Appendix

A.1 Index Notation

Many equations in physics deal with vector quantities which have both a magnitude and an orientation in physical space. For instance, the simplified form of Newton's second law,

$$\boldsymbol{f} = m\boldsymbol{a}, \tag{A.1}$$

connects the vector quantity of force $\boldsymbol{f}$ and the vector quantity of acceleration $\boldsymbol{a}$, both having the same orientation.

Equations such as this can also be expressed more explicitly as three scalar equations, one for each spatial direction:

$$f_x = ma_x, \quad f_y = ma_y, \quad f_z = ma_z. \tag{A.2}$$

However, it is cumbersome to write all three equations in this way, especially when their only difference is that their index changes between x, y, and z. Instead, we can represent the same equation using only one generic index $\alpha \in \{x, y, z\}$ as

$$f_\alpha = ma_\alpha. \tag{A.3}$$

This style of notation, called *index notation*, retains the explicitness of the notation in (A.2) while remaining as brief as (A.1).

With simple vector equations like this, the advantage of index notation might not seem all that great. However, vectors are only first-order *tensors* (scalars being zeroth-order tensors). We can also apply index notation to a second-order tensor (or matrix) $\boldsymbol{A}$ by pointing to a generic element as $A_{\alpha\beta}$, α and β being two generic indices that may or may not be different. Higher-order tensors are equally explicit:

T. Krüger et al., *The Lattice Boltzmann Method*, Graduate Texts in Physics,
DOI 10.1007/978-3-319-44649-3

a generic element of the third-order tensor $\boldsymbol{R}$ is $R_{\alpha\beta\gamma}$. Indeed, this style of notation lets us immediately see the order of the tensor from the number of unique indices.

Another strength of index notation is that it allows the use of the *Einstein summation convention* where repeating the same index twice in a single term implies summation over all possible values of that index. Thus, the dot product of the vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ can be expressed as

$$a_\alpha b_\alpha = \sum_\alpha a_\alpha b_\alpha = a_x b_x + a_y b_y + a_z b_z = \boldsymbol{a} \cdot \boldsymbol{b}. \tag{A.4}$$

The dot product can be expressed equally briefly in index and vector notation.

The dot product is expressed in index notation using only the Einstein summation convention, while the vector notation uses a specific, dedicated symbol "$\cdot$" to express it. For the dyadic product,

$$\boldsymbol{A} = \boldsymbol{a} \otimes \boldsymbol{b} \quad \Leftrightarrow \quad A_{\alpha\beta} = a_\alpha b_\beta, \tag{A.5}$$

the vector notation requires yet another specific, dedicated symbol "$\otimes$" while the index notation is explicit and clear: the $\alpha\beta$-component of the second-order tensor $\boldsymbol{A}$ equals the product of the α-component of the vector $\boldsymbol{a}$ and the β-component of the vector $\boldsymbol{b}$.

We may also use index notation to generalise coordinate notation: a general component of the spatial coordinate vector $\boldsymbol{x} = (x, y, z) = (x_1, x_2, x_3)$ can be written as x_α. In this way, we can also express, e.g., gradients in index notation:

$$\nabla\lambda(\boldsymbol{x}) \quad \Leftrightarrow \quad \frac{\partial\lambda(\boldsymbol{x})}{\partial x_\alpha} \quad \Leftrightarrow \quad \partial_\alpha\lambda(\boldsymbol{x}). \tag{A.6}$$

The third option is a common shorthand for derivatives, used throughout the literature and this book. Similarly, the time derivative can be expressed using the shorthand $\partial\lambda(t)/\partial t = \partial_t\lambda(t)$.

Most common vector and tensor operations can be conveniently expressed in index notation, as shown in Table A.1. One exception to this convenience is the always inconvenient cross product, which must be expressed using the *Levi-Civita symbol*

$$\varepsilon_{\alpha\beta\gamma} = \begin{cases} +1 & \text{if } (\alpha, \beta, \gamma) \text{ is } (1,2,3), (3,1,2) \text{ or } (2,3,1), \\ -1 & \text{if } (\alpha, \beta, \gamma) \text{ is } (3,2,1), (1,3,2) \text{ or } (2,1,3), \\ 0 & \text{if } \alpha = \beta, \beta = \gamma, \text{or } \gamma = \alpha. \end{cases} \tag{A.7}$$

However, while the cross product is widely used in fields like electromagnetics, it is far less used for the topics covered in this book.

Table A.1 Examples of common operations in vector and index notation, including an index notation shorthand for derivatives

Operation	Vector notation	Index notation	Shorthand
Vector dot product	$\lambda = \boldsymbol{a} \cdot \boldsymbol{b}$	$\lambda = a_\alpha b_\alpha$	
Vector outer product	$\boldsymbol{A} = \boldsymbol{a} \otimes \boldsymbol{b}$	$A_{\alpha\beta} = a_\alpha b_\beta$	
Vector cross product	$\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}$	$c_\alpha = \varepsilon_{\alpha\beta\gamma} a_\beta b_\gamma$	
Tensor contraction	$\lambda = \boldsymbol{A} : \boldsymbol{B}$	$\lambda = A_{\alpha\beta} B_{\alpha\beta}$	
Gradient	$\boldsymbol{a} = \boldsymbol{\nabla} \lambda$	$a_\alpha = \partial\lambda / \partial x_\alpha$	$a_\alpha = \partial_\alpha \lambda$
Laplacian	$\Lambda = \nabla^2 \lambda$	$\Lambda = \partial^2 \lambda / (\partial x_\alpha \partial x_\alpha)$	$\Lambda = \partial_\alpha \partial_\alpha \lambda$
1st order tensor divergence	$\lambda = \boldsymbol{\nabla} \cdot \boldsymbol{a}$	$\lambda = \partial a_\alpha / \partial x_\alpha$	$\lambda = \partial_\alpha a_\alpha$
2nd order tensor divergence	$\boldsymbol{a} = \boldsymbol{\nabla} \cdot \boldsymbol{A}$	$a_\alpha = \partial A_{\alpha\beta} / \partial x_\beta$	$a_\alpha = \partial_\beta A_{\alpha\beta}$
3rd order tensor divergence	$\boldsymbol{A} = \boldsymbol{\nabla} \cdot \boldsymbol{R}$	$A_{\alpha\beta} = \partial R_{\alpha\beta\gamma} / \partial x_\gamma$	$A_{\alpha\beta} = \partial_\gamma R_{\alpha\beta\gamma}$

In this book, we use Greek indices for the Cartesian indices x, y, and z. We also use Roman indices such as i, j, and k for non-Cartesian indices; typically to index discrete velocities as e.g. $\boldsymbol{\xi}_i$. Einstein's summation convention is used *only* for the Cartesian indices.

A.2 Details in the Chapman-Enskog Analysis

A.2.1 Higher-Order Terms in the Taylor-Expanded LBE

In (4.5) we found the Taylor expansion of the $f_i(\boldsymbol{x} + \boldsymbol{c_i}\Delta t, t + \Delta t) - f_i(\boldsymbol{x}, t)$ terms in the LBE to be

$$\sum_{n=1}^{\infty} \frac{\Delta t^n}{n!} (\partial_t + c_{i\alpha} \partial_\alpha)^n \,. \tag{A.8}$$

We neglected terms at third order and higher in the subsequent analysis. If we can show that these terms are at least two orders higher in Kn than the lowest-order terms, this neglection is justified. This is because the ansatz was that it is only necessary to keep the two lowest orders in Kn. Let us take a closer look.

Recall that the Knudsen number is $\mathrm{Kn} = \ell_{\mathrm{mfp}}/\ell$, where ℓ_{mfp} is the mean free path and ℓ is a macroscopic length scale. Kn can be related to a similar ratio in times instead of lengths using the speed of sound c_s. As c_s is on the order of the mean particle speed in the gas [1], we find that the mean time between collisions is $\mathcal{T}_{\mathrm{mfp}} = O(\ell_{\mathrm{mfp}}/c_\mathrm{s})$. Additionally, we can define an acoustic time scale as $\mathcal{T}_{c_\mathrm{s}} = \ell/c_\mathrm{s}$, this being the time it takes for an acoustic disturbance to be felt across the length scale ℓ. Together, these two relations show that $\mathrm{Kn} = \ell_{\mathrm{mfp}}/\ell = O(\mathcal{T}_{\mathrm{mfp}}/\mathcal{T}_{c_\mathrm{s}})$.

Collisions are the mechanism by which the distribution function relaxes to equilibrium, and relatively few collisions are required for this[1], so $\tau = O(\mathcal{T}_{\text{mfp}})$. From the space and time discretisation of Sect. 3.5, we know that $\Delta t = O(\tau)$: $\tau/\Delta t$ must be larger than 0.5 for reasons of linear stability, while we should avoid choosing $\tau/\Delta t \gg 1$ for reasons of accuracy, explained in Sect. 4.5.

The *acoustic* time scale $\mathcal{T}_{c_\text{s}} = \ell/c_\text{s}$ is typically shorter than the *advective* time scale $\mathcal{T}_u = \ell/u$ where u is a characteristic fluid velocity. Indeed, it can readily be found that $\mathcal{T}_{c_\text{s}}/\mathcal{T}_u = u/c_\text{s} = \text{Ma}$. Therefore, $\mathcal{T}_{\text{mfp}}/\mathcal{T}_{c_\text{s}} = O(\text{Kn})$ and $\mathcal{T}_{\text{mfp}}/\mathcal{T}_u = O(\text{Kn} \times \text{Ma})$; these two ratios scale at the same order in the Knudsen number.

With this said, we can now take another look at the terms in (4.5) and examine their order in the Knudsen number, allowing the characteristic time scale to be either advective or acoustic:

$$\begin{aligned}
\text{Advective:} \qquad & O(\Delta t \partial_t f_i) \sim O(\tau/\mathcal{T}_u) \sim O(\text{Ma}\,\mathcal{T}_{\text{mfp}}/\mathcal{T}_{c_\text{s}}) \sim O(\text{Ma} \times \text{Kn}) \\
\text{Acoustic:} \qquad & O(\Delta t \partial_t f_i) \sim O(\tau/\mathcal{T}_{c_\text{s}}) \sim O(\mathcal{T}_{\text{mfp}}/\mathcal{T}_{c_\text{s}}) \sim O(\text{Kn}) \\
& O(\Delta t c_{i\alpha} \partial_\alpha f_i) \sim O(\tau c_\text{s}/\ell) \sim O(\mathcal{T}_{\text{mfp}}/\mathcal{T}_{c_\text{s}}) \sim O(\text{Kn})
\end{aligned} \tag{A.9}$$

Consequently, we have that $\Delta t^n(\partial_t + c_{i\alpha}\partial_\alpha)^n f_i$ scales with Kn^n. Neglecting third- and higher-order terms in (4.5) is therefore consistent with our ansatz of keeping only terms of the two lowest orders in Kn.

A.2.2 The Moment Perturbation

To be able to find the macroscopic momentum equation through the Chapman-Enskog analysis in Sect. 4.1, we must determine the moment $\Pi^{(1)}_{\alpha\beta}$. This can be found from (4.10c) as

$$\Pi^{(1)}_{\alpha\beta} = -\tau \left(\partial^{(1)}_t \Pi^{\text{eq}}_{\alpha\beta} + \partial^{(1)}_\gamma \Pi^{\text{eq}}_{\alpha\beta\gamma} \right). \tag{A.10}$$

From this we would like to find an explicit expression for $\Pi^{(1)}_{\alpha\beta}$ through the macroscopic quantities ρ and $\boldsymbol{u}$ and their derivatives.

We already know the two equilibrium moments in (A.10) explicitly:

$$\Pi^{\text{eq}}_{\alpha\beta} = \rho u_\alpha u_\beta + \rho c_\text{s}^2 \delta_{\alpha\beta}, \quad \Pi^{\text{eq}}_{\alpha\beta\gamma} = \rho c_\text{s}^2 \left(u_\alpha \delta_{\beta\gamma} + u_\beta \delta_{\alpha\gamma} + u_\gamma \delta_{\alpha\beta} \right). \tag{A.11}$$

Since we would like the resulting momentum equation to be similar to the Euler and Navier-Stokes equations, all time derivatives in $\Pi^{(1)}_{\alpha\beta}$ should be eliminated. We can

[1] We show in Sect. 12.1.1 that the quantity of *viscous relaxation time* τ_{vi} is $O(\tau)$, and it is shown elsewhere [2] that $\tau_{\text{vi}} = O(\mathcal{T}_{\text{mfp}})$.

do this by rewriting (4.10) more explicitly as

$$\partial_t^{(1)}\rho = -\partial_\alpha^{(1)}\left(\rho u_\alpha\right), \quad \partial_t^{(1)}(\rho u_\alpha) = -\partial_\beta^{(1)}\left(\rho u_\alpha u_\beta + \rho c_\mathrm{s}^2\delta_{\alpha\beta}\right). \tag{A.12}$$

We also need to make use of a corollary of the product rule; if ∂_* is a generic derivative and a, b, and c are generic variables, then

$$\partial_*(abc) = a\partial_*(bc) + b\partial_*(ac) - ab\partial_* c. \tag{A.13}$$

The following derivation is simplified by our use of the isothermal equation of state $p = c_\mathrm{s}^2\rho$ where the pressure and density are linearly related through the constant c_s^2. In other cases, we would have to treat the pressure in a more complicated fashion. For monatomic gases, for example, we would need to express pressure changes using the conservation equation for translational energy [3].

