Received: 28 June 2019 | Revised: 21 January 2020 | Accepted: 17 March 2020

DOI: 10.1002/fld.4835

RESEARCH ARTICLE

WILEY

Efficient viscosity contrast calculation for blood flow simulations using the lattice Boltzmann method

Moritz Lehmann | **Sebastian Johannes Müller** | **Stephan Gekle**

Biofluid Simulation and Modeling – Theoretische Physik VI, Universität Bayreuth, Bayreuth, Germany

Correspondence
Moritz Lehmann, Biofluid Simulation and Modeling – Theoretische Physik VI, Universität Bayreuth, 95448 Bayreuth, Germany.
Email: moritz.lehmann@uni-bayreuth.de

Summary

The lattice Boltzmann method (LBM) combined with the immersed boundary method is a common tool to simulate the movement of red blood cel ls (RBCs) through blood vessels. With very few exceptions, such simulations neglect the difference in viscosities between the hemoglobin solution inside the cells and the blood plasma outside, although it is well known that this viscosity contrast can severely affect cell deformation. While it is easy to change the local viscosity in LBM, the challenge is to distinguish whether a given lattice point is inside or outside the RBC at each time step. Here, we present a fast algorithm to solve this issue by tracking the membrane motion and computing the scalar product between the local surface normal and the distance vector between the closest LBM lattice point and the surface. This approach is much faster than, for example, the ray-casting method. With the domain tracking applied, we investigate the shape transition of a RBC in a microchannel for different viscosity contrast and validate our method by comparing with boundary-integral simulations.

KEYWORDS

lattice Boltzmann Method, immersed boundary, viscosity contrast, red blood cell, microchannel

1 | INTRODUCTION

Red blood cells (RBCs) flowing through small blood vessels or microchannels show a fascinating wealth of flow states including steady shapes, dynamic states where the membrane periodically rotates around the cell interior, or tumbling motions.[1-12] Experimental techniques to visualize these flow states are still mostly limited to two-dimensional (2D) video microscopy,[13-16] although progress toward three-dimensional (3D) imaging techniques has recently been made.[17] A lot of insight into the flow behavior of RBCs is therefore gained by computer simulations.

A RBC consists of a thin elastic membrane surrounding the interior hemoglobin solution which to a good approximation can be viewed as a Newtonian liquid with a viscosity about five times larger than the surrounding blood plasma.[10,18] This viscosity contrast λ is essential for the RBC dynamics.[19-26] Depending on the numerical technique, it can be more or less tedious to include the parameter λ into numerical simulations. In boundary-integral methods (BIM), the consideration of a viscosity contrast is conceptually straightforward, although computationally costly.[27-30] Particle methods such as smoothed dissipative particle dynamics (SDPD) are able to include viscosity contrast,[7] although this is not always done.[5,31]

In finite volume methods, a viscosity contrast has recently been included using an indicator function advected with the fluid.[32] One of the most widely used techniques in blood flow simulations is the combination of the Eulerian lattice Boltzmann method (LBM)[33] for the flow and the Lagrangian immersed boundary method (IBM)[34-37] for the fluid-structure interaction with the RBC membrane. Although it is simple to locally change the fluid viscosity in LBM, the key difficulty is to determine whether or not a given LBM lattice point is located inside or outside a RBC in order to assign the correct viscosity. A standard solution to such inside/outside problems consists of tracing a beam originating from the point of interest and counting the number of RBC membrane crossings (ray-casting). Carrying out this determination for every time step during a simulation, where cells move and deform dynamically, would clearly make the ray-casting approach far too computationally expensive (except in 2D[38]). In 3D, cluster algorithms[39] and volume-of-fluid like methods[40] can be expected to be relatively costly. Some recent works[39,41,42] therefore raised the general idea of tracking membrane vertices in order to determine the interior volume of a moving cell. Nevertheless, the vast majority of LBM-IBM simulations of RBC flow still neglect viscosity contrast altogether.

