymbol{L}\cdot(\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1})).$$

with $\boldsymbol{L} = \boldsymbol{R}^{-1}$. We find that

$$\boldsymbol{\beta}_{i-1/2} = \frac{1}{2\hat{c}}\left(\begin{array}{c} \Phi_l - \phi_r + (\hat{u}+\hat{c})(q_r - q_l) \\ \Phi_r - \phi_l - (\hat{u}-\hat{c})(q_r - q_l) \end{array}\right), \tag{5.35}$$

where Φ is the shorthand notation: $\Phi = u^2h + gh^2/2$. The f-waves are then

$$\boldsymbol{Z}_{1,i-1/2} = \boldsymbol{\beta}_{1,i-1/2}\boldsymbol{w}_1 = \frac{\Phi_l - \Phi_r + (\hat{u}+\hat{c})(q_r - q_l)}{2\hat{c}}\left(\begin{array}{c} 1 \\ \hat{u}-\hat{c} \end{array}\right) \tag{5.36}$$

and

$$\boldsymbol{Z}_{2,i-1/2} = \boldsymbol{\beta}_{2,i-1/2}\boldsymbol{w}_2 = \frac{\Phi_r - \Phi_l - (\hat{u}-\hat{c})(q_r - q_l)}{2\hat{c}}\left(\begin{array}{c} 1 \\ \hat{u}+\hat{c} \end{array}\right). \tag{5.37}$$

5.4.2 Implementation in Pyclaw

```
def shallow_water_fwave_1d(q_l, q_r, aux_l, aux_r, problem_data):
    r"""Shallow water Riemann solver using fwaves

    *problem_data* should contain:
     - *grav* - (float) Gravitational constant
     - *dry_tolerance* - (float) Set velocities to zero if h is below this
       tolerance.
    """

    g = problem_data['grav']
    dry_tolerance = problem_data['dry_tolerance']


    num_rp = q_l.shape[1]
    num_eqn = 2
    num_waves = 2

    # initializing f-waves
    fwave = np.empty( (num_eqn, num_waves, num_rp) )
    # right eigenvectors
    r1 = np.empty( (num_waves, num_rp) )
    r2 = np.empty( (num_waves, num_rp) )
    # initializing fluctuations and shock speeds
    amdq = np.zeros( (num_eqn, num_rp) )
    apdq = np.zeros( (num_eqn, num_rp) )
    s = np.empty( (num_waves, num_rp) )


    # Extract state
    hl = q_l[0, :]
    ql = q_l[1, :]
    ul = np.where(hl > dry_tolerance, ql/hl , 0.0)
    hr = q_r[0, :]
    qr = q_r[1, :]
    ur = np.where(hr > dry_tolerance, qr/hr, 0.0)

    phi_l = hl * ul**2 + 0.5 * g * hl**2
    phi_r = hr * ur**2 + 0.5 * g * hr**2
    h_bar = 0.5 * (hr + hl)

    # Speeds
    u_hat = (np.sqrt(hl) * ul + np.sqrt(hr) * ur) / (np.sqrt(hl) + np.sqrt(
   hr) )
    c_hat = np.sqrt(g * h_bar)
    lambda1 = u_hat - c_hat
    lambda2 = u_hat + c_hat


    beta1 = (phi_l - phi_r +lambda2*(qr-ql))/2/c_hat
    beta2 = (phi_r - phi_l -lambda1*(qr-ql))/2/c_hat



    r1[0, :] = 1.
    r1[1, :] = u_hat - c_hat
    r2[0, :] = 1.
    r2[1, :] = u_hat + c_hat

    s[0,:] = u_hat - c_hat
    s[1,:] = u_hat + c_hat

    # 1st f-wave
    fwave[0,0,:] = beta1*r1[0,:]
    fwave[1,0,:] = beta1*r1[1,:]
    # 2nd f-wave
    fwave[0,1,:] = beta2*r2[0,:]
    fwave[1,1,:] = beta2*r2[1,:]

    for m in range(num_eqn):
        for mw in range(num_waves):
            amdq[m, :] += (s[mw, :] < 0.0) * fwave[m, mw, :]
            apdq[m, :] += (s[mw, :] > 0.0) * fwave[m, mw, :]

            amdq[m, :] += (s[mw, :] == 0.0) * fwave[m, mw, :] * 0.5
            apdq[m, :] += (s[mw, :] == 0.0) * fwave[m, mw, :] * 0.5

    return fwave, s, amdq, apdq
```

5.5 Example: dam break

We consider a dam break problem with the following initial conditions: $h_l = 10$ m et $u_l = 0$ for $x \leq 0$, and $h_l = 0.5$ m et $u_l = 0$ for $x > 0$. We compare the three solvers: Roe (with or without the entropy fix), the HLLE solver, and the f-wave formulation. Figures 5.4 and 5.5 show the comparison.

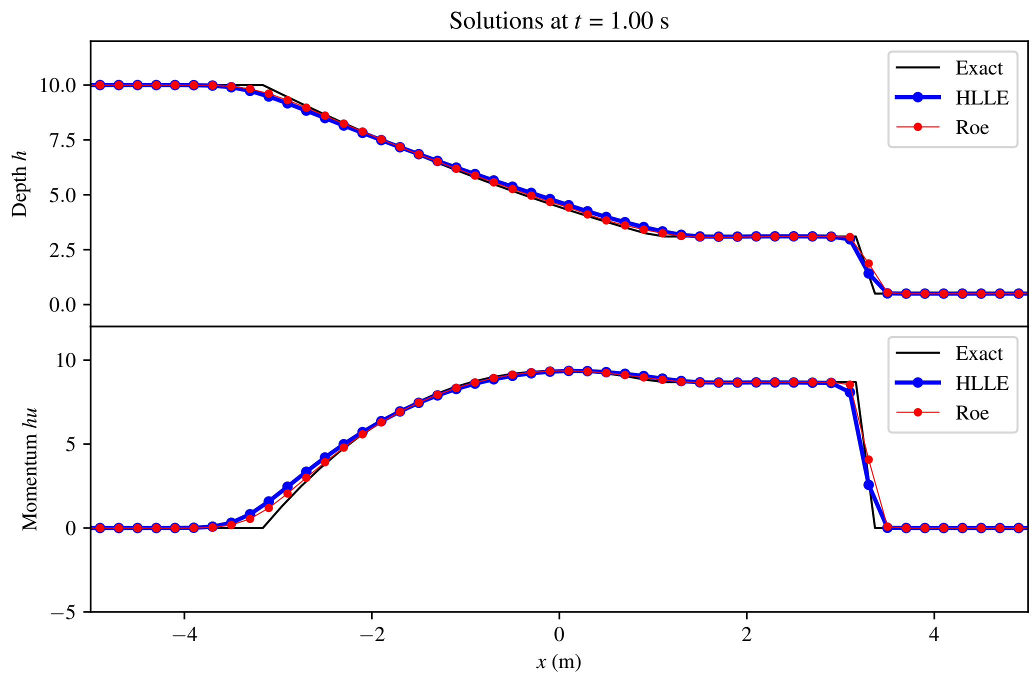


Figure 5.4 Comparison between the analytical solution, the Roe (with entropy fix) and HLLE solution.