We will now resolve the two equilibrium moment derivatives in (A.10) separately, starting with the one which is the simplest to resolve:

$$\begin{aligned}\partial_\gamma^{(1)}\Pi_{\alpha\beta\gamma}^{\mathrm{eq}} &= \partial_\gamma^{(1)}\left(\rho c_\mathrm{s}^2\left[u_\alpha\delta_{\beta\gamma} + u_\beta\delta_{\alpha\gamma} + u_\gamma\delta_{\alpha\beta}\right]\right)\\ &= c_\mathrm{s}^2\left(\partial_\beta^{(1)}\rho u_\alpha + \partial_\alpha^{(1)}\rho u_\beta\right) + c_\mathrm{s}^2\delta_{\alpha\beta}\partial_\gamma^{(1)}(\rho u_\gamma).\end{aligned} \tag{A.14}$$

The other equilibrium moment derivative is more complicated, and we will resolve it in steps. First of all, we apply (A.13) and find

$$\begin{aligned}\partial_t^{(1)}\Pi_{\alpha\beta}^{\mathrm{eq}} &= \partial_t^{(1)}(\rho u_\alpha u_\beta + \rho c_\mathrm{s}^2\delta_{\alpha\beta})\\ &= u_\alpha\partial_t^{(1)}(\rho u_\beta) + u_\beta\partial_t^{(1)}(\rho u_\alpha) - u_\alpha u_\beta\partial_t^{(1)}\rho + c_\mathrm{s}^2\delta_{\alpha\beta}\partial_t^{(1)}\rho.\end{aligned} \tag{A.15a}$$

Then we apply (A.12) to replace the time derivatives and subsequently rearrange:

$$\begin{aligned}\partial_t^{(1)}\Pi_{\alpha\beta}^{\mathrm{eq}} &= -u_\alpha\partial_\gamma^{(1)}\left(\rho u_\beta u_\gamma + \rho c_\mathrm{s}^2\delta_{\beta\gamma}\right) - u_\beta\partial_\gamma^{(1)}\left(\rho u_\alpha u_\gamma + \rho c_\mathrm{s}^2\delta_{\alpha\gamma}\right)\\ &\quad + u_\alpha u_\beta\partial_\gamma^{(1)}(\rho u_\gamma) - c_\mathrm{s}^2\delta_{\alpha\beta}\partial_\gamma^{(1)}(\rho u_\gamma)\\ &= -\left[u_\alpha\partial_\gamma^{(1)}(\rho u_\beta u_\gamma) + u_\beta\partial_\gamma^{(1)}(\rho u_\alpha u_\gamma) - u_\alpha u_\beta\partial_\gamma^{(1)}(\rho u_\gamma)\right]\\ &\quad - c_\mathrm{s}^2\left(u_\alpha\partial_\beta^{(1)}\rho + u_\beta\partial_\alpha^{(1)}\rho\right) - c_\mathrm{s}^2\delta_{\alpha\beta}\partial_\gamma^{(1)}(\rho u_\gamma).\end{aligned} \tag{A.15b}$$

Finally, the bracketed terms can be simplified by using (A.12) in reverse, giving

$$\partial_t^{(1)}\Pi_{\alpha\beta}^{\mathrm{eq}} = -\partial_\gamma^{(1)}\left(\rho u_\alpha u_\beta u_\gamma\right) - c_\mathrm{s}^2\left(u_\alpha\partial_\beta^{(1)}\rho + u_\beta\partial_\alpha^{(1)}\rho\right) - c_\mathrm{s}^2\delta_{\alpha\beta}\partial_\gamma^{(1)}(\rho u_\gamma). \tag{A.15c}$$

Now that we have explicit forms of the two equilibrium moment derivative terms in (A.14) and (A.15c), we insert them into (A.10). After using the product rule and having some terms cancel, we end up with the explicit expression

$$\Pi^{(1)}_{\alpha\beta} = -\tau \left[\rho c_s^2 \left(\partial^{(1)}_\beta u_\alpha + \partial^{(1)}_\alpha u_\beta \right) - \partial^{(1)}_\gamma \left(\rho u_\alpha u_\beta u_\gamma \right) \right]. \tag{A.16}$$

The last term is an error term: it would have been entirely cancelled if $\Pi^{\mathrm{eq}}_{\alpha\beta\gamma}$ in (A.11) had contained the $\rho u_\alpha u_\beta u_\gamma$ term which it includes in the exact kinetic theory. The reason why this term is missing is that we have truncated the equilibrium distribution f_i^{eq} to $O(u^2)$. This truncation allows using smaller velocity sets like D2Q9, D3Q15, D3Q19, and D3Q27 without any undesirable anisotropy (cf. Sect. 4.2.1).

In other words, removing the $O(u^3)$ error term in (A.16) would require an extended lattice. This would slow down computations and make boundary conditions more difficult to deal with. However, recent work suggests that this error term can also be nearly cancelled by using a modified collision operator where τ depends on $\boldsymbol{u}$ [4].

A.2.3 Chapman-Enskog Analysis for the MRT Collision Operator

The Chapman-Enskog analysis in Sect. 4.1 assumes the use of the BGK collision operator. However, it is not that much more difficult to perform the analysis for the general multiple-relaxation-time (MRT) collision operator described in Chap. 10. We will here show how its Chapman-Enskog analysis differs from that in Sect. 4.1. (We will not give a full analysis here, only highlight the differences to the one given previously.) While this description will be mainly tied to the D2Q9 MRT collision operators presented in Sect. 10.4, we can use this approach for any MRT-based collision operator. At the end of this section we will point out the minor differences arising for the D3Q15 and D3Q19 results described in Sect. A.6.

The fundamental difference to the BGK analysis is that MRT collision operators can have different moments. The analysis relies on the three collision operator moments

$$\sum_i \Omega_i = 0, \quad \sum_i c_{i\alpha} \Omega_i = 0, \quad \sum_i c_{i\alpha} c_{i\beta} \Omega_i. \tag{A.17}$$

Of these moments, the first two are zero due to mass and momentum conservation in collisions. For the BGK collision operator, the third moment becomes $-(\Delta t/\tau)\Pi^{\mathrm{neq}}_{\alpha\beta}$, with $\Pi_{\alpha\beta} = \sum_i c_{i\alpha} c_{i\beta} f_i$. As we shall see, the results of the corresponding MRT analysis hinge on the differences in this moment.

We can find the second moment by left-multiplying an MRT collision operator $\boldsymbol{\Omega} = -\boldsymbol{M}^{-1}\boldsymbol{S}\boldsymbol{M}\left(\boldsymbol{f} - \boldsymbol{f}^{\mathrm{eq}}\right)$ individually with the row vectors $\boldsymbol{M}_{\Pi_{xx}}$, $\boldsymbol{M}_{\Pi_{yy}}$ and $\boldsymbol{M}_{\Pi_{xy}}$

where $M_{\Pi_{\alpha\beta},i} = c_{i\alpha}c_{i\beta}$. (While it is feasible to find $\boldsymbol{SM}(\boldsymbol{f}-\boldsymbol{f}^{\mathrm{eq}})$ analytically, $\boldsymbol{M}_{\Pi_{xx}}\boldsymbol{M}^{-1}$ *etc.* are best computed numerically.)

Thus, using the Hermite polynomial-based MRT approach from Sect. 10.4.1, we find after some algebra that

$$\sum_i c_{i\alpha}c_{i\beta}\Omega_i = -\omega_\nu \Pi^{\mathrm{neq}}_{\alpha\beta} - \frac{\omega_\zeta - \omega_\nu}{2}\delta_{\alpha\beta}\Pi^{\mathrm{neq}}_{\gamma\gamma}. \tag{A.18}$$

Thus, only the relaxation rates ω_ν and ω_ζ affect the macroscopic momentum equation at the Navier-Stokes level. If $\omega_\nu = \omega_\zeta$, this equation is equivalent with that of the BGK collision operator with $\omega_\nu = 1/\tau$.

Using the Gram-Schmidt approach in Sect. 10.4.2 instead, we find the same result as in (A.18) with $\omega_\zeta \to \omega_e$. Except for this tiny change in notation, the Chapman-Enskog procedure for the Hermite and Gram-Schmidt approaches are therefore identical.

Now, let us look at how the analysis itself differs from the previous BGK analysis. Using a generic collision operator Ω_i, the LBE after Taylor expansion and some algebra becomes

$$\Delta t\,(\partial_t + c_{i\alpha}\partial_\alpha) f_i = \Omega_i - \Delta t\,(\partial_t + c_{i\alpha}\partial_\alpha)\frac{\Delta t}{2}\Omega_i \tag{A.19}$$

instead of (4.7). Expanding f_i and the derivatives, the different moments at different orders in ϵ of this equation become as in (4.10) and (4.12), except that two equations have a few extra terms stemming from the $\Pi_{\alpha\beta}$ moment in (A.18):

$$\partial_t^{(1)}\Pi^{\mathrm{eq}}_{\alpha\beta} + \partial_\gamma^{(1)}\Pi^{\mathrm{eq}}_{\alpha\beta\gamma} = -\omega_\nu\Pi^{(1)}_{\alpha\beta} - \frac{\omega_\zeta-\omega_\nu}{2}\delta_{\alpha\beta}\Pi^{(1)}_{\gamma\gamma}, \tag{A.20a}$$

$$\partial_t^{(2)}(\rho u_\alpha) = -\partial_\beta\left[\left(1-\frac{\omega_\nu\Delta t}{2}\right)\Pi^{(1)}_{\alpha\beta} - \frac{(\omega_\zeta-\omega_\nu)\Delta t}{4}\delta_{\alpha\beta}\Pi^{(1)}_{\gamma\gamma}\right]. \tag{A.20b}$$

Using the same procedure as in Sect. A.2.2, we can find from (A.20a) that

$$\Pi^{(1)}_{\alpha\beta} = -\frac{\rho c_{\mathrm{s}}^2}{\omega_\nu}\left(\partial_\alpha u_\beta + \partial_\beta u_\alpha\right) - \frac{1}{2}\left(\frac{\omega_\zeta}{\omega_\nu} - 1\right)\delta_{\alpha\beta}\Pi^{(1)}_{\gamma\gamma}, \tag{A.21}$$

having neglected the $O(u^3)$ errors. We can make this more explicit by multiplying with $\delta_{\alpha\beta}$. As we are using the two-dimensional D2Q9 velocity set, $\delta_{\alpha\beta}\delta_{\alpha\beta} = \delta_{\gamma\gamma} = 2$, and after some rearranging we find

$$\Pi^{(1)}_{\gamma\gamma} = -\frac{2\rho c_{\mathrm{s}}^2}{\omega_\zeta}\partial_\gamma u_\gamma. \tag{A.22}$$

When re-assembling the different orders in ϵ of the momentum equation, we find that the resulting viscous stress tensor $\sigma'_{\alpha\beta}$ is given by the right-hand side of (A.20b). After some algebra we find

$$\begin{aligned}\sigma'_{\alpha\beta} &= -\left(1-\frac{\omega_\nu \Delta t}{2}\right)\Pi^{(1)}_{\alpha\beta} + \frac{(\omega_\zeta-\omega_\nu)\Delta t}{4}\delta_{\alpha\beta}\Pi^{(1)}_{\gamma\gamma} \\ &= \eta\left(\partial_\alpha u_\beta + \partial_\beta u_\alpha - \frac{2}{3}\delta_{\alpha\beta}\partial_\gamma u_\gamma\right) + \eta_{\mathrm{B}}\delta_{\alpha\beta}\partial_\gamma u_\gamma\end{aligned} \tag{A.23}$$

with the dynamic shear and bulk viscosities

$$\eta = \rho c_{\mathrm{s}}^2\left(\frac{1}{\omega_\nu}-\frac{\Delta t}{2}\right), \quad \eta_{\mathrm{B}} = \rho c_{\mathrm{s}}^2\left(\frac{1}{\omega_\zeta}-\frac{\Delta t}{2}\right) - \frac{\eta}{3}. \tag{A.24}$$

This result is valid for the Hermite and Gram-Schmidt D2Q9 MRT of Sect. 10.4, with $\omega_\zeta \to \omega_e$ in the Gram-Schmidt approach. We can easily confirm this by simulating a free sound wave as described in Sect. 12.1.3 and verifying that its amplitude decay varies with η and η_{B} as predicted. (Note that the amplitude may ripple around the expected exponential decay if the wave is not initialised perfectly [5].)

For the D3Q15 and D3Q19 MRT described in Sect. A.6, the analysis is the same apart from two minor differences. First, the D3Q15 and D3Q19 moments are both like in (A.18), except with a different coefficient $(\omega_e - \omega_\nu)/3$ in the last term. Additionally, $\delta_{\gamma\gamma} = 3$ instead of 2. These differences end up changing the bulk viscosity slightly, so that

$$\eta = \rho c_{\mathrm{s}}^2\left(\frac{1}{\omega_\nu}-\frac{\Delta t}{2}\right), \quad \eta_{\mathrm{B}} = \frac{2}{3}\rho c_{\mathrm{s}}^2\left(\frac{1}{\omega_e}-\frac{\Delta t}{2}\right). \tag{A.25}$$

This result can be verified in the same way as the D2Q9 result.

A.3 Taylor-Green Vortex Flow

The decaying *Taylor-Green vortex flow*, shown in Fig. A.1, solves the incompressible Navier-Stokes equations, (1.18). As this flow is known analytically, it is often used as a benchmark test for Navier-Stokes solvers.