Here we provide a detailed description and systematic analysis of such a tracking algorithm to include viscosity contrast of red blood (and other) cells into LBM-IBM simulations. Given an initial configuration in which the inside/outside status of each LBM lattice point is known, the first step of the algorithm identifies those LB points which could potentially, due to motion/deformation of the RBC membrane, switch from inside to outside or vice versa. This task is greatly simplified in the present case since for LBM-IBM methods the typical distance between membrane vertices is kept similar to the LBM grid distance even during large deformations of the cell, as we verified by postprocessing a large set of our existing simulation data. It thus proves sufficient to consider only those LBM points which are in the immediate vicinity of membrane vertices. In the second step, geometrical considerations allow us to determine for each LBM lattice point whether or not the inside/outside flag needs to be switched for the next time step. Every few hundred time steps an efficient heuristic correction step is carried out to remove spurious errors. We thus obtain a highly accurate and highly parallelizable tool for RBC simulations with viscosity contrast using the LBM-IBM approach. We validate our tool by studying the croissant-slipper transition for a RBC in a rectangular microchannel as a function of λ for which we find very good agreement with highly resolved BIM simulations.[11]

2 | METHODS

2.1 | Lattice Boltzmann for the fluid

The LBM[33,43] is a powerful mesoscopic fluid solver. The fluid is represented by a discrete set of particle populations f_i along fixed directions $\vec{c}_i$ located on a Cartesian lattice. For concurrency reasons, two copies f_i^c and f_i^s need to be stored in memory for the collision and streaming steps. The so-called equilibrium populations f_i^{eq} are only temporarily held in register. ρ and $\vec{u}$ denote the density and velocity of the fluid for every lattice point and $c_s = \frac{1}{\sqrt{3}}\frac{\Delta x}{\Delta t}$ is the lattice speed of sound with the grid distance being denoted as Δx and the time step as Δt. In each time step, the populations stream into neighboring lattice points where they collide and are redistributed into the streaming directions for the next step. In a nutshell, LBM can be written down as just five equations:

1. Streaming (pull method)

$$f_i^s(\vec{x}, t) = f_i^c(\vec{x} - \vec{c}_i \Delta t, t). \tag{1}$$

2. Collision (MRT operator)

$$\rho(\vec{x}, t) = \sum_i f_i^s(\vec{x}, t) \quad \vec{u}(\vec{x}, t) = \frac{1}{\rho} \sum_i \vec{c}_i f_i^s(\vec{x}, t). \tag{2}$$

$$f_i^{eq}(\vec{x}, t) = w_i \rho \cdot \left(\frac{\vec{u} \circ \vec{c}_i}{c_s^2} + \frac{(\vec{u} \circ \vec{c}_i)^2}{2c_s^4} + 1 - \frac{\vec{u} \circ \vec{u}}{2c_s^2} \right). \tag{3}$$

$$f_i^c(\vec{x}, t + \Delta t) = f_i^s(\vec{x}, t) - \left(M^{-1} S \left(M f^s(\vec{x}, t) - M f^{eq}(\vec{x}, t) \right) \right)_i. \tag{4}$$

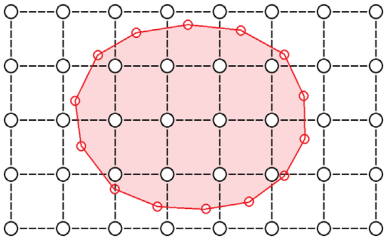

FIGURE 1 Two-dimensional illustration of the immersed boundary method. Membrane vertices can be located anywhere in between lattice points [Color figure can be viewed at wileyonlinelibrary.com]

In Equation (4) M is a transformation matrix into moment space and S is a diagonal matrix containing all relaxation times. The kinematic shear viscosity ν of the simulated fluid is

$$\nu = c_s^2 \left(\tau - \frac{\Delta t}{2} \right). \tag{5}$$

The relaxation time τ can be different at every lattice point, which makes it possible to change ν locally in space. The tracking algorithm presented below uses this possibility to implement a viscosity contrast between the interior and the exterior of a flowing cell according to a flag lattice.