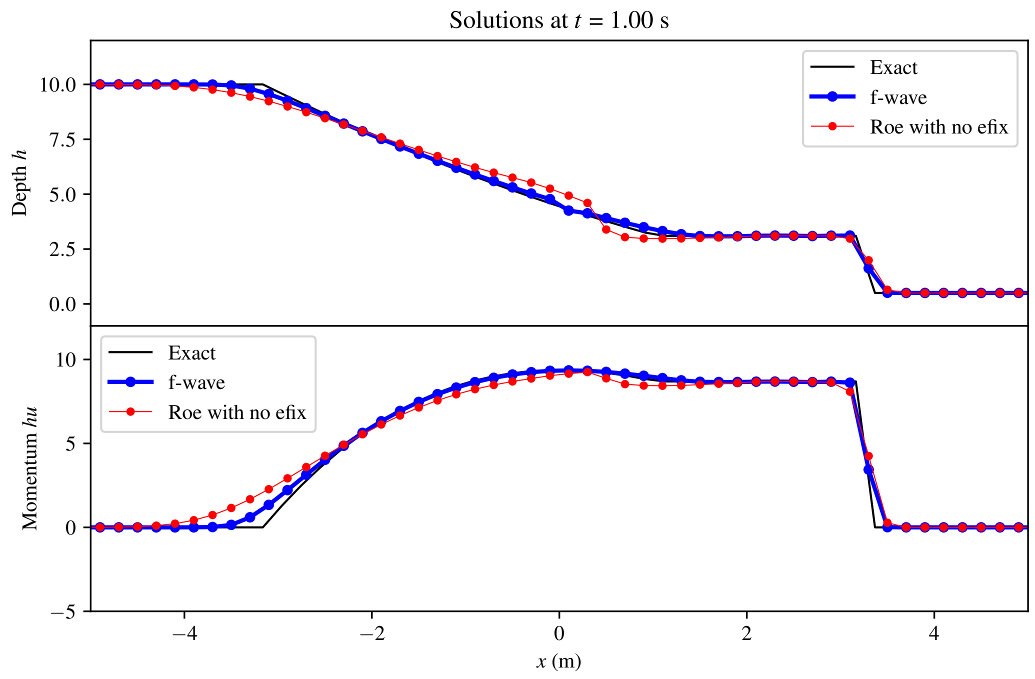
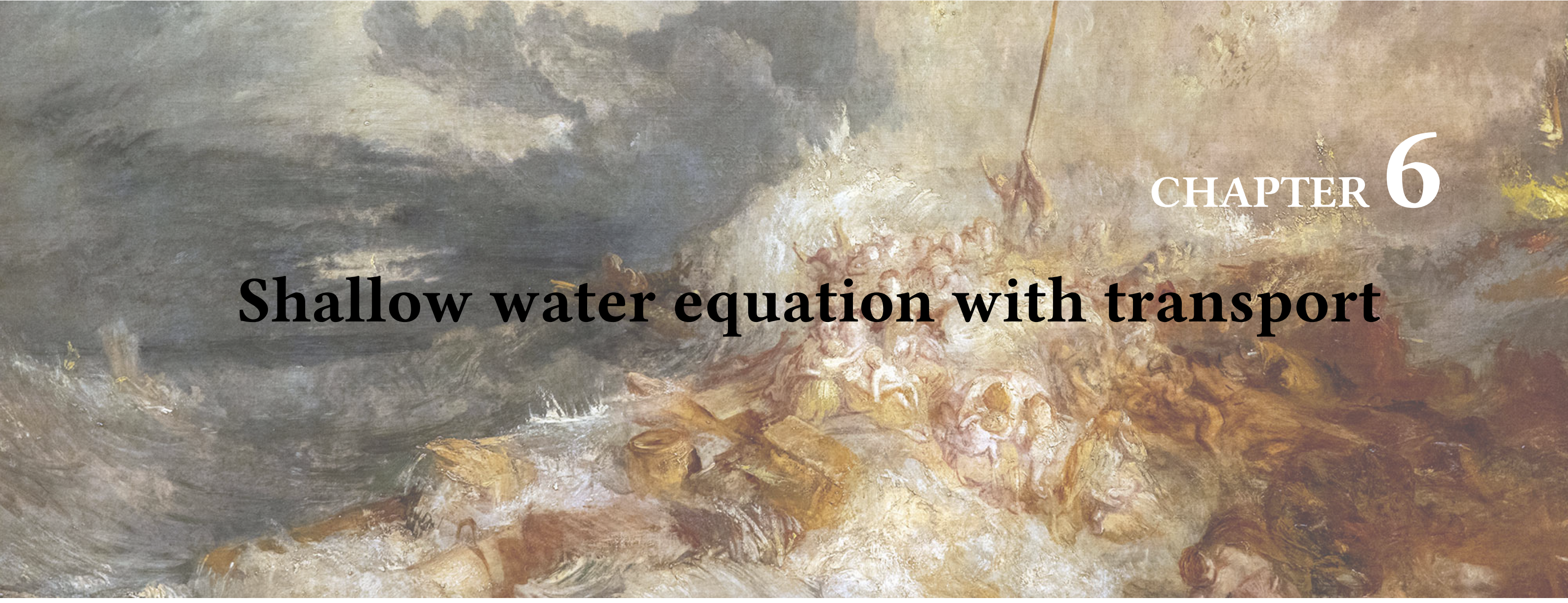


Figure 5.5 Comparison between the analytical solution, the Roe (with no entropy fix) and f-wave solution.



CHAPTER 6

Shallow water equation with transport

6.1 Theory

Let us consider the shallow water equations seen in Chap. 5, supplemented with an equation representing the advection of a scalar quantity $\phi(x,\ t)$ (for instance, the concentration of a pollutant that does not interplay with the water flow):

$$\frac{\partial h}{\partial t}+\frac{\partial q}{\partial x}=0, \tag{6.1}$$

$$\frac{\partial q}{\partial t}+\frac{\partial hu^2}{\partial x}+gh\frac{\partial h}{\partial x}=0, \tag{6.2}$$

$$\frac{\partial h\phi}{\partial t}+\frac{\partial q\phi}{\partial x}=0, \tag{6.3}$$

where h denotes the flow depth, $q=hu$ is the flow rate, and u the depth-averaged velocity. where g is gravitational acceleration, and the unknowns are q, h and ϕ. In a matrix form, Eqs. (6.1)-(6.3) takes the form:

$$\frac{\partial}{\partial t}\boldsymbol{Q}+\frac{\partial}{\partial x}\boldsymbol{f}(\boldsymbol{Q})=0, \tag{6.4}$$

where

$$\boldsymbol{f}=\begin{pmatrix} q \\ q^2/h+gh^2/2 \\ q\phi \end{pmatrix} \text{ and } \boldsymbol{Q}=\begin{pmatrix} h \\ q \\ h\phi \end{pmatrix}. \tag{6.5}$$

The Jacobian is

$$\boldsymbol{f}'=\begin{pmatrix} 0 & 1 & 0 \\ -u^2+gh & 2u & 0 \\ -u\phi & \phi & u \end{pmatrix}, \tag{6.6}$$

whose eigenvalues are

$$\lambda_1=u-\sqrt{gh}, \lambda_2=u \text{ and } \lambda_3=u+\sqrt{gh}, \tag{6.7}$$

associated with the right eigenvectors:

$$\boldsymbol{w}_1=\begin{pmatrix} 1 \\ u-\sqrt{gh} \\ \phi \end{pmatrix}, \boldsymbol{w}_2=\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \text{ and } \boldsymbol{w}_3=\begin{pmatrix} 1 \\ u+\sqrt{gh} \\ \phi \end{pmatrix}. \tag{6.8}$$

The scalar quantity ϕ is decoupled from the water flow, and its speed depends only on the water flow velocity: $\lambda_2=u$. The associated field is said to be *linearly degenerate* because $\nabla\lambda_2\cdot\boldsymbol{w}_2=0$. This gives rise to *contact discontinuities*: when ϕ experiences a shock, there is a discontinuity in u, and thus the characteristic speeds are equal on either side of the shock waves (the characteristic curves are parallel to the shock cuves). The condition $\nabla\lambda_2\cdot\boldsymbol{w}_2=0$ means that the eigenvalue is unchanged when we move along the integral curve $\boldsymbol{w}_2(\zeta)$.