The Taylor-Green flow is unsteady and fully periodic in a domain of size $\ell_x \times \ell_y$. Formulated in two spatial dimensions its velocity and pressure fields read

$$\boldsymbol{u}(\boldsymbol{x},t) = u_0\begin{pmatrix} -\sqrt{k_y/k_x}\cos(k_x x)\sin(k_y y) \\ \sqrt{k_x/k_y}\sin(k_x x)\cos(k_y y)\end{pmatrix}\mathrm{e}^{-t/t_{\mathrm{d}}} \tag{A.26}$$

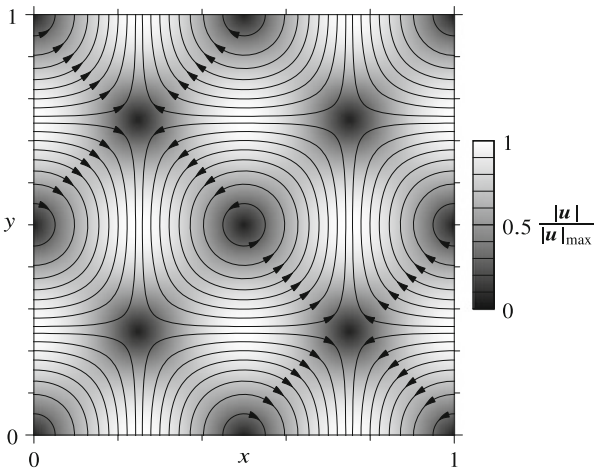

Fig. A.1 Structure of a Taylor-Green vortex flow with $k_x = k_y = 2\pi$ in $[0, 1] \times [0, 1]$. The flow maintains the same structure while decaying exponentially

and

$$p(\boldsymbol{x}, t) = p_0 - \rho \frac{u_0^2}{4} \left[\frac{k_y}{k_x} \cos(2k_x x) + \frac{k_x}{k_y} \cos(2k_y y) \right] \mathrm{e}^{-2t/t_\mathrm{d}}. \tag{A.27}$$

Here, u_0 is the initial velocity scale, $k_{x,y} = 2\pi/\ell_{x,y}$ are the components of the wave vector $\boldsymbol{k}$ and

$$t_\mathrm{d} = \frac{1}{\nu(k_x^2 + k_y^2)} \tag{A.28}$$

is the vortex decay time. The pressure average p_0 is arbitrary and does not enter the Navier-Stokes equations. The initial state is defined by $\boldsymbol{u}(\boldsymbol{x}, 0)$ and $p(\boldsymbol{x}, 0)$.

Exercise A.1 Show that the velocity field in (A.26) leads to a deviatoric stress tensor with components

$$\begin{aligned}
\sigma_{xx} &= 2\rho\nu u_0 \sqrt{k_x k_y} \sin(k_x x) \sin(k_y y)\, \mathrm{e}^{-t/t_\mathrm{d}}, \\
\sigma_{xy} &= \rho\nu u_0 \left(\sqrt{k_x^3/k_y} - \sqrt{k_y^3/k_x} \right) \cos(k_x x) \cos(k_y y)\, \mathrm{e}^{-t/t_\mathrm{d}}, \\
\sigma_{yx} &= \sigma_{xy}, \\
\sigma_{yy} &= -\sigma_{xx}.
\end{aligned} \tag{A.29}$$

In particular this means that the stress tensor is symmetric and traceless. Furthermore, σ_{xy} and σ_{yx} vanish if $k_x = k_y$, i.e. if $\ell_x = \ell_y$. This means that $\ell_x \neq \ell_y$ should be chosen if one wants to investigate the accuracy of the off-diagonal stress tensor components.

Exercise A.2 Show that the velocity and pressure in (A.26) and (A.27) solve the incompressible Navier-Stokes equations. Which pairs of terms cancel each other?

A.4 Gauss-Hermite Quadrature

One of the most useful features of Hermite polynomials for numerical integration is the Gauss-Hermite quadrature rule [6]: the integral of any 1D function $f(x)$ multiplied by the weight function $\omega(x)$ (cf. (3.22)) can be approximated by a finite series of function values in certain points x_i, also called *abscissae*:

$$\int_{-\infty}^{\infty} \omega(x) f(x)\,\mathrm{d}x \approx \sum_{i=1}^{q} w_i f(x_i). \tag{A.30}$$

The accuracy of the integration depends on the values and number q of point values x_i. If one chooses x_i as the n roots of the Hermite polynomials of order n, i.e. $H^{(n)}(x_i) = 0$ and $q = n$, then it is guaranteed that any polynomial $P^{(N)}(x)$ of order $N = 2n - 1$ can be integrated exactly:

$$\int_{-\infty}^{\infty} \omega(x) P^{(N)}(x)\,\mathrm{d}x = \sum_{i=1}^{n} w_i P^{(N)}(x_i). \tag{A.31}$$

The weights can be found as [7]

$$w_i = \frac{n!}{\left(nH^{(n-1)}(x_i)\right)^2}. \tag{A.32}$$

The abscissae and weigths required to integrate polynomials up to fifth order ($N = 5$) are shown in Table A.2.

Example A.1 To integrate a third-order polynomial $P^{(3)}(x)$, one needs $n = 2$, and therefore the polynomial $H^{(2)}(x)$ with two abscissae points at ± 1 (cf. Table A.2).

Table A.2 Abscissae x_i and weights w_i for exact integration of polynomials up to fifth order

Number of abscissae	Polynomial degree	Abscissae	Weights
n	$N = 2n - 1$	x_i	w_i
1	1	0	1
2	3	± 1	1/2
3	5	0	2/3
		$\pm\sqrt{3}$	1/6

Together with the weights $w_{1,2} = 1/2$ for $H^{(2)}$, one obtains

$$\int_{-\infty}^{\infty} \omega(x) P^{(3)}(x)\,\mathrm{d}x = \frac{1}{2}P^{(3)}(+1) + \frac{1}{2}P^{(3)}(-1). \tag{A.33}$$

Those abscissae allow us to obtain the most common lattices for the LBM as we will describe below. More elaborate examples and advanced lattices can be found in the seminal work of Shan et al. [8] or in [9].

The extension to multiple dimensions is straightforward. Any real polynomial of order N in d-dimensional space can be written in the form

$$P^{(N)}(\boldsymbol{x}) = \sum_{N_1+\ldots+N_d \leq N} a_{N_1\ldots N_d} x_1^{N_1} \cdots x_d^{N_d} \tag{A.34}$$

where the $a_{N_1\ldots N_d}$ are real coefficients and $\{N_1, \ldots, N_d\}$ are integers. A well-known example of a second-order polynomial in two dimensions is $P^{(2)}(\boldsymbol{x}) = \boldsymbol{x}\cdot\boldsymbol{x} = x_1^2 + x_2^2 + x_3^2$. Here, the mixed coefficients a_{12}, a_{23} etc. all vanish and $a_{11} = a_{22} = a_{33} = 1$. The integral of such a polynomial multiplied by the multidimensional weight function $\omega(\boldsymbol{x})$ can be written as sum of integrals with 1D weight functions $\omega(x)$:

$$\begin{aligned}\int \omega(\boldsymbol{x}) P^{(N)}(\boldsymbol{x})\,\mathrm{d}^d x &= \int \omega(\boldsymbol{x}) \sum a_{N_1\ldots N_d} x_1^{N_1}\cdots x_d^{N_d}\,\mathrm{d}^d x \\ &= \sum a_{N_1\ldots N_d} \prod_{j=1}^{d} \int \omega(x_j) x_j^{N_j}\,\mathrm{d}x_j,\end{aligned} \tag{A.35}$$

where we have used $\omega(\boldsymbol{x}) = \prod_{j=1}^{d} \omega(x_j)$. Each of the 1D integrals can now be decomposed using the Gauss-Hermite quadrature rule from (A.31):

$$\sum a_{N_1\ldots N_d} \prod_{j=1}^{d} \int \omega(x_j) x_j^{N_j}\,\mathrm{d}x_j = \sum a_{N_1\ldots N_d} \prod_{j=1}^{d} \sum_{i=1}^{n_j} w_{i,j} x_{i,j}^{N_j}. \tag{A.36}$$

Here, $x_{i,j}$ is the j-component of the i-th abscissa.

Let us assume that all 1D integrals are discretised using the same Hermite polynomial, i.e. $n_1 = \ldots = n_d = n$, $x_{i,1} = \ldots = x_{i,d} = x_i$ and $w_{i,1} = \ldots = w_{i,d} = w_i$. In this case, we can rewrite the product of sums:

$$\prod_{j=1}^{d} \sum_{i=1}^{n_j} w_{i,j} x_{i,j}^{N_j} = \sum_{i_1=1}^{n} \cdots \sum_{i_d=1}^{n} w_{i_1}\cdots w_{i_d} x_{i_1}^{N_1}\cdots x_{i_d}^{N_d}. \tag{A.37}$$

Introducing new multi-dimensional abscissae $\boldsymbol{x}_i = (x_{i_1}, \ldots, x_{i_D})$ and weights $w_i = w_{i_1} \cdots w_{i_D}$ allows us to obtain the multi-dimensional Gauss-Hermite quadrature rule:

$$\int \omega(\boldsymbol{x}) P^{(N)}(\boldsymbol{x}) \, \mathrm{d}^d x = \sum_{i=1}^{n^d} w_i P^{(N)}(\boldsymbol{x}_i). \tag{A.38}$$

Example A.2 We demonstrate the multi-dimensional Gauss-Hermite quadrature rule by integrating the polynomial $P^{(3)}(\boldsymbol{x}) = x^2 y$ in 2D (we write x and y instead of x_1 and x_2). Since $N = 3$, we need two abscissae ($n = 2$) for each dimension, i.e. $n^d = 2^2 = 4$ in total. First we note that, for a 1D polynomial of third order, we get

$$\int w(x) x^N \, \mathrm{d}x = w_1 x_1^N + w_2 x_2^N = \frac{1}{2}(-1)^N + \frac{1}{2} 1^N \tag{A.39}$$

for $N \leq 3$. In the last step we have used the known weights and abscissae from Table A.2. It follows that

$$\begin{aligned} \int \frac{1}{2\pi} \mathrm{e}^{-(x^2+y^2)/2} x^2 y \, \mathrm{d}x \, \mathrm{d}y &= \int \frac{1}{\sqrt{2\pi}} \mathrm{e}^{-x^2/2} x^2 \, \mathrm{d}x \int \frac{1}{\sqrt{2\pi}} \mathrm{e}^{-y^2/2} y \, \mathrm{d}y \\ &= \left(w_1 x_1^2 + w_2 x_2^2 \right) \left(w_1 y_1 + w_2 y_2 \right). \end{aligned} \tag{A.40}$$

Using $w_1 = w_2 = \frac{1}{2}$, $x_1 = y_1 = -1$ and $x_2 = y_2 = 1$, it is straightforward to show that the result is zero.

We have seen in Sect. 3.4 that one needs to integrate fifth-order polynomials in order to obtain Navier-Stokes behaviour. This implies $N = 5$ and $n = 3$. From Table A.2 we find that $n = 3$ leads to the abscissae 0 and $\pm\sqrt{3}$. After rescaling the velocities to get rid of the factor $\sqrt{3}$ (cf. Sect. 3.4.5) one obtains the D1Q3 lattice in Table 3.2.

It is now straightforward to build the corresponding 2D and 3D lattices *via* (A.37). In 2D and 3D we require $q = n^d = 3^2 = 9$ and $q = n^d = 3^3 = 27$ abscissae, respectively. As a result, we obtain the D2Q9 (cf. Table 3.3) and the D3Q27 (cf. Table 3.6) lattices [10].

Based on symmetry considerations, one can construct other lattices than D2Q9 and D3Q27. First of all, any integral of the form in (A.31) vanishes if the polynomial $P^{(N)}(x)$ contains only odd orders, e.g. $P^{(5)}(x) = 2x^5 - x^3 + x$. This can be generalised: if all monomials of a multi-dimensional polynomial contain at least one odd-order term, e.g. xy^2z^2 (which is odd in x) or $x^5y^3z^2$ (which is odd in x and y), the integral vanishes. This means that we can directly abandon all of those monomials since they do not contribute to the integral anyway. As a result, we keep only monomials of the form $x^{2a}y^{2b}z^{2c}$ where a, b and c are non-negative integers. Examples are x^2y^2 ($a = 1$, $b = 1$, $c = 0$) or $x^2y^4z^2$ ($a = 1$, $b = 2$, $c = 1$). Additionally, if our scope is limited to Navier-Stokes simulations we have to care about polynomials up to the fifth order only. The lowest-order even monomial containing x, y and z, however, is

Table A.3 Abscissae and weights for exact integration of 2D polynomials of up to fifth order

Number of abscissae	Abscissae	Weights	
q	x_i	w_i	
7	$(0,0)$	$1/2$	
	$2\left(\cos\frac{m\pi}{3}, \sin\frac{m\pi}{3}\right)$	$1/12$	$m = 1,\ldots,6$
9	$(0,0)$	$4/9$	
	$\left(0, \pm\sqrt{3}\right), \left(\pm\sqrt{3}, 0\right)$	$1/9$	
	$\left(\pm\sqrt{3}, \pm\sqrt{3}\right)$	$1/36$	

$x^2y^2z^2$ and therefore already of sixth order. It is therefore possible to devise a lattice without the $(\pm1, \pm1, \pm1)$-velocities: D3Q19 in Table 3.5.

We are able to obtain different lattices with even fewer abscissae with other methods than symmetry arguments. For example, by using the abscissae of integrals with another weight function, one can construct D2Q7 (which is not on a square but on a hexagonal lattice) and D3Q13. These are the smallest possible sets in 2D and 3D that can still be used to solve the NSE. We skip the mathematical details and refer to [8, 9] instead.

Tables A.3 and A.4 contain the **multi-dimensional abscissae and weights** for the most common discretisations in 2D and 3D. All of these are **suitable for LB simulations of the NSE** after an appropriate renormalisation to obtain integer lattice velocity components (cf. Sect. 3.4.7).

A.5 Integration Along Characteristics for the BGK Operator

In Sect. 3.5.1, we presented a rather general scheme for the integration along characteristics where the collision operator was not yet specified. Here we will take a closer look ath the integration along characteristics with the BGK operator.