Our simulations are based on the LBM implementation of the simulation package ESPResSo[44-46] which uses a multi-relaxation-time collision operator and halfway bounce-back conditions for the solid boundaries. With an additional volume force term in the collision operator following the Guo scheme,[47] a persisting flow is created.

2.2 | IBM for cell membranes

The IBM[34,48,49] enables the Lagrangian movement of a tesselated membrane along with the LBM velocity field and couples back membrane forces into the fluid. The membrane vertices can move freely between lattice points (Figure 1) and their movement is coupled to the lattice in two ways: To obtain the velocity for advecting a membrane vertex and the velocity of the surrounding lattice points is interpolated. Then the elastic forces between membrane vertices are calculated (see below) and the force for each vertex is spread across all nearby lattice points as an additional local volume force term in LBM. A necessary requirement for this two-way coupling is that the distance between membrane vertices is at the same scale as the distance between neighboring lattice points in order to prevent "holes" in the membrane for too large and bad velocity interpolation for too small vertex spacing.

2.3 | RBC model

To illustrate an application of our tracking algorithm, we will present below investigations of the behavior of a RBC with different viscosity contrasts flowing in a microchannel. Figure 2 visualizes how we generate the RBC surface by recursively splitting the faces of an icosahedron, following Loop's subdivision surface scheme.[50] The triangle vertices are ordered such that all surface normals point outwards, which is a requirement of our tracking algorithm.

For the RBC membrane mechanics, we employ the standard model described in more detail in References 11 and 18 and many other works. Briefly, elastic forces arising from membrane deformation are due to shear elasticity, area dilatation and bending forces. The former two are modeled via the empirical Skalak law[51] while bending forces are computed from the Helfrich model using the method denoted "B" in References 52 and 53, originally developed by Gompper and Kroll.[54]

2.3.1 | Strain energy

For a small element of a 2D membrane with the dimensions dx_1 and dx_2 along the x_1 and x_2 axes, the expansion ratios

$$\lambda_1 := \frac{dy_1}{dx_1}, \quad \lambda_2 := \frac{dy_2}{dx_2}, \tag{6}$$

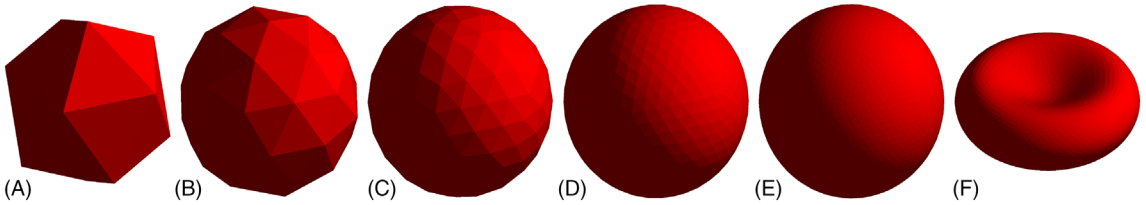

FIGURE 2 Generation of the tesselated red blood cell surface visualized. The shape starts as an icosahedron in (A), then in (B) to (E) the triangles are split recursively to increase resolution and finally the top and bottom caps of the sphere are dented inwards to produce the characteristic RBC shape in (F). The sphere in (E) and the RBC in (F) have 5120 triangles each [Color figure can be viewed at wileyonlinelibrary.com]

are defined. The strain invariants I_1 and I_2 then are

$$I_1 = \lambda_1^2 + \lambda_2^2 - 2, \quad I_2 = \lambda_1^2\,\lambda_2^2 - 1, \tag{7}$$

and the strain energy E_S following the Skalak model[51] only depends on these invariants:[18,55]

$$E_S = \frac{B}{4}\left(\frac{1}{2}I_1^2 + I_1 - I_2\right) + \frac{C}{8}I_2^2. \tag{8}$$

The constants B and C are material properties of the membrane.