6.2 Roe solver

6.2.1 Derivation

The Roe solver is close to the version derived in Chap. 5. The only difference lies in the adding of a third wave. We need the following equations to write the Roe solver's algorithm:

- The velocities associated with the intermediate state
$$\hat{u} = \frac{q_{i-1}/\sqrt{h_{i-1}} + q_i/\sqrt{h_i}}{\sqrt{h_{i-1}} + \sqrt{h_i}} \text{ and } \bar{c} = \sqrt{\frac{1}{2}(h_{i-1} + h_i)}. \tag{6.9}$$

- The waves $\boldsymbol{W}_k$:
$$\boldsymbol{W}_k = \alpha_k \boldsymbol{w}_k,\ k = 1,\ 3 \tag{6.10}$$
where α_k are the components of the $\boldsymbol{\alpha}$ vector obtained by inverting the matrix $\boldsymbol{R}$
$$\boldsymbol{\alpha} = \boldsymbol{R}^{-1} \cdot \Delta\boldsymbol{Q} = \frac{1}{2\hat{c}} \begin{pmatrix} (\hat{u} + \hat{c})\Delta Q_1 - \Delta Q_2 \\ \Delta Q_3 - \phi \Delta Q_1 \\ (-\hat{u} + \hat{c})\Delta Q_1 + \Delta Q_2 \end{pmatrix} \tag{6.11}$$
where $\hat{c} = \sqrt{g\bar{h}}$ and $\Delta\boldsymbol{Q} = (\Delta Q_1,\ \Delta Q_2, ,\ \Delta Q_3)$. For the second wave, we impose that there is no jump ΔQ_1 associated with the contact discontinuity, and thus we impose
$$\alpha_2 = \Delta Q_3.$$

- the characteristic speeds
$$s_1 = \hat{u} - \bar{c},\ s_2 = \hat{u} \text{ and } s_3 = \hat{u} + \bar{c}. \tag{6.12}$$

- The fluctuations are
$$\boldsymbol{A}^+ \cdot \Delta\boldsymbol{Q}_{i-1/2} = \sum_{k=1}^{3} \min(\lambda^k_{i-1/2}, 0)\boldsymbol{W}_{k,i-1/2},$$
$$\boldsymbol{A}^- \cdot \Delta\boldsymbol{Q}_{i+1/2} = \sum_{k=1}^{3} \max(\lambda^k_{i-1/2}, 0)\boldsymbol{W}_{k,i-1/2},$$
which gives in the present context:
  - if $s_k > 0$, then `amdq(m,i)␣=␣s*wave`.
  - if $s_k < 0$, then `apdq(m,i)␣=␣s*wave`.

6.2.2 Implementation in Clawpack

```
subroutine rp1(maxmx,num_eqn,num_waves,num_aux,num_ghost,num_cells, &
               ql,qr,auxl,auxr,wave,s,amdq,apdq)

! Solve Riemann problems for the 1D shallow water equations
! with an additional passively advected tracer:
!    (h)_t + (u h)_x = 0
!    (uh)_t + ( uuh + .5*gh^2 )_x = 0
!    c_t + uc_x = 0
! using Roe's approximate Riemann solver with entropy fix for
! transonic rarefractions.

! waves: 3
! equations: 3

! Conserved quantities:
!       1 depth
!       2 momentum
!       3 tracer

! See http://www.clawpack.org/riemann.html for a detailed explanation
! of the Riemann solver API.

    implicit none

    integer, intent(in) :: maxmx, num_eqn, num_waves, num_aux, num_ghost, &
                            num_cells
    real(kind=8), intent(in), dimension(num_eqn,1-num_ghost:maxmx+num_ghost
   ) :: ql, qr
    real(kind=8), intent(in), dimension(num_aux,1-num_ghost:maxmx+num_ghost
   ) :: auxl, auxr
    real(kind=8), intent(out) :: s(num_waves, 1-num_ghost:maxmx+num_ghost)
    real(kind=8), intent(out) :: wave(num_eqn, num_waves, 1-num_ghost:maxmx
   +num_ghost)
    real(kind=8), intent(out), dimension(num_eqn,1-num_ghost:maxmx+
   num_ghost) :: amdq,apdq

    ! local variables:
    real(kind=8) :: a1,a2,ubar,cbar,s0,s1,s2,s3,hr1,uhr1,hl2,uhl2,sfract,df
    real(kind=8) :: delta(2)
    integer :: i,m,mw
    logical :: efix

    data efix /.true./    ! Use entropy fix for transonic rarefactions

    ! Gravity constant set in setprob.f or the shallow1D.py file
    real(kind=8) :: grav
    common /cparam/ grav

    ! Main loop of the Riemann solver.
    do 30 i=2-num_ghost,num_cells+num_ghost

        ! compute  Roe-averaged quantities:
        ubar = (qr(2,i-1)/dsqrt(qr(1,i-1)) + ql(2,i)/dsqrt(ql(1,i)))/ &
               ( dsqrt(qr(1,i-1)) + dsqrt(ql(1,i)) )
        cbar=dsqrt(0.5d0*grav*(qr(1,i-1) + ql(1,i)))

        ! delta(1)=h(i)-h(i-1) and  delta(2)=hu(i)-hu(i-1)
        delta(1) = ql(1,i) - qr(1,i-1)
        delta(2) = ql(2,i) - qr(2,i-1)

        ! Compute coeffs in the evector expansion of delta(1),delta(2)
        a1 = 0.5d0*(-delta(2) + (ubar + cbar) * delta(1))/cbar
        a2 = 0.5d0*( delta(2) - (ubar - cbar) * delta(1))/cbar

        ! Finally, compute the waves.
        wave(1,1,i) = a1
        wave(2,1,i) = a1*(ubar - cbar)
        wave(3,1,i) = 0.d0
        s(1,i) = ubar - cbar

        wave(1,2,i) = a2
        wave(2,2,i) = a2*(ubar + cbar)
        wave(3,2,i) = 0.d0
        s(2,i) = ubar + cbar

        wave(1,3,i) = 0.d0
        wave(2,3,i) = 0.d0
        wave(3,3,i) = ql(3,i) - qr(3,i-1)
        s(3,i) = ubar

    30 enddo

    ! Compute fluctuations amdq and apdq
    ! ------------------------------------

    if (efix) go to 110

    ! No entropy fix
    ! ----------------------------------------------
    ! amdq = SUM s*wave   over left-going waves
    ! apdq = SUM s*wave   over right-going waves

    do m=1,num_waves
        do i=2-num_ghost, num_cells+num_ghost
            amdq(m,i) = 0.d0
            apdq(m,i) = 0.d0
            do mw=1,num_waves
                if (s(mw,i) < 0.d0) then
                    amdq(m,i) = amdq(m,i) + s(mw,i)*wave(m,mw,i)
                else
                    apdq(m,i) = apdq(m,i) + s(mw,i)*wave(m,mw,i)
                endif
            enddo
        enddo
    enddo

    ! with no entropy fix we are done...
    return

    ! -----------------------------------------------
    110 continue

    ! compute the rightgoing flux differences:
    ! df = SUM s*wave   is the total flux difference and apdq = df - amdq

    do i = 2-num_ghost, num_cells+num_ghost
        do m=1,2
            df = 0.d0
            do mw=1,2
                df = df + s(mw,i)*wave(m,mw,i)
                enddo
            apdq(m,i) = df - amdq(m,i)
            enddo
        ! tracer (which is in non-conservation form)
        if (s(3,i) < 0) then
            amdq(m,i) = amdq(m,i) + s(3,i)*wave(m,3,i)
          else
            apdq(m,i) = apdq(m,i) + s(3,i)*wave(m,3,i)
          endif

        enddo

    return

end subroutine rp1
```