With the BGK collision operator it is possible to partially solve the continuous Boltzmann equation of the form

$$\frac{\partial f_i}{\partial t} + c_{i\alpha}\frac{\partial f_i}{\partial x_\alpha} = -\frac{f_i - f_i^{\mathrm{eq}}}{\tau}. \tag{A.41}$$

Table A.4 Abscissae and weights for exact integration of 3D polynomials of up to fifth order

Number of abscissae	Abscissae	Weights	
q	x_i	w_i	
13	$(0,0,0)$	$2/5$	
	$(\pm r, \pm s, 0)$	$1/20$	$r^2 = (5+\sqrt{5})/2$
	$(0, \pm r, \pm s)$	$1/20$	$s^2 = (5-\sqrt{5})/2$
	$(\pm s, 0, \pm r)$	$1/20$	
15	$(0,0,0)$	$2/9$	
	$(\pm\sqrt{3},0,0)$, $(0,\pm\sqrt{3},0)$, $(0,0,\pm\sqrt{3})$	$1/9$	
	$(\pm\sqrt{3},\pm\sqrt{3},\pm\sqrt{3})$	$1/72$	
19	$(0,0,0)$	$1/3$	
	$(\pm\sqrt{3},0,0)$, $(0,\pm\sqrt{3},0)$, $(0,0,\pm\sqrt{3})$	$1/18$	
	$(\pm\sqrt{3},\pm\sqrt{3},0)$, $(\pm\sqrt{3},0,\pm\sqrt{3})$, $(0,\pm\sqrt{3},\pm\sqrt{3})$	$1/36$	
27	$(0,0,0)$	$8/27$	
	$(\pm\sqrt{3},0,0)$, $(0,\pm\sqrt{3},0)$, $(0,0,\pm\sqrt{3})$	$2/27$	
	$(\pm\sqrt{3},\pm\sqrt{3},0)$, $(\pm\sqrt{3},0,\pm\sqrt{3})$, $(0,\pm\sqrt{3},\pm\sqrt{3})$	$1/54$	
	$(\pm\sqrt{3},\pm\sqrt{3},\pm\sqrt{3})$	$1/216$	

To find the solution for f_i, we need to solve the following system in terms of the newly introduced variable ζ:

$$\frac{\mathrm{d}f_i}{\mathrm{d}\zeta} = \frac{\partial f_i}{\partial t}\frac{\mathrm{d}t}{\mathrm{d}\zeta} + \frac{\partial f_i}{\partial x_\alpha}\frac{\mathrm{d}x_\alpha}{\mathrm{d}\zeta} = -\frac{f_i(\zeta) - f_i^{\mathrm{eq}}(\zeta)}{\tau} \tag{A.42}$$

with

$$\frac{\mathrm{d}t}{\mathrm{d}\zeta} = 1, \quad \frac{\mathrm{d}x_\alpha}{\mathrm{d}\zeta} = c_{i\alpha}. \tag{A.43}$$

The dependence of f_i^{eq} on ζ enters through $f_i^{\mathrm{eq}}(\rho(\zeta), \boldsymbol{u}(\zeta))$ with the parametrisations $\rho(\zeta) = \rho(\boldsymbol{x}(\zeta), t(\zeta))$ and $\boldsymbol{u}(\zeta) = \boldsymbol{u}(\boldsymbol{x}(\zeta), t(\zeta))$.

We can easily integrate equation (A.43) to obtain the characteristics equation:

$$t = \zeta + t_0, \quad \boldsymbol{x} = \boldsymbol{c}_i\zeta + \boldsymbol{x}_0 \tag{A.44}$$

with $t_0 = t(\zeta = 0)$ and $\boldsymbol{x}_0 = \boldsymbol{x}(\zeta = 0)$.

To integrate equation (A.42), we consider the ODE

$$\frac{\mathrm{d}y(\zeta)}{\mathrm{d}\zeta} = g(\zeta)y(\zeta) + h(\zeta) \tag{A.45}$$

for $y(\zeta)$ with given coefficient functions $g(\zeta)$ and $h(\zeta)$. The solution can be obtained following the well-known *variation of constants*:

$$y(\zeta) = \mathrm{e}^{G(\zeta)} \left[C + \int_{\zeta_0}^{\zeta} \mathrm{e}^{-G(\zeta')} h(\zeta') \, \mathrm{d}\zeta' \right] \tag{A.46}$$

with

$$G(\zeta) = \int_{\zeta_0}^{\zeta} g(\zeta') \, \mathrm{d}\zeta' \tag{A.47}$$

and integration constants C and ζ_0.

Exercise A.3 Show that (A.46) solves (A.45).

We can recast (A.42) into the form of (A.45):

$$\frac{\mathrm{d}f_i(\zeta)}{\mathrm{d}\zeta} = -\frac{1}{\tau} f_i(\zeta) + \frac{f_i^{\mathrm{eq}}}{\tau}. \tag{A.48}$$

Now we identify $f_i(\zeta)$ with $y(\zeta)$, $-1/\tau$ with $g(\zeta)$ and $f_i^{\mathrm{eq}}(\zeta)/\tau$ with $h(\zeta)$. This leads to $G(\zeta) = -(\zeta - \zeta_0)/\tau$.

Using the integration limits ζ_0 and $\zeta = \zeta_0 + \Delta t$, i.e. integrating over one time step, we first obtain

$$f_i(\zeta_0 + \Delta t) = \mathrm{e}^{-\Delta t/\tau} \left[C + \frac{1}{\tau} \int_{\zeta_0}^{\zeta_0 + \Delta t} \mathrm{e}^{\zeta'/\tau} f_i^{\mathrm{eq}}(\zeta') \, \mathrm{d}\zeta' \right]. \tag{A.49}$$

We write $C = f_i(\zeta_0)$ and replace the ζ-dependence by introducing $\boldsymbol{x}$ and t again. Furthermore, we drop the index 0 from $\boldsymbol{x}_0$ and t_0 as these integration constants can be chosen arbitrarily:

$$f_i(\boldsymbol{x} + \boldsymbol{c}_i \Delta t, t + \Delta t) = \mathrm{e}^{-\Delta t/\tau} \left[f_i(\boldsymbol{x}, t) + \frac{1}{\tau} \int_{t}^{t + \Delta t} \mathrm{e}^{(t' - t)/\tau} f_i^{\mathrm{eq}}(\boldsymbol{x} + \boldsymbol{c}_i (t' - t), t') \, \mathrm{d}t' \right]. \tag{A.50}$$

This is the integral-form solution of the LBGK equation.

Now we want to discretise the integral in equation (A.50) to make it applicable in computer simulations. We have several options to achieve this discretisation, for instance forward Euler (first-order accurate) or the trapezoidal rule (second-order accurate).

If we choose a first-order approximation, we replace an integral of the form $\int_t^{t+\Delta t} g(t')\,\mathrm{d}t'$ by $g(t)\Delta t$. This results in

$$f_i(\boldsymbol{x}+\boldsymbol{c}_i\Delta t, t+\Delta t) = \mathrm{e}^{-\Delta t/\tau} f_i(\boldsymbol{x},t) + \frac{\mathrm{e}^{-\Delta t/\tau}}{\tau} f_i^{\mathrm{eq}}(\boldsymbol{x},t)\Delta t. \tag{A.51}$$

Expanding the exponentials and keeping only terms up to first order in Δt gives

$$f_i(\boldsymbol{x}+\boldsymbol{c}_i\Delta t, t+\Delta t) = \left(1-\frac{\Delta t}{\tau}\right) f_i(\boldsymbol{x},t) + \frac{\Delta t}{\tau} f_i^{\mathrm{eq}}(\boldsymbol{x},t) + O(\Delta t^2). \tag{A.52}$$

This is exactly the standard discretised LBGK equation, but it is only first-order accurate in time.

To achieve a second-order discretisation, we can approximate the integral with the trapezoidal rule: $\int_t^{t+\Delta t} g(t')\,\mathrm{d}t' \approx [g(t)+g(t+\Delta t)]\Delta t/2$. Therefore we have

$$\begin{aligned} f_i(\boldsymbol{x}+\boldsymbol{c}_i\Delta t, t+\Delta t) = {} & \mathrm{e}^{-\Delta t/\tau} f_i(\boldsymbol{x},t) \\ & + \frac{\mathrm{e}^{-\Delta t/\tau}\Delta t}{2\tau}\left(\mathrm{e}^{\Delta t/\tau} f_i^{\mathrm{eq}}(\boldsymbol{x}+\boldsymbol{c}_i\Delta t, t+\Delta t) + f_i^{\mathrm{eq}}(\boldsymbol{x},t)\right). \end{aligned} \tag{A.53}$$

Again, we expand the exponentials, but this time we keep all terms up to second order in Δt:

$$\begin{aligned} f_i(\boldsymbol{x}+\boldsymbol{c}_i\Delta t, t+\Delta t) = {} & \left(1-\frac{\Delta t}{\tau}+\frac{\Delta t^2}{2\tau^2}\right) f_i(\boldsymbol{x},t) \\ & + \frac{\Delta t}{2\tau}\left[f_i^{\mathrm{eq}}(\boldsymbol{x}+\boldsymbol{c}_i\Delta t, t+\Delta t) + \left(1-\frac{\Delta t}{\tau}\right) f_i^{\mathrm{eq}}(\boldsymbol{x},t)\right] + O(\Delta t^3). \end{aligned} \tag{A.54}$$

Our aim is to find a suitable transformation $f_i \to \bar{f}_i$ to bring (A.54) into the form of (3.76) with the BGK collision operator $\Omega_i = -(f_i - f_i^{\mathrm{eq}})/\tau$. In fact, this is possible by introducing the new population

$$\bar{f}_i = f_i - \frac{\Omega_i \Delta t}{2} = f_i + \frac{\left(f_i - f_i^{\mathrm{eq}}\right)\Delta t}{2\tau}. \tag{A.55}$$

As Ω_i conserves mass and momentum, this new population $\bar{f}_i$ has the same mass and momentum moments as f_i,

$$\begin{aligned} \sum_i \bar{f}_i &= \sum_i f_i - \sum_i \frac{\Omega_i \Delta t}{2} = \sum_i f_i = \rho, \\ \sum_i \bar{f}_i \boldsymbol{c}_i &= \sum_i f_i \boldsymbol{c}_i - \sum_i \frac{\Omega_i \boldsymbol{c}_i \Delta t}{2} = \sum_i f_i \boldsymbol{c}_i = \rho \boldsymbol{u}. \end{aligned} \tag{A.56}$$

After a series of algebraic manipulations [11] we arrive at

$$\bar{f}_i(\boldsymbol{x} + \boldsymbol{c}\Delta t, t + \Delta t) = \bar{f}_i(\boldsymbol{x}, t) - \frac{\bar{f}_i(\boldsymbol{x}, t) - f_i^{\mathrm{eq}}(\boldsymbol{x}, t)}{\bar{\tau}} + O(\Delta t^3) \tag{A.57}$$

with a modified relaxation time

$$\bar{\tau} = \tau + \frac{\Delta t}{2}. \tag{A.58}$$

A.6 MRT for D3Q15, D3Q19, and D3Q27 Velocity Sets

We provide the matrices $\boldsymbol{M}$, the equilibrium moments m_k^{eq} and the collision rates ω_k for the most widespread 3D MRT models. The D3Q15 and D3Q19 models are based on the Gram-Schmidt procedure [12]. We only provide references for D3Q27 as it is not often used.

Note that the matrices presented here assume that the velocity sets are ordered as in Sect. 3.4.7. Different choices of velocity order would lead to matrices with differently ordered columns.

A.6.1 D3Q15

The D3Q15 Gram-Schmidt moments are

$${}^{\mathrm{G}}\boldsymbol{m} = (\rho, e, \epsilon, j_x, q_x, j_y, q_y, j_z, q_z, p_{xx}, p_{ww}, p_{xy}, p_{yz}, p_{zx}, m_{xyz})^{\top}. \tag{A.59}$$

These correspond to density, energy, energy squared, momentum, heat flux and momentum flux. The relaxation rates are

$${}^{\mathrm{G}}\boldsymbol{S} = \mathrm{diag}(0, \omega_e, \omega_\epsilon, 0, \omega_q, 0, \omega_q, 0, \omega_q, \omega_\nu, \omega_\nu, \omega_\nu, \omega_\nu, \omega_\nu, \omega_m) \tag{A.60}$$

where collision rates of 0 are specified for the four conserved moments, i.e. density and momentum. (Note that collision rates of 0 for momentum are not suitable for simulating a force density $\boldsymbol{F}$ in the Navier-Stokes equation [13].)

The moment vectors are

$${}^{\mathrm{G}}M_{\rho,i} = 1, \quad {}^{\mathrm{G}}M_{e,i} = c_{ix}^2 + c_{iy}^2 + c_{iz}^2 - 2,$$

$${}^{\mathrm{G}}M_{\epsilon,i} = \frac{1}{2}(15(c_{ix}^2 + c_{iy}^2 + c_{iz}^2)^2 - 55(c_{ix}^2 + c_{iy}^2 + c_{iz}^2) + 32),$$

$$
\begin{gathered}
{}^{\mathrm{G}}M_{j_x,i} = c_{ix}, \quad {}^{\mathrm{G}}M_{q_x,i} = \frac{1}{2}(5(c_{ix}^2 + c_{iy}^2 + c_{iz}^2) - 13)c_{ix}, \\
{}^{\mathrm{G}}M_{j_y,i} = c_{iy}, \quad {}^{\mathrm{G}}M_{q_y,i} = \frac{1}{2}(5(c_{ix}^2 + c_{iy}^2 + c_{iz}^2) - 13)c_{iy}, \\
{}^{\mathrm{G}}M_{j_z,i} = c_{iz}, \quad {}^{\mathrm{G}}M_{q_z,i} = \frac{1}{2}(5(c_{ix}^2 + c_{iy}^2 + c_{iz}^2) - 13)c_{iz}, \\
{}^{\mathrm{G}}M_{p_{xx},i} = 3c_{ix}^2 - (c_{ix}^2 + c_{iy}^2 + c_{iz}^2), \quad {}^{\mathrm{G}}M_{p_{ww},i} = c_{iy}^2 - c_{iz}^2, \\
{}^{\mathrm{G}}M_{p_{xy},i} = c_{ix}c_{iy}, \quad {}^{\mathrm{G}}M_{p_{yz},i} = c_{iy}c_{iz}, \quad {}^{\mathrm{G}}M_{p_{xz},i} = c_{ix}c_{iz}, \\
{}^{\mathrm{G}}M_{m_{xyz},i} = c_{ix}c_{iy}c_{iz}.
\end{gathered}
\tag{A.61}
$$

They form the transformation matrix

$$
{}^{\mathrm{G}}\boldsymbol{M} = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-2 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
16 & -4 & -4 & -4 & -4 & -4 & -4 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 \\
0 & -4 & 4 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\
0 & 0 & 0 & -4 & 4 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & -4 & 4 & 1 & -1 & -1 & 1 & 1 & -1 & 1 & -1 \\
0 & 2 & 2 & -1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & -1 & -1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1
\end{pmatrix}. \tag{A.62}
$$

The inverse matrix ${}^{\mathrm{G}}\boldsymbol{M}^{-1}$ can be found using a computer program.