2.3.2 | Bending forces

The idea is to calculate the bending force $\vec{F}$ for each membrane vertex (i) at position $\vec{x^{(i)}}$

$$\vec{F}(\vec{x^{(i)}}) = -\frac{\partial E_B}{\partial \vec{x}^{(i)}}, \quad i = \{1, \ldots, N\}, \tag{9}$$

from the Helfrich bending energy E_B[53,56]

$$E_B = 2\,\kappa_B \int_S (H(\vec{x}))^2 dS(\vec{x}) \approx \frac{\kappa_B}{2}\sum_{i=1}^{N}\left(2\,H(\vec{x}^{(i)})\right)^2 A^{(i)}_{\text{Voronoi}}, \tag{10}$$

whereby κ_B denotes the bending modulus, S denotes the instantaneous smooth surface, and N denotes the number of membrane vertices. The local mean curvature H of the RBC surface is calculated as

$$H(\vec{x}) = \frac{1}{2}\sum_{i=1}^{3}(\Delta_S x_i)\,n_i(\vec{x}), \quad \vec{x} \in S, \tag{11}$$

with the approximation for the the Laplace-Beltrami operator Δ_S of Gompper and Kroll:[52,54]

$$\Delta_S x_l^{(i)} \approx \frac{\sum_{j(i)}(\cot\,\vartheta_1^{(ij)} + \cot\,\vartheta_2^{(ij)})(x_l^{(i)} - x_l^{(j)})}{2\,A^{(i)}_{\text{Voronoi}}}, \quad i = \{1, \ldots, N\}, \quad l = \{1, 2, 3\}, \tag{12}$$

where the superscript (j) denotes the index of membrane vertices adjacent to the membrane vertex (i) and $A^{(i)}_{\text{Voronoi}}$ is the area of the Voronoi cell containing (i) as illustrated in figure 3:

$$A^{(i)}_{\text{Voronoi}} := \frac{1}{8}\sum_{j(i)}(\cot\,\vartheta_1^{(ij)} + \cot\,\vartheta_2^{(ij)})|\vec{x^{(i)}} - \vec{x^{(j)}}|, \quad i = \{1, \ldots, N\}. \tag{13}$$

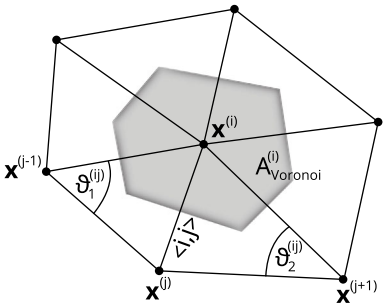

FIGURE 3 A sketch of a membrane vertex $\vec{x}^{(i)}$ with six neighbors to illustrate the naming conventions [Color figure can be viewed at wileyonlinelibrary.com]

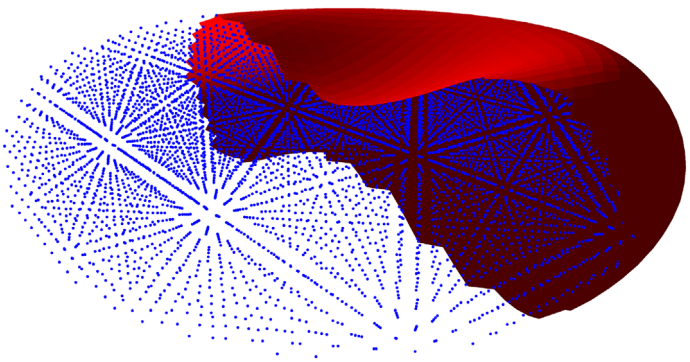

FIGURE 4 A cut-open red blood cells on the Boltzmann method lattice with all points on the inside marked blue [Color figure can be viewed at wileyonlinelibrary.com]