6.2.3 Implementation in Pyclaw

```
def shallow_roe_1D(q_l, q_r, aux_l, aux_r, problem_data):
    r"""
    Roe shallow water solver in 1d
    """

    # Array shapes
    num_rp    = q_l.shape[1]
    num_eqn   = 3
    num_waves = 3

    g = problem_data['grav']

    # Output arrays
    wave = np.empty( (num_eqn, num_waves, num_rp) )
    s = np.zeros( (num_waves, num_rp) )
    amdq = np.zeros( (num_eqn, num_rp) )
    apdq = np.zeros( (num_eqn, num_rp) )

    # Compute roe-averaged quantities
    ubar = ( (q_l[1,:]/np.sqrt(q_l[0,:]) + q_r[1,:]/np.sqrt(q_r[0,:])) /
             (np.sqrt(q_l[0,:]) + np.sqrt(q_r[0,:])) )
    cbar = np.sqrt(0.5 * g * (q_l[0,:] + q_r[0,:]))

    # Compute Flux structure
    delta  = q_r - q_l
    delta1 = q_r[0,:] - q_l[0,:]
    delta2 = q_r[1,:] - q_l[1,:]
    alpha1 = 0.5 * (-delta2 + (ubar + cbar) * delta1) / cbar
    alpha2 = 0.5 * ( delta2 - (ubar - cbar) * delta1) / cbar

    # Compute each family of waves
    wave[0,0,:] = alpha1
    wave[1,0,:] = alpha1 * (ubar - cbar)
    wave[2,0,:] = 0.
    s[0,:]      = ubar - cbar

    wave[0,2,:] = alpha2
    wave[1,2,:] = alpha2 * (ubar + cbar)
    wave[2,2,:] = 0.
    s[2,:]      = ubar + cbar

    wave[0,1,:] = 0.
    wave[1,1,:] = 0.
    wave[2,1,:] = q_r[2,:] - q_l[2,:]
    s[1,:]      = ubar

    s_index = np.zeros((3,num_rp))
    for m in range(num_eqn):
            for mw in range(num_waves):
                s_index[0,:] = s[mw,:]
                amdq[m,:] += np.min(s_index,axis=0) * wave[m,mw,:]
                apdq[m,:] += np.max(s_index,axis=0) * wave[m,mw,:]


    return wave, s, amdq, apdq
```

6.3 HLLC Solver

6.3.1 *Principle*

The HLLC solver is an extension of the HLL scheme proposed by Eleuterio Toro (Toro, 2001) to cope with the existence of a contact discontinuity. The HLL solver defines an intermediate state separating the left and right initial states. The HLLC introduces two distinct intermediate states split by the second characteristic $x = \lambda_2 t$ (see Fig. 6.1). The fluxes associated with the two intermediate states are defined using the Rankine-Hugoniot equation:

$$\boldsymbol{F}_{*,l} - \boldsymbol{F}_l = s_1(\boldsymbol{Q}_{*,l} - \boldsymbol{Q}_l), \tag{6.13}$$

$$\boldsymbol{F}_{*,r} - \boldsymbol{F}_r = s_3(\boldsymbol{Q}_{*,r} - \boldsymbol{Q}_r) \tag{6.14}$$

where $\boldsymbol{F}_{*,l} = \boldsymbol{f}(\boldsymbol{Q}_{*,l})$ and $\boldsymbol{F}_{*,r} = \boldsymbol{f}(\boldsymbol{Q}_{*,r})$.

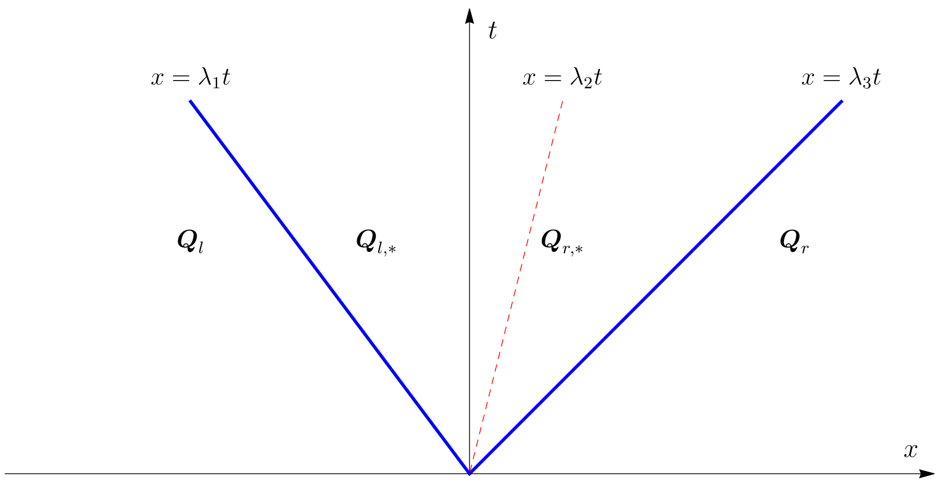


Figure 6.1 The three waves separating the left and right initial states.

In the absence of the advection equation (6.3), there would be only one intermediate state. As tracer advection does not interplay with water flow, we impose that the first two components (those associated with the water flow) of $\boldsymbol{F}_{*,r}$ and $\boldsymbol{F}_{*,l}$ are identical:

$$F_{*,l,1} = F_{*,r,1} = \frac{\lambda_3 F_{l,1} - \lambda_1 F_{r,1}}{\lambda_3 - \lambda_1} - \lambda_1\lambda_3 \frac{\lambda_3 - \lambda_1}{h_l - h_r}, \tag{6.15}$$

$$F_{*,l,2} = F_{*,r,2} = \frac{\lambda_3 F_{l,2} - \lambda_1 F_{r,2}}{\lambda_3 - \lambda_1} - \lambda_1\lambda_3 \frac{\lambda_3 - \lambda_1}{h_l u_l - h_r u_r}, \tag{6.16}$$

where $F_1 = hu$ and $F_2 = hu^2 + gh^2/2$. For the third component, we impose that there is no jump across the 1- and 3-characteristics. The only jump in $hu\phi$ is across $x = \lambda_2 t$. Because the third component $hu\phi$ is the product of ϕ and the first $\boldsymbol{F}$ component hu, then we can write

$$F_{*,l,3} = F_{*,l,1}\phi_l, \tag{6.17}$$

$$F_{*,r,3} = F_{*,r,2}\phi_r. \tag{6.18}$$

The flux at the interface $x = 0$ is thus $F_{*,l,3}$ if $\lambda_2 > 0$, and $F_{*,r,3}$ if $\lambda_2 > 0$.