The corresponding equilibrium moments ${}^{\mathrm{G}}m_k^{\mathrm{eq}}$ are

$$
\begin{gathered}
e^{\mathrm{eq}} = -\rho + \rho(u_x^2 + u_y^2 + u_z^2), \quad \epsilon^{\mathrm{eq}} = \rho - 5\rho(u_x^2 + u_y^2 + u_z^2), \\
q_x^{\mathrm{eq}} = -\frac{7}{3}\rho u_x, \quad q_y^{\mathrm{eq}} = -\frac{7}{3}\rho u_y, \quad q_z^{\mathrm{eq}} = -\frac{7}{3}\rho u_z, \\
p_{xx}^{\mathrm{eq}} = 2\rho u_x^2 - \rho(u_y^2 + u_z^2), \quad p_{ww}^{\mathrm{eq}} = \rho(u_y^2 - u_z^2), \\
p_{xy}^{\mathrm{eq}} = \rho u_x u_y, \quad p_{yz}^{\mathrm{eq}} = \rho u_y u_z, \quad p_{xz}^{\mathrm{eq}} = \rho u_x u_z, \\
m_{xyz}^{\mathrm{eq}} = 0.
\end{gathered}
\tag{A.63}
$$

Note that the equilibrium moment ϵ^{eq} is not uniquely defined and can be tuned [12]. The resulting macroscopic stress tensor and viscosity are given in Sect. A.2.3.

Exercise A.4 Show that the equilibrium moments can be calculated as ${}^{\mathrm{G}}m_k^{\mathrm{eq}} = {}^{\mathrm{G}}M_{ki} f_i^{\mathrm{eq}}$ with f_i^{eq} from (3.54).

A.6.2 D3Q19

The D3Q19 Gram-Schmidt moments are

$$
{}^{\mathrm{G}}\boldsymbol{m} = (\rho, e, \epsilon, j_x, q_x, j_y, q_y, j_z, q_z, p_{xx}, \pi_{xx}, p_{ww}, \pi_{ww}, p_{xy}, p_{yz}, p_{xz}, m_x, m_y, m_z)^{\top}. \tag{A.64}
$$

There are additional moments compared to the D3Q15 model in Sect. A.6.1. They correspond to third-order (m_x, m_y, m_z) and fourth-order polynomials (π_{xx}, π_{ww}). The relaxation matrix reads

$$
{}^{\mathrm{G}}\boldsymbol{S} = \mathrm{diag}(0, \omega_e, \omega_\epsilon, 0, \omega_q, 0, \omega_q, 0, \omega_q, \omega_\nu, \omega_\pi, \omega_\nu, \omega_\pi, \omega_\nu, \omega_\nu, \omega_\nu, \omega_m, \omega_m, \omega_m). \tag{A.65}
$$

The moment vectors are

$$
\begin{aligned}
{}^{\mathrm{G}}M_{\rho,i} &= 1, \quad {}^{\mathrm{G}}M_{e,i} = 19(c_{ix}^2 + c_{iy}^2 + c_{iz}^2) - 30, \\
{}^{\mathrm{G}}M_{\epsilon,i} &= \big(21(c_{ix}^2 + c_{iy}^2 + c_{iz}^2)^2 - 53(c_{ix}^2 + c_{iy}^2 + c_{iz}^2) + 24\big)/2, \\
{}^{\mathrm{G}}M_{j_x,i} &= c_{ix}, \quad {}^{\mathrm{G}}M_{q_x,i} = \big(5(c_{ix}^2 + c_{iy}^2 + c_{iz}^2) - 9\big)c_{ix}, \\
{}^{\mathrm{G}}M_{j_y,i} &= c_{iy}, \quad {}^{\mathrm{G}}M_{q_y,i} = \big(5(c_{ix}^2 + c_{iy}^2 + c_{iz}^2) - 9\big)c_{iy}, \\
{}^{\mathrm{G}}M_{j_z,i} &= c_{iz}, \quad {}^{\mathrm{G}}M_{q_z,i} = \big(5(c_{ix}^2 + c_{iy}^2 + c_{iz}^2) - 9\big)c_{iz}, \\
{}^{\mathrm{G}}M_{p_{xx},i} &= 3c_{ix}^2 - (c_{ix}^2 + c_{iy}^2 + c_{iz}^2), \\
{}^{\mathrm{G}}M_{\pi_{xx},i} &= \big(3(c_{ix}^2 + c_{iy}^2 + c_{iz}^2) - 5\big)\big(3c_{ix}^2 - (c_{ix}^2 + c_{iy}^2 + c_{iz}^2)\big), \\
{}^{\mathrm{G}}M_{p_{ww},i} &= c_{iy}^2 - c_{iz}^2, \quad {}^{\mathrm{G}}M_{\pi_{ww},i} = \big(3(c_{ix}^2 + c_{iy}^2 + c_{iz}^2) - 5\big)\big(c_{iy}^2 - c_{iz}^2\big), \\
{}^{\mathrm{G}}M_{p_{xy},i} &= c_{ix}c_{iy}, \quad {}^{\mathrm{G}}M_{p_{yz},i} = c_{iy}c_{iz}, \quad {}^{\mathrm{G}}M_{p_{xz},i} = c_{ix}c_{iz} \\
{}^{\mathrm{G}}M_{m_x,i} &= (c_{iy}^2 - c_{iz}^2)c_{ix}, \quad {}^{\mathrm{G}}M_{m_y,i} = (c_{iz}^2 - c_{ix}^2)c_{iy}, \quad {}^{\mathrm{G}}M_{m_z,i} = (c_{ix}^2 - c_{iy}^2)c_{iz}.
\end{aligned} \tag{A.66}
$$

They define the transformation matrix

$$
{}^{\mathrm{G}}\boldsymbol{M} = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-30 & -11 & -11 & -11 & -11 & -11 & -11 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\
12 & -4 & -4 & -4 & -4 & -4 & -4 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 \\
0 & -4 & 4 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & -4 & 4 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & -1 & 1 & -1 & 1 \\
0 & 0 & 0 & 0 & 0 & -4 & 4 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & -1 & 1 & -1 & 1 \\
0 & 2 & 2 & -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 & -2 & -2 & 1 & 1 & 1 & 1 & -2 & -2 \\
0 & -4 & -4 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & -2 & -2 & 1 & 1 & 1 & 1 & -2 & -2 \\
0 & 0 & 0 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 \\
0 & 0 & 0 & -2 & -2 & 2 & 2 & 1 & 1 & -1 & -1 & 0 & 0 & 1 & 1 & -1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & -1 & 1 & 1 & -1
\end{pmatrix}. \tag{A.67}
$$

Also in this case, the inverse ${}^{\mathrm{G}}\boldsymbol{M}^{-1}$ can be found using a computer program.

The equilibrium moments are

$$
\begin{aligned}
e^{\mathrm{eq}} &= -11\rho + 19\rho(u_x^2 + u_y^2 + u_z^2), \quad \epsilon^{\mathrm{eq}} = 3\rho - \frac{11}{2}\rho(u_x^2 + u_y^2 + u_z^2), \\
q_x^{\mathrm{eq}} &= -\frac{2}{3}\rho u_x, \quad q_y^{\mathrm{eq}} = -\frac{2}{3}\rho u_y, \quad q_z^{\mathrm{eq}} = -\frac{2}{3}\rho u_z, \\
p_{xx}^{\mathrm{eq}} &= 2\rho u_x^2 - \rho(u_y^2 + u_z^2), \quad \pi_{xx}^{\mathrm{eq}} = -\frac{1}{2}\big(2\rho u_x^2 - \rho(u_y^2 + u_z^2)\big), \\
p_{ww}^{\mathrm{eq}} &= \rho(u_y^2 - u_z^2), \quad \pi_{ww}^{\mathrm{eq}} = -\frac{1}{2}\rho(u_y^2 - u_z^2), \\
p_{xy}^{\mathrm{eq}} &= \rho u_x u_y, \quad p_{yz}^{\mathrm{eq}} = \rho u_y u_z, \quad p_{xz}^{\mathrm{eq}} = \rho u_x u_z, \\
m_x^{\mathrm{eq}} &= 0, \quad m_y^{\mathrm{eq}} = 0, \quad m_z^{\mathrm{eq}} = 0.
\end{aligned} \tag{A.68}
$$

The resulting macroscopic stress tensor and viscosity are provided in Sect. A.2.3.

A.6.3 D3Q27

The D3Q27 model has the largest memory footprint among the common 3D models, and it is the most computationally demanding. The MRT formulation of the collision operator leads to additional computational overhead. Thus, although D3Q27 has the best isotropy properties, its MRT counterpart is not commonly used. One can construct the D3Q27 MRT model from velocity polynomials [14] and the Gram-Schmidt approach [15] that we have discussed in Sect. 10.2. Another alternative is to use a variation of the D3Q27 MRT model for the so-called cascaded lattice Boltzmann model [16, 17].

A.7 Planar Interface for the Free Energy Gas-Liquid Model

We show that the free energy multiphase model satisfies the Maxwell construction rule. To do this, let us examine the planar density profile. The stationary interface between gas and liquid phases is assumed at $x = 0$, and the density changes along the x-axis. Thus, we search the density profile in the form $\rho = \rho(x)$. Far away from the interface, we expect the fluid to assume the gas density $\rho_\mathrm{g} = \rho(-\infty)$ and the liquid density $\rho_\mathrm{l} = \rho(+\infty)$.

Since the interface is stationary, the momentum flux P_{xx} (cf. (9.30)) must be constant along the x-axis, i.e. $\mathrm{d}P_{xx}/\mathrm{d}x = 0$. Its value equals the bulk pressure p_0 far away from the interface where all density gradients are zero. These constraints result in

$$p_0 = p_\mathrm{b}(\rho) + \frac{k}{2}\rho'^2 - k\rho\rho'', \tag{A.69a}$$

$$p_0 = p_\mathrm{b}(\rho_\mathrm{g}) = p_\mathrm{b}(\rho_\mathrm{l}), \tag{A.69b}$$

where the prime denotes the derivative with respect to x. Our aim is to solve this system of equations to find the values of the liquid and gas densities, ρ_l and ρ_g.

Exercise A.5 To find the density profile as function of the spatial coordinate x from (A.69a), we can introduce a substitution $z = \rho'^2$. This substitution is widely used when in a second-order ODE there is no explicit involvement of the independent variable, i.e. x. Show that the second derivative of the density obeys $\rho'' = \dot{z}/2$, where the dot denotes the derivative with respect to density ρ.

After the introduction of $z = \rho'^2$, changing the independent variable x to ρ, and performing some calculations, the ODE for the density profile becomes

$$\dot{z} - \frac{z}{\rho} = \frac{2}{k\rho}(p_\mathrm{b}(\rho) - p_0). \tag{A.70}$$

This equation is of the form $\dot{z} + f(\rho)z = g(\rho)$ that has the solution

$$z(\rho) = \mathrm{e}^{-\int^{\rho} f(\tilde{\rho})\,\mathrm{d}\tilde{\rho}} \left(\int^{\rho} g(\tilde{\rho}) \mathrm{e}^{\int^{\tilde{\rho}} f(\tilde{\tilde{\rho}})\,\mathrm{d}\tilde{\tilde{\rho}}}\,\mathrm{d}\tilde{\rho} + C \right) \tag{A.71}$$

where C has to be found from the boundary conditions.

In our case, we identify $f(\rho) = -1/\rho$ and $g(\rho) = \frac{2}{k\rho}(p_{\mathrm{b}}(\rho) - p_0)$ and therefore

$$z(\rho) = \rho \left(\int^{\rho} \frac{2}{k\tilde{\rho}^2} \Big(p_{\mathrm{b}}(\tilde{\rho}) - p_0 \Big)\,\mathrm{d}\tilde{\rho} + C \right). \tag{A.72}$$

The boundary conditions are $z = \rho'^2 = 0$ far away from the interface where there are no density gradients, i.e. $z(\rho_{\mathrm{g}}) = 0$ and $z(\rho_{\mathrm{l}}) = 0$. This is only possible if z has the solution

$$z(\rho) = \frac{2\rho}{k} \int_{\rho_{\mathrm{g}}}^{\rho} (p_{\mathrm{b}}(\tilde{\rho}) - p_0) \frac{\mathrm{d}\tilde{\rho}}{\tilde{\rho}^2}. \tag{A.73}$$

The boundary condition $z(\rho_{\mathrm{l}}) = z(\rho_{\mathrm{g}}) = 0$ results is nothing else than the **Maxwell area construction rule for gas-liquid systems**:

$$\int_{\rho_{\mathrm{g}}}^{\rho_{\mathrm{l}}} (p_{\mathrm{b}}(\tilde{\rho}) - p_0) \frac{\mathrm{d}\tilde{\rho}}{\tilde{\rho}^2} = 0. \tag{A.74}$$

This is consistent as the gas-liquid model with the pressure tensor from (9.30) is obtained from the free-energy functional based on principles of thermodynamics.

If we want to find an expression for the density profile $\rho(x)$, we can use $z = (\mathrm{d}\rho/\mathrm{d}x)^2$ and solve the implicit integral equation assuming that the interface lies right in the middle between phases, i.e. $\bar{\rho} = \rho(x = 0) = (\rho_{\mathrm{g}} + \rho_{\mathrm{l}})/2$:

$$x = \int_{\bar{\rho}}^{\rho} \frac{\mathrm{d}\tilde{\rho}}{\sqrt{z(\tilde{\rho})}}. \tag{A.75}$$

If the equation of state $p_{\mathrm{b}}(\rho)$ includes a double-well potential as in (9.31), then the density profile will be of tanh-form as in (9.70).

A.8 Planar Interface for the Shan-Chen Liquid-Vapour Model

Appendix A.7 contains the free-energy calculations for the planar interface in a liquid-vapour system. Here we repeat these calculations in the context of the Shan-Chen (SC) model.