3 | ALGORITHM FOR TRACKING INSIDE AND OUTSIDE

The goal of the following algorithm is to calculate a flag lattice with the same dimensions as the LBM lattice which for each lattice point contains the *boolean* information whether the lattice point currently is inside or outside a cell. Our method consists of three steps: (i) an initialization step where the flag lattice is filled once depending on a prescribed initial geometry, (ii) a highly efficient update step where only the LBM points neighboring the cell membrane are evaluated, and (iii) a correction step employed every few hundred steps which removes spurious artifacts introduced by the update step using a simple set of heuristic rules. Figure 4 gives an impression on how large the LBM domain for a RBC typically is.

3.1 | Initialization step

If the geometry of the cell is known, the analytic condition for the surface can be applied to every lattice point. For example, a point with the coordinates x, y, z is within a sphere of radius r if $x^2 + y^2 + z^2 \leq r^2$. For a RBC with larger radius R, a similar condition[57] has been formulated based on experimental data:

$$r^2 := \frac{x^2 + y^2}{R^2} \quad c_0 := 0.207 \cdot R \quad c_2 := 2.00 \cdot R \quad c_4 := 1.12 \cdot R. \tag{14}$$

$$x^2 + y^2 + z^2 \leq R^2. \tag{0}$$

$$z^2 \leq \frac{1}{4}\left(c_0 + c_2 \cdot r^2 - c_4 \cdot r^4\right)^2 \cdot (1 - r^2). \tag{15}$$

This approach exhibits the same efficient scaling of the total compute time t with the LBM lattice size N ($t \sim N^3$) as the LBM itself.

It is also possible to load an initially deformed state of the RBC (no analytic condition) from a previously generated checkpoint file. These checkpoint files are periodically generated during simulation and stored on the hard drive in case there is a crash or power outage. If no analytic condition for the initial geometry is known, we use a ray-cast-based algorithm (also known as *crossing number algorithm*[58]). As this method is only used once for initialization, its slow

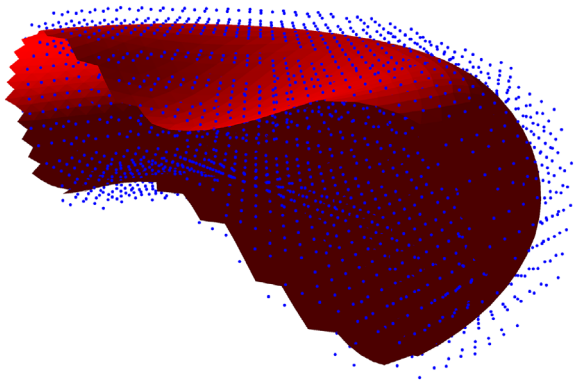

FIGURE 5 In the update step, only the lattice points next to the surface are considered (radius of lattice points around each membrane vertex $r_{\text{shell}} = 1$). These points are located both inside and outside of the cell surface. The image shows a cell (cut in half for visualization) with the nearest lattice points to the surface marked as blue dots. Only for this shell of lattice points the decision needs to be made. The state of all other points which are far away from the cell surface has already been determined by either the initialization or the previous update steps [Color figure can be viewed at wileyonlinelibrary.com]

performance does not affect the overall speed of the simulation. For every lattice point, a ray from the lattice point in an arbitrary direction* is generated. Then the number of ray-triangle intersections is calculated using the Möller-Trumbore intersection algorithm.[59] If this number is odd, the lattice point is inside the cell, otherwise it is outside. This approach is very calculation intensive: it scales with the number of LBM lattice points (N^3) times the number of triangles T. The number of triangles is directly proportional to the total surface area of the RBC. Due to the requirement that the distance between two neighboring points on the RBC surface should be approximately the same as the distance between two neighboring points on the LBM lattice, we have $T \sim N^2$. The total scaling for the ray-casting approach is thus $t \sim N^3 T \sim N^5$.