An estimate of the wave speed λ_2 is (Toro, 2001):

$$\lambda_2 = \frac{\lambda_1 h_r(u_r - \lambda_3) - \lambda_3 h_l(u_l - \lambda_1)}{h_r(u_r - \lambda_3) - h_l(u_l - \lambda_1)}. \tag{6.19}$$

We consider three waves

$$\boldsymbol{W}_1 = \boldsymbol{Q}_{l,*} - \boldsymbol{Q}_l,\ \boldsymbol{W}_2 = \boldsymbol{Q}_{r,*} - \boldsymbol{Q}_{l,*} \text{ and } \boldsymbol{W}_3 = \boldsymbol{Q}_r - \boldsymbol{Q}_{r,*}. \tag{6.20}$$

In § 2.6 and 5.3.1, we have shown that the intermediate state for the water flow is:

$$\boldsymbol{Q}^\dagger_* = \frac{s_3\boldsymbol{Q}^\dagger_r - s_1\boldsymbol{Q}^\dagger_l}{s_3 - s_1} - \frac{\boldsymbol{F}^\dagger_r - \boldsymbol{F}^\dagger_l}{s_3 - s_1},$$

where $\boldsymbol{Q}^\dagger_* = (h,\ hu)$ and $\boldsymbol{F}^\dagger = (q,\ \Phi)$ (with $q = hu$ and $\Phi = hu^2 + gh^2/2$) are the first two components of $\boldsymbol{Q}$ and $\boldsymbol{F}$. We thus have

$$h_* = \frac{s_3 h_r - s_1 h_l}{s_3 - s_1} - \frac{s_3 q_r - s_1 q_l}{s_3 - s_1},$$

and

$$q_* = (hu)_* = \frac{s_3 q_r - s_1 q_l}{s_3 - s_1} - \frac{s_3\Phi_r - s_1\Phi_l}{s_3 - s_1}.$$

We then deduce:

$$\boldsymbol{W}^\dagger_1 = \begin{pmatrix} h_* - h_l \\ q_* - q_l \end{pmatrix},\ \boldsymbol{W}^\dagger_2 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \text{ and } \boldsymbol{W}^\dagger_3 = \begin{pmatrix} h_r - h_* \\ q_r - q_* \end{pmatrix}.$$

For the third component, we have

$$W_{1,3} = 0,\ W_{2,3} = \phi_r - \phi_l,\ \text{and } W_{1,3} = 0.$$

6.3.2 Implementation in Pyclaw

```python
def shallow_hllc_1D(q_l,q_r,aux_l,aux_r,problem_data):
    r"""
    HLLC shallow water solver ::
    """
    # Array shapes
    num_rp    = q_l.shape[1]
    num_eqn   = 3
    num_waves = 3

    g = problem_data['grav']

    # Output arrays
    wave = np.empty( (num_eqn, num_waves, num_rp) )
    s    = np.empty( (num_waves, num_rp) )
    amdq = np.zeros( (num_eqn, num_rp) )
    apdq = np.zeros( (num_eqn, num_rp) )

    h_l  = q_l[0,:]
    h_r  = q_r[0,:]
    hu_l = q_l[1,:]
    hu_r = q_r[1,:]
    u_r = hu_r/h_r
    c_r = np.sqrt(g * h_r)
    u_l = hu_l/h_l
    c_l = np.sqrt(g * h_l)
    Phi_l = u_l**2*h_l+0.5*g*h_l**2
    Phi_r = u_r**2*h_r+0.5*g*h_r**2

    # Compute Roe and right and left speeds
    u_hat = (hu_l/np.sqrt(h_l) + hu_r/np.sqrt(h_r))/(np.sqrt(h_l) + np.sqrt
(h_r))
    c_hat = np.sqrt(0.5 * g * (h_r + h_l))

    # Compute Einfeldt speeds
    s_index = np.empty((2,num_rp))
    s_index[0,:] = u_hat - c_hat
    s_index[1,:] = u_l - c_l
    s[0,:] = np.min(s_index,axis=0)
    s_index[0,:] = u_r + c_r
    s_index[1,:] = u_hat + c_hat
    s[2,:] = np.max(s_index,axis=0)

    lambda_1 = u_hat - c_hat
    lambda_3 = u_hat + c_hat
    u_toro = (lambda_1*h_r*(u_r-lambda_3) - lambda_3*h_l*(u_l-lambda_1)   )
        \
              /(h_r*(u_r-lambda_3) - h_l*(u_l-lambda_1))
    s[1,:] = u_hat

    # Compute middle state
    h_star  = (h_r * s[2,:] - h_l * s[0,:]-(hu_r-hu_l))/(s[2,:]-s[0,:])
    hu_star = (hu_r * s[2,:] - hu_l * s[0,:]-(Phi_r-Phi_l))/(s[2,:]-s[0,:])

    # Compute each family of waves
    wave[0,0,:] = h_star - h_l
    wave[1,0,:] = hu_star - hu_l
    wave[2,0,:] = 0.
    wave[0,1,:] = 0.
    wave[1,1,:] = 0.
    wave[2,1,:] = q_r[2,:]-q_l[2,:]
    wave[0,2,:] = h_r - h_star
    wave[1,2,:] = hu_r - hu_star
    wave[2,2,:] = 0.

    # Compute variations
    s_index = np.zeros((3,num_rp))
    for m in range(num_eqn):
        for mw in range(num_waves):
            s_index[0,:] = s[mw,:]
            amdq[m,:] += np.min(s_index,axis=0) * wave[m,mw,:]
            apdq[m,:] += np.max(s_index,axis=0) * wave[m,mw,:]

    return wave, s, amdq, apdq
```