The SC pressure tensor from (9.112) reads (again setting $\Delta t = 1$ for simplicity)

$$P^{\mathrm{SC}}_{\alpha\beta} = \left(c_{\mathrm{s}}^2 \rho + \frac{c_{\mathrm{s}}^2 G}{2} \psi^2 + \frac{c_{\mathrm{s}}^4 G}{4} (\nabla \psi)^2 + \frac{c_{\mathrm{s}}^4 G}{2} \psi \Delta \psi \right) \delta_{\alpha\beta} - \frac{c_{\mathrm{s}}^4 G}{2} (\partial_\alpha \psi)(\partial_\beta \psi). \tag{A.76}$$

We can distinguish between the equation of state $p_{\mathrm{b}}(\rho) = c_{\mathrm{s}}^2 \rho + (c_{\mathrm{s}}^2 G/2)\psi^2(\rho)$ from (9.111) that dictates the bulk behaviour and the other terms that contain derivatives of ψ and therefore are important near the interface between phases.

As in Appendix A.7, we consider a planar interface at $x = 0$. In mechanical equilibrium, the component P^{SC}_{xx} must be constant across the interface. This allows us to compute the density profile $\rho(x)$.

We can rewrite P^{SC}_{xx} as

$$P^{\mathrm{SC}}_{xx} = c_{\mathrm{s}}^2 \rho + \frac{c_{\mathrm{s}}^2 G}{2} \psi^2(\rho) - \frac{c_{\mathrm{s}}^4 G}{4} \left(\dot{\psi}(\rho) \rho' \right)^2 + \frac{c_{\mathrm{s}}^4 G}{2} \psi(\rho) \left(\ddot{\psi}(\rho) \rho' + \dot{\psi}(\rho) \rho'' \right). \tag{A.77}$$

The prime denotes the derivative with respect to x, the dot the derivative with respect to the density ρ. Far away from the interface, both in the gas (g) and liquid (l) phases, gradients vanish and we find the bulk pressure

$$p_0 = p_{\mathrm{b}}(\rho_{\mathrm{g}}) = c_{\mathrm{s}}^2 \rho_{\mathrm{g}} + \frac{c_{\mathrm{s}}^2 G}{2} \psi^2(\rho_{\mathrm{g}}) = p_{\mathrm{b}}(\rho_{\mathrm{l}}) = c_{\mathrm{s}}^2 \rho_{\mathrm{l}} + \frac{c_{\mathrm{s}}^2 G}{2} \psi^2(\rho_{\mathrm{l}}). \tag{A.78}$$

Now we introduce $z = \rho'^2$ and perform steps similar to those detailed in Appendix A.7. This leads to

$$z(\rho) = \frac{4}{c_{\mathrm{s}}^4 G} \frac{\psi(\rho)}{\dot{\psi}^2(\rho)} \int_{\rho_{\mathrm{g}}}^{\rho} \left(p_0 - c_{\mathrm{s}}^2 \tilde{\rho} - \frac{c_{\mathrm{s}}^2 G}{2} \psi^2(\tilde{\rho}) \right) \frac{\dot{\psi}(\tilde{\rho})}{\psi^2(\tilde{\rho})} \mathrm{d}\tilde{\rho}. \tag{A.79}$$

To satisfy the boundary conditions $z(\rho_{\mathrm{g}}) = 0$ and $z(\rho_{\mathrm{l}}) = 0$ we need

$$\int_{\rho_{\mathrm{g}}}^{\rho_{\mathrm{l}}} \left(p_0 - c_{\mathrm{s}}^2 \tilde{\rho} - \frac{c_{\mathrm{s}}^2 G}{2} \psi^2(\tilde{\rho}) \right) \frac{\dot{\psi}(\tilde{\rho})}{\psi^2(\tilde{\rho})} \mathrm{d}\tilde{\rho} = 0. \tag{A.80}$$

We compare this expression with the Maxwell area construction rule:

$$\int_{\rho_g}^{\rho_l} \left(p_0 - c_s^2 \tilde{\rho} - \frac{c_s^2 G}{2} \psi^2(\tilde{\rho}) \right) \frac{\mathrm{d}\tilde{\rho}}{\tilde{\rho}^2} = 0. \tag{A.81}$$

Obviously, the SC model can only reproduce thermodynamic consistency for

$$\frac{\dot{\psi}(\tilde{\rho})}{\psi^2(\tilde{\rho})} = \frac{1}{\tilde{\rho}^2} \tag{A.82}$$

which is solved by $\psi(\rho) = \rho$.

The final step is to find the density profile across the planar interface by solving the ordinary differential equation $\rho' = \sqrt{z}$ for $\rho = \rho(x)$:

$$x(\rho) = \int_{\bar{\rho}}^{\rho} \frac{\mathrm{d}\tilde{\rho}}{\sqrt{z(\tilde{\rho})}}. \tag{A.83}$$

Here we have defined the x-axis in such a way that the interface location at $x = 0$ coincides with the average $\bar{\rho} = (\rho_l + \rho_g)/2$. In practise, however, obtaining a closed form for (A.83) is usually not possible.

A.9 Programming Reference

To assist readers who are unfamiliar with programming in C or C++, this appendix reviews the main features of these languages that are used in the code presented in Chap. 13 and accompanying the book. The code is written in standard C++ (1998). It uses only several features of C++ for convenience, such as function overloading (which allows functions to have the same name provided they have different parameters), and could be converted easily to a C code.

The text that constitutes a program, or a portion of one, is called source code. Source code for C and C++ does not need to follow strict formatting rules, and whitespace (spaces, tabs, and new lines) can be freely used to assist interpretation of the code.

Source code consists of a set of declarations that describe the units of the program and the tasks they perform. The source code for programs is often split among different files to separate it into portions that perform related tasks and to allow these portions of code to be used in different programs. The separate source files are combined into one complete program during compilation and linking (Sect. A.9.15).

One of the most important declarations in the source code of a program is for the `main()` function: this is where the program starts when the operating system runs it. In general, functions are named sequences of statements. They are used to group together the statements that accomplish a specific task and should have a descriptive

name that describes what they do. Functions also allow programmers to repeat a task in different parts of a program without having to write out the same statements.

Simple statements end with a semicolon, while compound statements are sequences of simple statements enclosed in braces, { and }. Statements can involve variable declarations, evaluation of an expression and storage of the result to a variable through the use of the assignment operator =, or function calls. The various types of statements are explained in the sections that follow.

A.9.1 Comments

Comments are blocks of text that are ignored by the compiler but are useful for anyone who is reading the code. They are used to document the behaviour of code and describe design decisions. Double slashes, `//`, indicate the start of a comment that extends to the end of the current line. Comments can also be enclosed between `/*` and `*/`. Such comments may span multiple lines.

A.9.2 Expressions and Operators

Expressions can involve a variety of operators. The assignment operator = is used to save the result of evaluating its right hand side to the variable on the left hand side, as in `x = y + 5`.

The arithmetic operators are `+`, `-`, `*`, `/`, `%`, for addition, subtraction, multiplication, division, and remainder upon division, respectively, as well as the increment (`++`) and decrement (`--`) operators. These latter two operators have different effects when they appear before or after the variable they are applied to. When appearing before a variable, as in `b = ++a` or `b = --a`, they return the value in the variable after the operation is performed on it. In other words, `b = ++a` is equivalent to `a = a+1; b = a;` and `b = --a` is equivalent to `a = a-1; b = a;`. In contrast, when the operators appear after the variable, they return the value of that variable before it is modified. In this case, `b = a++` is equivalent to `b = a; a = a+1;` and `b = a--` is equivalent to `b = a; a = a-1;`. The pre- and post-increment/decrement operators are a common source of confusion, and code should be written so that the task it performs is clear. These operators can also be used without assignment, for example `a++;` by itself is equivalent to `a = a+1`.

Relational operators are used to compare values, and they are == (equal), `!=` (not equal), < (less than), > (greater than), <= (less than or equal), >= (greater than or equal). The difference between the assignment and equality operators is particularly important because assignment can be used in a conditional expression and returns the value that was assigned.

The logical operators are `||` (or), `&&` (and), and `!` (not). The last of these is used before the expression it modifies, for example `!(a || b)` is logically equivalent

to `!a && !b`. Readers may consult standard references about the precedence rules for these operators. Parentheses can be used to specify the order of subexpression evaluation.

A.9.3 Data Types

The type of data stored in each variable must be explicitly stated before the variable is used in another statement. The names of all variables and functions are case sensitive. The main data types used in the code in this chapter are `int`, `unsigned int`, and `double`, which on the architectures we use correspond to a 32 bit signed integer (positive and negative values allowed), 32 bit unsigned (non-negative) integer, and 64 bit (double precision) floating point value, respectively. Text characters (or small numbers) are stored in `char` variables, which occupy one byte (8 bits). In special circumstances the unsigned integer type `size_t` is used, for example, to store the sizes of memory regions in bytes. On common 64 bit x86 architectures, `size_t` is a 64 bit unsigned integer. Variables that store text are arrays of `chars` (see Sect. A.9.8).

The statements `int i;` and `double d;` declare integer variables `i` and `d` to be an `int` and a `double`, respectively. Variables should not be used until they are given an initial value. Variables can be initialised when they are declared: `int x = 5;` is effectively shorthand for `int x; x = 5;`.

Variables declared with the `const` keywords are read-only variables. After they are initialised, they cannot be modified. Compilation of `const int five = 5; five = 6;` will stop with an error message.

A.9.4 Composite Data Types

Structures are composite data types that group several variables together. The components of a structure can be used in expressions or modified by writing the name of the structure variable followed by a dot and then the name of the variable within the structure. For example, to describe a vector structure containing two variables `x` and `y` we write:

```
struct vect {
    double x; double y;
};
```

We can then create one of these vectors, initialise it, and compute its norm:

```
vect v;
v.x = 5.0;
v.y = 0.1;
double norm = sqrt(v.x*v.x + v.y*v.y);
```

A.9.5 Variable Scope

Variables declared within a block of code or a function, i.e. between { and }, can only be used inside that block/function. The memory for these variables is automatically managed: space is set aside for the variables before entering the block/function and it is released at the end.

Global variables are declared outside of any function and can be used anywhere in the code (after they have been declared).

A.9.6 Pointers

An important feature of C and C++ is pointer variables. These are special variables that "refer" or "point" to another variable rather than holding a value. Such variables instead store the location of the variable they "point" to, i.e. their address in the system's memory.

The syntax for declaring a pointer is `type *v`, which declares that `v` is a variable that contains the location in memory of a variable of type `type`. Supposing we have declared `int *i;`, it is incorrect to write `i = 5;` because one would be using an integer value (5) where a valid memory address is needed (one that the program is permitted to access). To use `i` correctly, we must ensure that `i` points to a location in memory that has been reserved for an `int`. This can be done by explicitly allocating memory (see Sect. A.9.7) for an integer or by assigning the pointer the address of an integer variable.

To assign a pointer variable the address of another variable, we use the "address of" operator `&`. For example, we can write `int a = 10; int *i = &a;` to create a pointer `i` that refers to the variable `a`. We can then either directly change `a`, using `a = 2;`, or change it through the pointer `i` by using `*i = 2;`. Here, `*` is the pointer dereferencing operator, which indicates that we want to work with the value the pointer refers to rather than the location of that value in the computer's memory. Informally, we can say that `i` is the pointer, and `*i` is the value that the pointer refers to.

Pointer Arithmetic

When pointers refer to elements in an array (cf. Sect. A.9.8), arithmetic expressions involving the pointer variable can be used to access other elements in that array. For example, when `p` points to a `double`, `p+1` points to the next double, and `p-1` points to the previous double. Parentheses are essential for pointer arithmetic: `*(p+1)` refers to the contents of the double after the one at `p`, while `*p+1` is the result of adding 1 to the value pointed to by `p`.

Pointers must not be used to refer to memory that has not been reserved for use by the program (Sect. A.9.7), i.e. that is outside the bounds of an allocated array (Sect. A.9.8 and Sect. A.9.7). In such cases, behaviour is undefined and can cause the operating system to terminate the program.

A.9.7 Dynamic Memory Allocation

When the amount of memory that needs to be reserved for a variable is not known at compilation time or the required amount of memory is too large to reside in the space that is managed automatically (called the stack), programmers need to explicitly reserve regions of memory in what is called the "heap" for these variables. The allocation of memory for variables during the execution of a program is called dynamic memory allocation.

In C, the function `void* malloc(size_t size)` requests `size` bytes of memory and returns the address of the first reserved byte or a null pointer (value of zero) if the request could not be satisfied due to insufficient memory being available. Since the return type is a pointer to `void` it must be converted to a pointer to the correct type of variable. For example, the statement `double *data = (double*) malloc(100*sizeof(double))` requests space for 100 `doubles`. Note that `malloc` knows nothing about the type of variable being allocated: the `size` specified in `malloc` is the number of bytes required, not the number of variables of the desired type. We therefore use the `sizeof` operator to determine the number of bytes that a `double` occupies, since the number of bytes used for different variable types can vary across platforms.

The memory reserved by `malloc` is not initialised: it contains whatever was held at those memory addresses previously. This memory remains reserved until a later call to `void free(void *ptr)`. This function releases the memory, allowing it to be reserved by subsequent calls to `malloc`. Repeated use of `malloc` without matching calls to `free` will cause a program to use more and more memory over time. This is called a *memory leak* and must be avoided because it can interfere with other programs running on a system and will eventually lead to a call to `malloc` failing due to a lack of available memory.

In C++, the operators `new`, `delete`, and `delete[]` are used to allocate memory, free individual variables, and free arrays of variables, respectively. These operators are useful when working with the object oriented features in C++. The C functions `malloc` and `free` may still be used in C++ and they are sufficient for the simple data types we employ in the code.