6.4 F-wave formulation

6.4.1 *Principle*

The f-wave method consists of decomposing the jump in the flux (6.5) into three f-waves

$$\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1}) = \sum_{k=1}^{3} \boldsymbol{Z}_{k,i-1/2},$$

where the f-wave $\boldsymbol{Z}_{k,i-1/2}$ can be related to the right eigenvector $\hat{\boldsymbol{w}}_{k,i-1/2}$ of the Roe matrix:

$$\boldsymbol{Z}_{k,i-1/2} = \beta_{k,i-1/2}\hat{\boldsymbol{w}}_{k,i-1/2}$$

where the coefficient $\beta_{k,i-1/2}$ is the linear solution (see § 2.7):

$$\boldsymbol{\beta}_{i-1/2} = \boldsymbol{L} \cdot (\boldsymbol{f}(\boldsymbol{Q}_i) - \boldsymbol{f}(\boldsymbol{Q}_{i-1})).$$

with $\boldsymbol{L} = \boldsymbol{R}^{-1}$. We find that:

$$\boldsymbol{\beta}_{i-1/2} = \frac{1}{2\hat{c}} \left(\begin{array}{c} \Phi_l - \Phi_r + (\hat{u}+\hat{c})(q_r - q_l) \\ 2\hat{c}(\phi_r q_r - \phi_l q_l + \phi(q_r - q_l)) \\ \Phi_r - \Phi_l - (\hat{u}-\hat{c})(q_r - q_l) \end{array} \right), \tag{6.21}$$

where $\Phi = hu^2 + gh^2/2$. The f-waves are then:

$$\boldsymbol{Z}_{1,i-1/2} = \boldsymbol{\beta}_{1,i-1/2}\boldsymbol{w}_1 = \frac{\Phi_l - \Phi_r + (\hat{u}+\hat{c})(q_r - q_l)}{2\hat{c}} \left(\begin{array}{c} 1 \\ \hat{u}-\hat{c} \\ \phi \end{array} \right),$$

$$\boldsymbol{Z}_{2,i-1/2} = \boldsymbol{\beta}_{2,i-1/2}\boldsymbol{w}_2 = (\phi_r q_r - \phi_l q_l - \phi(q_r - q_l)) \left(\begin{array}{c} 0 \\ 0 \\ 1 \end{array} \right)$$

and

$$\boldsymbol{Z}_{3,i-1/2} = \boldsymbol{\beta}_{3,i-1/2}\boldsymbol{w}_3 = \frac{\Phi_r - \Phi_l - (\hat{u}-\hat{c})(q_r - q_l)}{2\hat{c}} \left(\begin{array}{c} 1 \\ \hat{u}+\hat{c} \\ \phi \end{array} \right).$$

As for the Roe solver, we assume that there is no jump in ϕ for the 1- and 3- shock waves while for the 2-wave, there is no jump in h and hu (and so $q_r = q_l = \hat{q} = \hat{u}\bar{h}$), and so the correct f-waves are:

$$\boldsymbol{Z}_{1,i-1/2} = \boldsymbol{\beta}_{1,i-1/2}\boldsymbol{w}_1 = \frac{\Phi_l - \Phi_r + (\hat{u}+\hat{c})(q_r - q_l)}{2\hat{c}} \begin{pmatrix} 1 \\ \hat{u}-\hat{c} \\ 0 \end{pmatrix}, \tag{6.22}$$

$$\boldsymbol{Z}_{2,i-1/2} = \boldsymbol{\beta}_{2,i-1/2}\boldsymbol{w}_2 = (\phi_r - \phi_l)\hat{q} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{6.23}$$

and

$$\boldsymbol{Z}_{3,i-1/2} = \boldsymbol{\beta}_{3,i-1/2}\boldsymbol{w}_3 = \frac{\Phi_r - \Phi_l - (\hat{u}-\hat{c})(q_r - q_l)}{2\hat{c}} \begin{pmatrix} 1 \\ \hat{u}+\hat{c} \\ 0 \end{pmatrix}. \tag{6.24}$$

6.4.2 Implementation in Pyclaw

```python
def shallow_hllc_fwave_1d(q_l, q_r, aux_l, aux_r, problem_data):
    r"""Shallow water Riemann solver using fwaves
    """

    g             = problem_data['grav']
    dry_tolerance = problem_data['dry_tolerance']

    num_rp    = q_l.shape[1]
    num_eqn   = 3
    num_waves = 3

    # Initializing arrays
    fwave = np.empty( (num_eqn, num_waves, num_rp) )
    s     = np.empty( (num_waves, num_rp) )
    amdq  = np.zeros( (num_eqn, num_rp) )
    apdq  = np.zeros( (num_eqn, num_rp) )
    r1    = np.zeros( (num_waves, num_rp) )
    r2    = np.zeros( (num_waves, num_rp) )
    r3    = np.zeros( (num_waves, num_rp) )

    # Extract state
    h_l  = q_l[0, :]
    h_r  = q_r[0, :]
    hu_l = q_l[1, :]
    hu_r = q_r[1, :]
    u_l  = np.where(h_l > dry_tolerance, hu_l / h_l, 0.0)
    u_r  = np.where(h_r > dry_tolerance, hu_r / h_r, 0.0)

    # Flux and Roe depth
    phi_l = h_l * u_l**2 + 0.5 * g * h_l**2
    phi_r = h_r * u_r**2 + 0.5 * g * h_r**2
    h_bar = 0.5 * (h_l + h_r)

    # Speeds
    u_hat = (np.sqrt(h_l)*u_l + np.sqrt(h_r)*u_r)/ (np.sqrt(h_l) + np.sqrt(
   h_r))
    c_hat = np.sqrt(g * h_bar)

    s[0, :] = np.amin(np.vstack((u_l - np.sqrt(g * h_l), u_hat - c_hat)),
   axis=0)
    s[1, :] = u_hat
    s[2, :] = np.amax(np.vstack((u_r + np.sqrt(g * h_r), u_hat + c_hat)),
   axis=0)

    beta1 = (phi_l - phi_r + (u_hat+c_hat)*(hu_r-hu_l))/2./c_hat
    beta2 = (q_r[2, :]- q_l[2, :])*u_hat
    beta3 = (phi_r - phi_l - (u_hat-c_hat)*(hu_r-hu_l))/2./c_hat

    r1[0, :] = 1.
    r1[1, :] = u_hat - c_hat
    r1[2, :] = 0.

    r2[0, :] = 0.
    r2[1, :] = 0.
    r2[2, :] = 1.

    r3[0, :] = 1.
    r3[1, :] = u_hat + c_hat
    r3[2, :] = 0.

    fwave[0, 0, :] = beta1 * r1[0, :]
    fwave[1, 0, :] = beta1 * r1[1, :]
    fwave[2, 0, :] = beta1 * r1[2, :]

    fwave[0, 1, :] = beta2 * r2[0, :]
    fwave[1, 1, :] = beta2 * r2[1, :]
    fwave[2, 1, :] = beta2 * r2[2, :]

    fwave[0, 2, :] = beta3 * r3[0, :]
    fwave[1, 2, :] = beta3 * r3[1, :]
    fwave[2, 2, :] = beta3 * r3[2, :]

    for m in range(num_eqn):
        for mw in range(num_waves):
            amdq[m, :] += (s[mw, :] < 0.0) * fwave[m, mw, :]
            apdq[m, :] += (s[mw, :] > 0.0) * fwave[m, mw, :]

    return fwave, s, amdq, apdq
```

6.5 Example: dam break

We consider a dam break problem with the following initial conditions: $h_l = 3$ m et $u_l = 0$ for $x \leq 0$, and $h_l = 1$ m et $u_l = 0$ for $x > 0$. We compare the two solvers: Roe (with no entropy fix) and the f-wave formulation of the HLLC solver. Figures 6.2 shows the comparison.