A.9.8 Arrays

Many algorithms use sets of data that are conveniently stored in arrays. Arrays are sequences of variables of the same type that occupy contiguous blocks of memory. The values in arrays are accessed through their index (location) within the array. The syntax for declaring an array is

```
type varname[N];
```

where `N` is an integer constant that specifies the number of elements of type `type` that are available in the array with name `varname`. When the initial content of the array is known, the array can be initialised when it is declared, for example as

```
int integers[] = {1,2,3,4,5};
```

for an array of five integers. In this case, the number of elements in the array does not need to be specified and is inferred from the initialising list of values. If initial values are not provided, no assumptions can be made about the contents of the array; it does not contain zeroes by default, and its initial contents must be specified in subsequent code.

Elements of an array can be read or modified by using the array subscript operator that consists of an integer expression in between brackets. For example, the third element of the `integers` array could be modified by writing `integers[2] = 10;`. The syntax is the same when an element of an array is used in an expression, such as `x = integers[2]+5;`, which adds 5 to the third element in `integers` and stores the result to a variable `x`. Note that the first element of an array has index 0, and the index of the last element in an array with N elements is $N-1$. Programmers must ensure that index expressions are bounded between these limits; it is an error to access memory outside the limits, and can cause the operating system to terminate the program.

Dynamically allocated arrays are used for the large data sets that store the populations for the LBM simulations, as described in Sect. 13.3.1. When a pointer is known to refer to a section of memory that is occupied by an array, the array subscript operator can be used with the pointer. For example, if `double *p` points to the start of an array of 10 `doubles`, one may use `p[0]`, `p[1]`, up to `p[9]` to read or modify the elements of this array. Array indexing expressions with pointers are equivalent to applying an offset to the pointer, for example `p[5]` is equivalent to `*(p+5)`.

Text is stored and manipulated using arrays of `chars`. Such variables can be initialised using text entered in between double quotes, for example `"sample text"`. The first element of the resulting array is the letter `'s'`. Backslashes, `\`, followed by one letter can be used to include special characters in literal text: among others, `\n` is replaced with a new line character, `\t` represents a tab character, and `\\` is replaced with a single backslash.

A.9.9 If Statement

Several special statements are used to selectively execute and repeat sequences of statements, the first of which is an `if` statement:

```
if( conditional_expression )
{
  // statements executed if conditional_expression is true
}
else
{
  // statements executed if conditional_expression is false
}
```

If the `conditional_expression` evaluates to a logical value that is considered `true`, the first block of statements (enclosed between { and }) is executed. Otherwise the block following `else` is executed. When blocks contain only a single statement, the braces are often omitted. Such unneeded braces are usually omitted when a chain of `if` statements is used to test which one of several conditions holds. For example, it is common to write

```
if( case_1 )
{
  // statements for case 1
}
else if( case_2 )
{
  // statements for case 2
}
else if( case_3 )
{
  // statements for case 3
}
else
{
  // for when none of the previous cases hold
}
```

This is much easier to read than code that includes additional braces around each nested `if` statement.

A.9.10 While Loop

A `while` loop is written as:

```
while( conditional_expression )
{
  // statements executed while conditional_expression is true
}
```

When this loop is executed, the conditional expression is evaluated and the body is executed if the result is `true`. Evaluation of the conditional and execution of the

inner block continue indefinitely until the conditional evaluates as `false`. When the conditional is `false`, execution continues with the statements that follow after the `while` loop.

A.9.11 For Loop

The second common type of loop is a `for` loop that is commonly used when an index variable is needed to keep track of the iterations of the loop. This loop has the structure

```
for(init; condition; post_loop)
{
  // body of for loop
}
```

and it is effectively equivalent to the code

```
{
  init;
  while(condition)
  {
    // body of for loop

    post_loop;
  }
}
```

This shows that a `for` loop consists of four elements: an initialisation statement `init` that is executed before the loop starts, a conditional expression `condition`, a body that is repeated as long as the conditional expression is `true`, and an update statement `post_loop` that is executed after each repetition of the loop body.

`for` loops are useful when repeating a task a particular number of times and using the iteration number within the body. For example, one might compute a sum as follows:

```
double sum = 0.0;
for(int n = 1; n < 5; ++n)
{
  sum = sum + 1.0/n;
}
```

The initialisation statement declares an integer variable `n` and initialises it to 0. The update statement increments this integer, and the conditional expression allows terms to be added to the sum as long as `n` is less than 5 (i.e. up to and including 4). Variables declared in the initialisation statement can only be used within the `for` loop.

A.9.12 Functions

Functions are defined as follows:

```
return_type function_name(type1 param1, type2 param2)
{
  // body of function
  return return_value;
}
```

This creates a function with the name `function_name` whose body consists of a sequence of statements that uses the data supplied in the parameters to generate an output. The output value is `return_value`, a variable of type `return_type`.

The parameters of the function (`param1` and `param2` or more as needed) are variables that can only be used within the body of the function. The result (or output) of the function is specified with a `return` statement that ends execution of the function, allowing execution to continue in whichever function called this function. The use of `void` as the return type indicates that the function does not return a value. In this case, `return_value` is omitted from the `return` statement, which then becomes `return;`. If no `return;` statement is used in a `void` function, the function automatically returns when the end of the its body is reached.

Functions are invoked (or called) using their name and a list of the values (called arguments) to be used for each parameter. The values of the arguments are copied into temporary variables that are then used within the function. Functions that return a value can be used within expressions, such as when using mathematical functions: `y = sin(2*x)+1.5;`. Functions that do not return a value (`void` functions) are used as a statement by themselves, for example `perform_task(value1,value2);`. Non-`void` functions can also be used in this way when their return value is not needed, such as `finish_task();` instead of `status = finish_task();` when `status` is not used subsequently.

When function parameters are pointers, the function body can modify the data they point to unless a `const` keyword is used to disallow modification.

A.9.13 Screen and File Output

The code accompanying this book uses the standard C functions for displaying text and writing data to files. The complete details about the functions mentioned below can be found in many references about the C language.

The `void printf(const char* format_string,...)` function is used to display text and convert other variables, such as integers and floating point numbers, to text. This conversion is performed by scanning `format_string` for special character sequences, called format specifiers, that start with a percent sign, `%`. The ellipsis in the function definition indicates a list of variables that is matched (in order) with each format specifier in `format_string`. The characters after the percent sign in the format specifiers indicate how the variables in the variable list

should be interpreted and displayed as text. For example, `%d` is used to show signed integers in decimal notation, and `%e`, `%f`, or `%g` are used for displaying floating point numbers in different ways. Format specifiers can also include numbers and other special characters to further describe how the variables will be represented as text.

Binary data is written to files using the `fopen`, `fwrite`, and `fclose` functions as demonstrated in Sect. 13.3.3.

A.9.14 Header Files

When a source code file is compiled, the compiler does not need to know all the details about the functions and variables that are used in that file. The compiler only needs to know enough to set up function calls, allocate memory correctly, and check whether expressions involving the variables/function are permitted. The compiler assumes that the details of what functions do will be supplied later (during linking; see Sect. A.9.15). The declarations of functions (their names, parameters, and return types[2]) and their definitions (what the function does) can therefore be separated into different files. This helps organise the source code for long programs.

Files that contain the declarations of functions and variables that are used in other files are called header files, and their names end in the extension `.h`. Source files that make use of the functions and variables declared in header files must indicate that these files should be loaded before continuing with compilation. This is done with the preprocessor directive[3] `#include`. This directive has two forms: `#include <filename>` and `#include "filename"`. The first form is used to load header files for standard functions, such as `stdio.h` and `math.h`, and other libraries installed in system-specific directories. The second form is used to load header files that are stored together with the source code. The preprocessor searches for these header files in the directory that contains the source file being compiled.

A.9.15 Compilation and Linking

Compilers are programs that translate source code into binary files that contain the instructions that a processor will perform when running the program. The process of converting source code into machine instructions is called compilation. Compilation

[2]The declarations of functions are effectively the definitions of the functions with the body removed and replaced with a semicolon to indicate the end of the declaration.

[3]This is a special statement that is handled by the preprocessor, a program that performs several pre-compilation tasks that include inserting the contents of any files specified with `#include` into the source code that is then passed along to the compiler.

of a source code file produces an object file with machine instructions together with information about the functions and data structures it contains. Object files do not necessarily contain all the functions and data structures needed to form a complete program. A process called linking combines all the object files that are needed to generate a particular program. Linking creates the binary executable file that can then be run by an operating system.

In the GNU Compiler Collection, the compiler for C code is `gcc` and the compiler for C++ code is `g++`. To use `g++` to compile code for this book, a sample command is

```
g++ -c -O3 source1.cpp -o source1.o
```

This compiles the source code file `source1.cpp` with level 3 optimisation to generate the object file `source1.o`. The `-c` option indicates that we are only compiling the source file into an object file, linking should not yet be performed, and the output is not a full program.

To link several object files (and optionally first compile a source code file as well), the command is:

```
g++ -O3 source1.o source2.o main.cpp -o program
```

This compiles `main.cpp` and links the result with `source1.o` and `source2.o` to generate the program named `program`. The same command `g++` is used for both compilation and linking here. `g++` invokes other programs to carry out different steps in the compilation process, such as `ld`, which is the "linker" used for linking. `g++` determines what task to perform from the options that are used (such as `-c`) and the extensions of the files that are specified (`.cpp` or `.cc` for C++ source code and `.o` for object files).

A shortcut for quickly compiling and linking several source files directly to an executable is available:

```
g++ -O3 source1.cpp source2.cpp main.cpp -o program
```

This generates the executable file `program` from the source code files `source1.cpp`, `source2.cpp`, and `main.cpp`. On Windows, the file name of the output executable requires the extension `.exe`. After compilation, one can run the generated program on the command line by using the command `./program` on Unix systems and `program.exe` in Windows.

References

1. P.A. Thompson, *Compressible-Fluid Dynamics* (McGraw-Hill, New York, 1972)
2. L.E. Kinsler, A.R. Frey, A.B. Coppens, J.V. Sanders, *Fundamentals of Acoustics*, 4th edn. (Wiley, New York, 2000)
3. P.J. Dellar, Phys. Rev. E **64**(3) (2001)
4. P.J. Dellar, J. Comput. Phys. **259**, 270 (2014)
5. E.M. Viggen, The lattice Boltzmann method: Fundamentals and acoustics. Ph.D. thesis, Norwegian University of Science and Technology (NTNU), Trondheim (2014)

6. T. Shao, T. Chen, R. Frank, Math. Comp. **18**, 598 (1964)
7. M. Abramowitz, I. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables* (U.S. Government Printing Office, Washington, D.C., 1964)
8. X. Shan, X.F. Yuan, H. Chen, J. Fluid Mech. **550**, 413 (2006)
9. W.P. Yudistiawan, S.K. Kwak, D.V. Patil, S. Ansumali, Phys. Rev. E **82**(4), 046701 (2010)
10. X. He, L.S. Luo, Phys. Rev. E **56**(6), 6811 (1997)
11. S. Ubertini, P. Asinari, S. Succi, Phys. Rev. E **81**(1), 016311 (2010)
12. D. d'Humières, I. Ginzburg, M. Krafczyk, P. Lallemand, L.S. Luo, Phil. Trans. R. Soc. Lond. A **360**, 437 (2002)
13. I. Ginzburg, F. Verhaeghe, D. d'Humières, Commun. Comput. Phys. **3**, 427 (2008)
14. R. Rubinstein, L.S. Luo, Phys. Rev. E **77**(036709), 1 (2008)
15. K. Suga, Y. Kuwata, K. Takashima, R. Chikasue, Comput. Math. Appl. **69**(6), 518 (2015)
16. M. Geier, A. Greiner, J. Korvink, Phys. Rev. E **73**(066705), 1 (2006)
17. K. Premnath, S. Banerjee, J. Stat. Phys. **143**, 747 (2011)

Index

accuracy, 136, 149, 275, 276, 311, 511
  order, 32, 136, 139, 141, 146
acoustic viscosity number, 500, 501, 510, 527
advection-diffusion equation, 225, 297, 300, 316

block, 625
Boltzmann equation, 21, 26, 55, 61, 70, 106
  discrete-velocity, 83, 94, 108, 237, 506
Bond number, 290
boundary condition
  wetting for multicomponent free energy model, 361
  deformable, 435, 446, 464, 465, 473, 483, 485
  Dirichlet, 158, 318, 319, 321, 327
  free-slip, 206, 208
  moving, 434, 440, 443, 446, 450–452, 463, 464, 475
  Neumann, 158, 317, 320, 321, 327
  no-slip, 156, 466
  non-reflecting, 519, 526
  Robin, 158
  symmetric, 206, 208, 327
boundary conditon
  wetting for multiphase free energy model, 358
boundary scheme
  anti-bounce-back, 200, 318, 324
  at corners, 201, 205
  bounce-back, 156, 199, 200, 250, 251, 435, 455
  ghost methods, 455, 463
  immersed boundary method, *see* immersed boundary method
  Inamuro, 199, 319, 320, 324
  interpolated bounce-back, 443, 447
  link-wise, 155
  multireflection, 446, 487
  non-equilibrium bounce-back, 196, 251, 254
  partially saturated bounce-back, 447, 451
  wet-node, 155, 156, 200, 201
  with forces, 249, 261
Boussinesq approximation, 298, 313, 316
Brownian motion, 299, 440
Buckingham π theorem, 267, 287, 294
bulk density, 333
bulk phase, 333
buoyancy, 232, 298, 313, 317

Cahn-Hilliard equation, 364
Cassie-Baxter state, 400
Cauchy equation, *see* momentum equation
Chapman-Enskog analysis, 26, 106, 122, 127, 168, 208, 212, 217, 244, 248, 250, 306, 309, 655, 660
characteristics, 95, 522
  method of, 94, 96, 237, 665, 669
checkerboard instability, 36, 37, 40, 42, 53, 87
cluster, 537, 593
code examples, 533, 557, 578, 580, 648
collision, 12, 13, 15, 16, 19–22, 26, 28, 44, 45, 55, 66, 67, 70, 101, 103, 224, 411, 414
collision operator, 21, 22, 28, 44, 64
  BGK, 22, 64, 65, 98, 99, 101, 130, 133, 135, 143, 303, 323, 407, 425, 428
  cascaded, 409