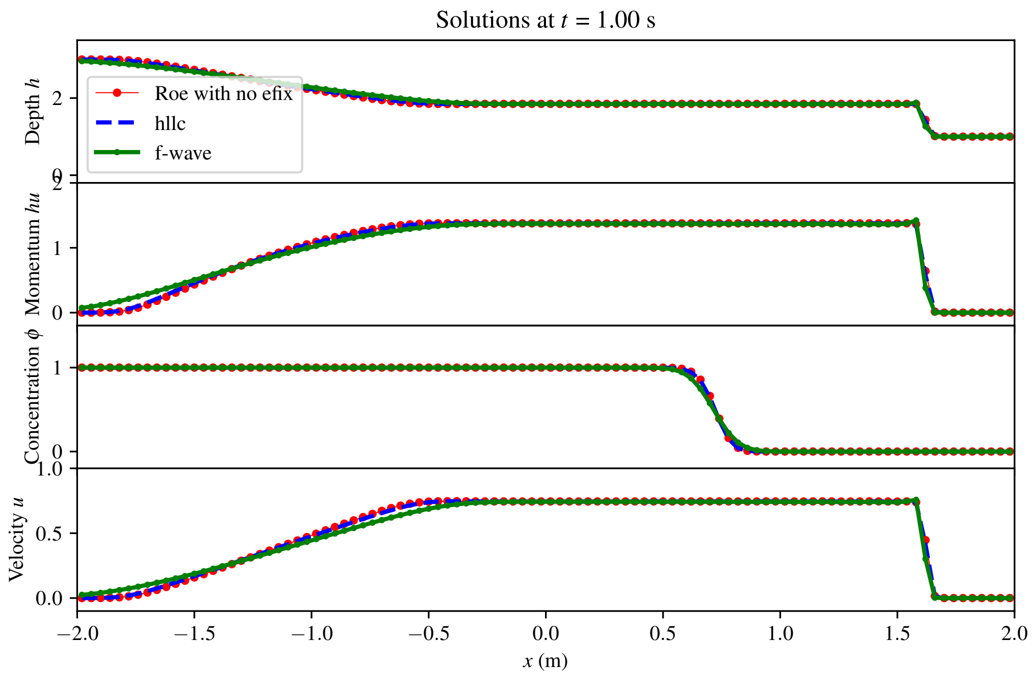
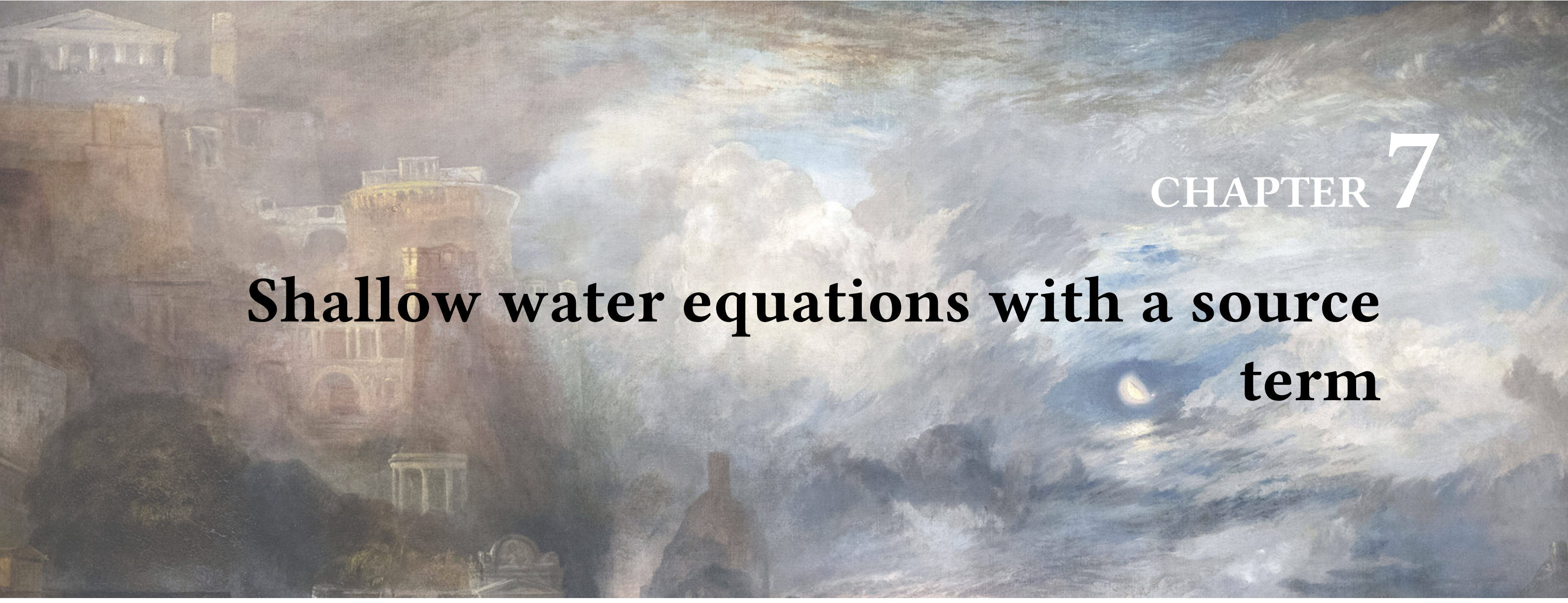


Figure 6.2 Comparison of the three methods: Roe solver, HLLC, and the f-wave variant of the HLLC algorithm. Computations done with $g = 1$ m/s^2.



CHAPTER 7

Shallow water equations with a source term

7.1 Theory

7.1.1 *flow resistance*

In an one-dimensional fixed Cartesian frame, the Saint-Venant equations take the tensorial form

$$\frac{\partial}{\partial t}\boldsymbol{Q} + \nabla \boldsymbol{f}(\boldsymbol{Q}) = \boldsymbol{S}, \tag{7.1}$$

where $\boldsymbol{Q} = (h, hu)$ is the unknown, and $\boldsymbol{S} = (0, S)$ is the source term. The computation strategy involves first solving the homogenous problem (LeVeque, 2002) :

$$\frac{\partial}{\partial t}\boldsymbol{Q} + \nabla \boldsymbol{f}(\boldsymbol{Q}) = 0, \tag{7.2}$$

then correcting the solution by taking the effect of the source term on the momentum $q = hu$:

$$\varrho\frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{q} = S(\boldsymbol{Q}), \tag{7.3}$$

where $S(\boldsymbol{Q})$ takes the following form if we consider a flow experiencing flow resistance:

$$\begin{aligned} S(\boldsymbol{U}) &= -\frac{\varrho g}{K^2 h^{1/3}}|u|u, && (7.4)\\ &= -\frac{\varrho g}{K^2 h^{7/3}}|q|q, && (7.5) \end{aligned}$$

where K is the Manning-Strickler coefficient.

Let us assume that we have computed the solution $\boldsymbol{q}_*$ to the homogenous equation (7.2), and we are now seeking the solution at time $k+1$. Using a semi-implicit discretization of (7.3) leads to

$$\begin{aligned} q^{k+1} &= q^* - \mathrm{d}t\frac{g}{K^2 h^{7/3}}|q^*|q^{k+1}, && (7.6)\\ q^* &= q^{k+1}\left(1 + \frac{g\mathrm{d}t}{K^2 h^{7/3}}|q^*|\right), && (7.7)\\ q^{k+1} &= \frac{q^*}{1 + \mathrm{d}t\dfrac{g}{K^2 h^{7/3}}|q^*|}. && (7.8) \end{aligned}$$

Antuono, M. 2010 A shock solution for the nonlinear shallow water equations. *Journal of Fluid Mechanics* **658**, 166–187.

Berger, M. J., George, D. L., LeVeque, R. J. *&* Mandli, K. T. 2011 The GeoClaw software for depth-averaged flows with adaptive refinement. *Advances in Water Resources* **34**, 1195–1206.

Berger, M. J. *&* LeVeque, R. J. 1998 Adaptive mesh refinement using wave-propagation algorithms for hyperbolic systems. *SIAM Journal on Numerical Analysis* **35** (6), 2298–2316.

Bladé, E., Cea, L., Corestein, G., Escolano, E., Puertas, J., Váquez-Cendón, E., Dolz, J. *&* Coll, A. 2014 Iber: herramienta de simulación numérica del flujo en ríos. *Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería* **30**, 1–10.

Cea, L. *&* Bladé, E. 2015 A simple and efficient unstructured finite volume scheme for solving the shallow water equations in overland flow applications. *Water Resources Research* **51**, 5464–5486.

Einfeldt, B. 1988 On Godunov-type methods for gas dynamics. *SIAM Journal on Numerical Analysis* **25** (2), 294–318.

George, D. L. 2006 Finite Volume Methods and Adaptive Refinement for Tsunami Propagation and Inundation. PhD thesis, University of Washington.

George, D. L. 2008 Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation. *Journal of Computational Physics* **227**, 3089–3113.

George, D. L. 2011 Adaptive finite volume methods with well-balanced Riemann solvers for modeling floods in rugged terrain: Application to the Malpasset dam-break flood (France, 1959). *International Journal for Numerical Methods in Fluids* **66**, 1000–1018.

Godunov, S. K. 1959 Finite difference method for numerical computation of discontinuous solutions of the equations of fluid dynamics (English translation by I. Bohachevsky). *Matematičeskij sbornik* **47**, 271–376.

Godunov, S. K. 1962 The problem of a generalized solution in the theory of quasilinear equations and in gas dynamics. *Uspekhi Matematicheskikh Nauk* **17** (3), 147–158.

Guinot, V. 2010 *Wave Propagation in Fluids—Models and Numerical Techniques*. Hoboken: John Wiley & Sons.

Harten, A. 1983 High resolution schemes for hyperbolic conservation laws. *Journal of computational physics* **49**, 357–393.

Harten, A., Lax, P. D. *&* van Leer, B. 1983 On upstream differencing and Godunov-type schemes for hyperbolic conservation laws. *SIAM Review* **25**, 35–61.

Holden, H. & Risebro, N. H. 2015 *Front Tracking for Hyperbolic Conservation lLaws*, , vol. 152. Berlin: Springer.

Kalisch, H., Mitrovic, D. & Teyekpiti, V. 2017 Delta shock waves in shallow water flow. *Physics Letters A* **381** (13), 1138–1144.

Ketcheson, D. I., Gottlieb, S. & Macdonald, C. B. 2011 Strong stability preserving two-step Runge–Kutta methods. *SIAM Journal on Numerical Analysis* **49** (6), 2618–2639.

Ketcheson, D. I., LeVeque, R. J. & del Razo, M. 2020 *Riemann Problems and Jupyter Solutions*. Philadelphia: Society for Industrial and Applied Mathematics.

Ketcheson, D. I., Mandli, K. T., Ahmadia, A. J., Alghamdi, A., Quezada de Luna, M., Parsani, M., Knepley, M. G. & Emmett, M. 2012 PyClaw: Accessible, extensible, scalable tools for wave propagation problems. *SIAM Journal on Scientific Computing* **34** (4), C210–C231.

Ketcheson, D. I., Parsani, M. & LeVeque, R. J. 2013 High-order wave propagation algorithms for hyperbolic systems. *SIAM Journal on Scientific Computing* **35** (1), A351–A377.

van Leer, B. 1979 Towards the ultimate conservative difference scheme. V. A second-order sequel to Godunov's method. *Journal of computational Physics* **32** (1), 101–136.

LeVeque, R. J. 1992 *Numerical Methods for Conservation Laws*. Basel: Birkhäuser.

LeVeque, R. J. 2002 *Finite Volume Methods for Hyperbolic Problems*. Cambridge: Cambridge University Press.

Mandli, K. T., Ahmadia, A. J., Berger, M., Calhoun, D., George, D. L., Hadjimichael, Y., Ketcheson, D. I., Lemoine, G. I. & LeVeque, R. J. 2016 Clawpack: building an open source ecosystem for solving hyperbolic PDEs. *PeerJ Computer Science* **2**, e68.

Paulsen, M. O. & Kalisch, H. 2020 Admissibility conditions for Riemann data in shallow water theory. *Zeitschrift für Naturforschung* **75A** (7), 637–648.

Peruzzetto, M., Mangeney, A., Bouchut, F., Grandjean, G., Levy, C., Thiery, Y. & Lucas, A. 2021 Topography curvature effects in thin-layer models for gravity-driven flows without bed erosion. *Journal of Geophysical Research: Earth Surface* **126**, e2020JF005657.

Rayleigh, J. W. S. 1914 On the theory of long waves and bores. *Proceedings of the Royal Society of London series A* **90** (619), 324–328.

Richard, G. L. & Gavrilyuk, S. L. 2013 The classical hydraulic jump in a model of shear shallow-water flows. *Journal of Fluid Mechanics* **725**, 492–521.

Roe, P. L. 1981 Approximate Riemann solvers, parameters vectors, and difference schemes. *Journal of Computational Physics* **44**, 357–372.

Saint-Venant, A. B. d. 1871 Théorie du mouvement non permanent des eaux, avec application aux crues des rivières et à l'introduction des marées dans leur lit. *Comptes Rendus de l'Académie des Sciences, série I* **173**, 147–154– 237–240.

Shu, C.-W. 1998 Essentially Non-oscillatory and Weighted Essentially Non-oscillatory Schemes for Hyperbolic Conservation Laws. In *Advanced Numerical Approximation of Nonlinear Hyperbolic Equations, Lectures given at the 2nd Session of the Centro Internazionale Matematico Estivo (C. I. M. E.) held in Cetraro, Italy, June 23-28, 1997* (ed. A. Quarteroni), , vol. 1697. Berlin: Springer.

Shu, C.-W. 2020 Essentially non-oscillatory and weighted essentially non-oscillatory schemes. In *Acta Numerica*, , vol. 29, pp. 701–762. Cambridge University Press.

Smoller, J. 1982 *Shock Waves and Reaction-Diffusion Equations*. New York: Springer.

Stoker, J. J. 1957 *Water Waves*. New York: Interscience Publishers.

Sweby, P. K. 1984 High resolution schemes using flux limiters for hyperbolic conservation laws. *SIAM journal on numerical analysis* **21** (5), 995–1011.

Toro, E. F. 2001 *Shock-Capturing Methods for Free-Surface Shallow Flows*. Chichester: Wiley.

Toro, E. F. 2009 *Riemann Solvers and Numerical Methods for Fluid Dynamics—A Practical Introduction*. Dordrecht: Springer.

Toro, E. F. 2019 The HLLC Riemann solver. *Shock Waves* **29**, 1065–1082.

Toro, E. F. *&* Garcia-Navarro, P. 2007 Godunov-type methods for free-surface shallow flows: A review. *Journal of Hydraulic Research* **45**, 736–751.

Whitham, G. B. 1974 *Linear and Nonlinear Waves*. New York: John Wiley & Sons.