T. Krüger et al., *The Lattice Boltzmann Method*, Graduate Texts in Physics,
DOI 10.1007/978-3-319-44649-3

cumulant, 409
entropic, 409
MRT, 112, 133, 135, 143, 147, 225, 408, 410, 411, 413, 423, 429, 658, 660, 669, 673
regularised, 409
TRT, 133, 135, 143, 147, 323, 408, 424, 427, 428
compiler, 535, 545
automatic optimisation, 545
compressibility
strong, 56, 111
weak, 54, 111, 126, 144, 149, 243
conservation
energy, 20, 21, 38, 48, 50, 51, 73, 74
mass, 4, 20, 21, 24, 38, 39, 41, 48, 50–52, 55, 73, 74, 99, 107, 434, 447
momentum, 5, 20, 21, 38, 39, 41, 48, 50–52, 73, 74, 99, 107, 434
conservation equation, 5, 23, 27, 38
conservation form, 5
material derivative form, 5
consistency
thermodynamic, 336
contact angle, 338
continuity equation, 4, 5, 23, 24, 32, 112, 117, 122, 124, 513
incompressible, 7, 10, 32, 314, 527
continuum, 3, 4, 11
convergence, 649
conversion factor, 266, 274, 283, 285
basic, 268
derived, 268
Couette flow, *see* flow
Courant number, 128
CPU
core, 541
critical point, 375
CUDA, 621
compilation, 627

DdQq, 63, 84
density
bulk, 333
energy, 17, 18, 314, 316
entropy, 27, 28
mass, 4, 17, 63, 83, 233, 239, 268, 271, 273, 280
mass flux, 4
momentum, 4, 17, 63, 83, 233, 239
momentum flux, 6, 24, 239
diffuse interface model, 339
diffusion flux, 300, 318, 321
diffusivity, 298, 300, 303
anisotropic, 300, 306, 310
thermal, 26, 36, 301
direct simulation Monte Carlo, 48, 51, 52
dissipative particle dynamics, 47, 48

energy equation, 9, 25, 32, 314, 316
Euler, 25
entropy, 9, 10, 27, 409, 519
equation of state, 8, 9, 11, 32, 124, 126, 334, 495
ideal gas, 9, 19
isentropic, 10, 11
isothermal, 10, 11, 82, 124
multicomponent free energy, 364
multicomponent Shan-Chen, 385
multiphase free energy, 348
multiphase Shan-Chen, 373
equilibrium, 25, 69, 222
distribution, 19, 21, 64, 69, 72, 77, 82, 92, 114, 120, 127, 298, 303, 306
moments, *see* moment
relaxation towards, 64, 99–101, 112
error
u^3, 111, 114, 119, 122, 143, 149, 658
compressibility, 144, 275
discretisation, 140, 275, 506
forcing, 247, 248
initialisation, 221, 224, 226, 228
iterative, 140
modelling, 143, 145
round-off, 139
truncation, 32, 136, 140, 142, 275
error norm, 138
Euler equation, *see* momentum equation
Euler scheme, 32, 33, 36, 96, 471, 525, 667
Eulerian system, 465. *see also* Lagrangian system, 466
exascale, 651

finite difference, 34, 38, 53, 136, 302
backward difference, 35, 36
central difference, 35, 36, 137
forward difference, 33, 35, 36, 136
finite element, 34, 41, 42, 53, 94
finite volume, 34, 38, 41, 53, 94
lattice Boltzmann method, 96
floating point, 538, 539, 543, 548, 550
double precision, 538
IEEE 754, 538
optimisation, 548, 550

flow
  Couette, 7, 8, 136
  incompressible, 7, 13, 123, 186, 268
  linear, 121, 123
  multicomponent, 332
  multiphase, 332
  Poiseuille, 8, 115, 254, 261, 283, 286, 326, 476
  Rayleigh-Bénard convection, 311, 312
  relativistic, 127
  shallow water, 127
  steady, 116, 117, 139, 142, 221
  Stokes, 123, 278
  Taylor-Green vortex, 221, 226, 539, 660, 662
  thermal, 311, 317
  unsteady, 221
  Womersley, 221, 287, 289
fluid
  immiscible, 333
  miscible, 333
fluid model
  Burnett, 26, 71
  Euler, 26, 522
  Navier-Stokes-Fourier, 26, 71, 84
force
  Shan-Chen, 370
force spreading, 466–468, 470, 480
forcing scheme, 233, 236, 240, 244, 423, 424
fresh node, 446, 450–452

Galilean invariance, xxii, 49, 143, 408
gas constant
  specific, 9, 316
GPU, 620
  bandwidth, 624
  block, 623, 624, 626, 644
  grid, 623, 624, 626
  kernel, 625
  memory, 624, 634
  multiprocessor, 622, 624, 644
  thread, 622–624, 626, 644
  warp, 622, 624, 644
graphics processing units, 620, 648
gravity, 232, 290, 312, 313, 316
Green's function, 503, 505, 514, 517
grid, 625

H-theorem, 27, 28, 50, 409
heat capacity, 9, 19
heat equation, 36, 301
heat flux, 25, 26
Helmholtz equation, 41, 501
Hermite expansion, 71, 74, 77, 80–82, 119, 234, 236
Hermite polynomials, 71, 74, 77, 80, 119, 235, 412, 415, 416, 662, 665
high performance computing, 537
hydrophilic surface, 338
hydrophobic surface, 338

ideal gas, *see* equation of state
immersed boundary method, 463, 487
  deformable boundaries, 483, 485
  direct-forcing, 478, 483
  explicit feedback, 474, 478
  implicit velocity correction-based, 480, 481
  multi-direct forcing, 482, 483
immiscible fluids, 333
index notation, 653, 655
initialisation, 67, 68, 154, 157, 220, 228, 249
interface model
  diffuse, 339
  sharp, 339
interface width, 339
interpreter, 535
isothermal fluid, *see* equation of state
isotropy
  lattice, 44, 46, 85, 87, 298, 306

kernel, 625, 626, 644
Knudsen number, 14, 26, 51, 106, 108–110, 280, 500, 655, 656

Lagrangian system, 465. *see also* Eulerian system, 466
Laplace pressure, 337
Laplace test, 342
lattice
  extended, 86, 124, 658
  isotropy, *see* isotropy
  projection, 85, 89
  pruning, 91
  reduced, 306
  vector, 86, 306
lattice Boltzmann equation, 64, 65, 97, 108, 239
lattice gas, 43, 47, 54, 62, 89
law of similarity, 14, 63, 265, 268, 279, 285
loop
  combining, 545
  optimisation, 543

peeling, 544
unrolling, 543
lotus effect, 400
lyophilic surface, 338
lyophobic surface, 338

Mach number, 13, 54, 111, 117, 123, 126, 144, 279
expansion, 71
machine epsilon, 538
macroscopic scale, 4, 12, 15
magic parameter, 134, 147, 322, 426
mass equation, *see* continuity equation
Maxwell area construction rule, 335
memory
access pattern, 567
bandwidth, 537, 551, 579, 592, 593, 624, 646
cache, 540, 551, 556, 567, 568, 572
cache line, 552
coalesced access (GPU), 634
distributed, 580
global (GPU), 624, 627, 644
local (GPU), 624
RAM, 540
register, 540
set associative cache, 553
shared, 579, 591
shared (GPU), 624, 627, 644
mesoscopic scale, 12, 15, 54
microscopic scale, 12, 15, 54
minimal surface, 337
miscible fluids, 333
Mlups, 563
molecular dynamics, 42, 43, 47
moment, 17, 19, 22, 23, 63, 70, 78, 83, 109, 113, 114, 410, 411
equilibrium, 93, 110, 113, 122, 124, 125, 307, 414, 415, 417, 421, 422, 670, 672
moment space, 409, 410, 413, 414
momentum density, *see* density
momentum equation
Cauchy, 6, 24
Euler, 6, 25, 106, 107, 110, 522
incompressible Navier-Stokes, 7, 10, 32, 34, 53, 55, 124, 139, 316, 496
Navier-Stokes, 5, 8, 14, 32, 54, 65, 107, 112, 117, 122, 247, 298, 418, 422
momentum exchange algorithm, 215, 218, 437, 459
monatomic gas, 16, 19, 21, 112, 657

MPI, 593, 620
blocking, 600
communication, 594, 600
compilation, 613
nonblocking, 600
rank, 594
reduction, 608
multi-particle collision dynamics, 48, 50
multicomponent flow, 332
multicomponent free energy
Bulk thermodynamics, 360
Chemical potential, 360
Equation of state, 364
Interfacial profile, 361
Lattice Boltzmann implementation, 364
Pressure tensor, 363
Surface tension, 361
Surface thermodynamics, 361
Wetting boundary condition, 361
multiphase
bottom-up, 341
top-down, 341
multiphase flow, 332
multiphase free energy
Bulk thermodynamics, 345
Chemical potential, 346
Equation of state, 348
Galilean invariance, 352
Interfacial profile, 346
Lattice Boltzmann implementation, 348
potential form, 351
pressure form, 351
Pressure tensor, 347
Surface tension, 347
Surface thermodynamics, 354
Wetting boundary conditon, 358
multiprocessor, 624
multirange force, 388

Navier-Stokes equation, *see* momentum equation
non-dimensionalisation, 14, 71, 73, 265, 271, 313, 314
non-equilibrium, 26, 65, 106, 119, 222, 224
distribution, 118, 119

OpenCL, 621
OpenMP, 580, 581, 593
clause, 581
compilation, 587

directive, 581
parallel block, 583
private variables, 585
reduction, 586
shared variables, 585
order parameter, 332

Péclet number, 301, 322, 325, 327
parallel computing, 578
strong scaling, 616
weak scaling, 616
parasitic currents, 387
pointer, 547, 548, 679
Poiseuille flow, *see* flow
Poisson equation, 34, 55, 127, 222
polyatomic gas, 16, 18, 502
polynomial evaluation, 542
population space, 409–411, 414
post-processing, 576
Prandtl number, 22, 301, 315, 502
pressure, 5, 11, 18, 53, 125, 222, 232, 273
Laplace, 337
pressure tensor, 340, 342
multicomponent free energy, 363
multiphase free energy, 347
processor
core, 541, 580
multi-core, 541, 580
profiling, 541
timing, 556
programming language
assembly language, 549
C, 536, 548
C++, 536, 548
compiled, 535–537
Fortran, 536, 548
interpreted, 535, 536
MATLAB, 535
Python, 535
pseudopotential, 369

quadrature
Gauss-Hermite, 80, 81, 83, 84, 662, 665

Rayleigh number, 312
relaxation matrix, 411, 418, 421, 423
relaxation rate, 408, 411, 425
relaxation time, 22, 99, 112, 143, 271, 408, 669
viscous, 496
rest velocity, 84, 306
Reynolds number, 13, 14, 128, 134, 269, 278, 281, 282, 407, 418
grid, 277, 278

scaling
acoustic, 276, 279
diffusive, 119, 145, 275, 278
Schmidt number, 301
Shan-Chen force, 370
Shan-Chen model, 368
sharp interface model, 339
smallness parameter, 26, 107, 114, 116
smoothed-particle hydrodynamics, 47, 52, 53
solvability conditions, 107, 115, 117, 245, 307
sound
absorption, 502, 503
attenuation, 494, 498, 500, 502, 503, 506, 507, 510
dispersion, 498, 500, 506, 507, 509, 510
generation, 512, 515
speed, 11, 13, 64, 84, 92, 125, 272, 276, 496, 500
sound wave
complex notation, 497, 498
cylindrical, 504
forced, 499, 509
free, 499, 507, 510
impedance, 503, 521
multipole, 505, 515, 518
non-linear, 506
plane, 497, 498, 504, 506
spherical, 504
speed of sound, *see* sound
spurious currents, 387
stability, 85, 86, 96, 97, 100, 101, 112, 127, 136, 276, 277, 310
stability condition
necessary, 129
optimal, 129, 131
sufficient, 129, 130
stability map, 128, 131–133
staircase approximation, 164, 181, 201, 436
steady flow, *see* flow
streaming, 45, 66–68, 101, 103, 224
stress
normal, 6
shear, 6, 215, 217, 453, 455
stress tensor, 6, 7, 24, 215, 274, 453
Lighthill, 514
viscous, 6, 26, 65, 67, 107, 111, 224, 233, 495
Strouhal number, 288
supercomputer, 593

supercomputers, 621
superhydrophobic, 400
superhydrophobic surface, 339
surface
  minimal, 337
surface tension, 336, 342
  multicomponent free energy, 361
  multiphase free energy, 347
  multiphase Shan-Chen, 375
surfactants, 393

thermal diffusivity, *see* diffusivity
thermal flow, *see* flow
thermal fluctuations, 46, 48, 50, 56, 312, 440
thermodynamic consistency, 336
thread, 580–582, 593, 625
time scale
  acoustic, 13, 288, 656
  advective, 13, 288, 656
  between collisions, 13, 15
  diffusive, 13, 288
traction, 158, 215, 453
transformation matrix, 410, 412, 414, 417, 420, 421

units
  conversion, 265
  lattice, 63, 265, 266, 270–272, 274, 416, 500
  physical, 63, 270, 271, 274
upwind scheme, 36, 40

velocity
  barycentric, 382
velocity interpolation, 466–468, 471, 473, 480
velocity set, 63, 64, 84, 93, 113, 664, 665
viscosity, 6, 143, 272, 273, 303
  bulk, 7, 65, 112, 126, 418, 422, 423, 500
  shear, 7, 65, 112, 418, 422, 427, 500
visualisation, 576
von Neumann analysis, 128, 132, 429

warp, 634
wave equation
  ideal, 496
  time-harmonic, *see* Helmholtz equation
  viscous, 496
weight function, 71, 74, 78, 80, 662
Wenzel state, 401
wet-node, *see* boundary scheme
Womersley number, 287

Young-Laplace test, 342, 379

Zou-He scheme, *see* boundary scheme → non-equilibrium bounce-back