AC++ implementation of the initialization with ray-casting is listed in Appendix A.

3.2 | Update step

Because of their bad performance and bad parallelizability, ray-casting algorithms are only suitable for initialization and not for the regular update happening at each time step. For the update step to be more efficient, we therefore implement a membrane tracking procedure as follows. In each time step, we only consider LBM lattice points next to the cell surface (Figure 5). For these points, we calculate the distance vector to the RBC surface and compare it to the normal vector on the surface via a scalar product, resulting in a negative or positive value discrimnating inside from outside.

Four arrays need to be allocated once at simulation startup. These store the normal vector for each membrane vertex (A), a list of all lattice points which are close to the surface and whose inside/outside flag may need to be updated (update list, (B)) as well as the index and distance of the closest membrane vertex ((C) and (D), respectively) for each lattice point. The algorithm consists of three consecutive loops:

1. In the first loop, we compute for each membrane vertex the normal vector as the average of the normal vectors of all adjoining triangles weighted by their area and store it in array (A).
2. The purpose of the second loop is to fill the update list (B), that is, to find a shell of lattice points located around the membrane. Therefore, for each of the membrane vertices the eight closest lattice points ($r_{\text{shell}} = 1$) are determined via integer casting. The distance from each of these to the membrane vertex is calculated. If this distance is smaller than the distance stored in the array (D) entry for the lattice point, the distance in array (D) is updated, the membrane vertex index is stored in array (C) and the 3D position indices $i, j,$ and k of the lattice point are stored in a the update list (B). (B) may contain some lattice points more than once, which is not a problem however, because the last entry for a given point will always be the one of the closest membrane vertex.
3. The third loop goes through the update list (B). The indices of the lattice points and their closest membrane vertices are fetched from (B) and (C). Then, the vector from the membrane vertex to the lattice point and the previously calculated

*Mathematically, the direction of the ray is arbitrary. However, if the ray passes exactly through the edge of two adjacent triangles, the intersection count is increased by two instead of one, which leads to the algorithm failing. The solution to the problem is statistical exclusion by pointing the ray in a direction that cannot be represented by floating point numbers. For example, a ray in the direction $(1.040.030.01)^T$ will never intersect the edge of a triangle, while a ray in the direction $(100)^T$ probably will. Alternatively, multiple rays in different directions can be computed.

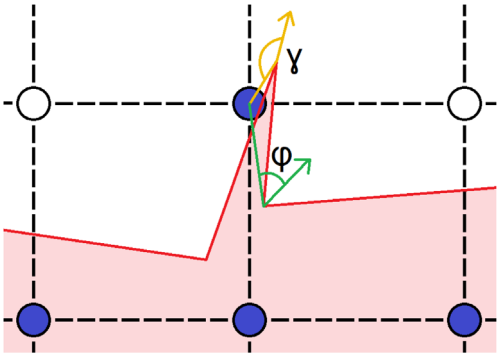

FIGURE 6 The special case where our algorithm fails visualized in two dimension. A point is located below the plane perpendicular to the closest vertex normal (yellow line, $\gamma > 90°$) and therefore activated even though it is actually located outside of the cell volume. This can only occur if the angle between triangles is small, otherwise the correct vertex (green line with the angle φ) would be closer [Color figure can be viewed at wileyonlinelibrary.com]

normal vector of the membrane vertex from (A) are compared via scalar product. A negative result defines the lattice point as inside the cell.

The main advantage of our proposed method is its computational speed and parallelizability. Our method is solely based on the knowledge of the surface geometry and also works when only parts of the surface are known in different simulation domains, which makes it ideal for multi-CPU parallelization. The update step scales with $t \sim (2r_{\text{shell}})^3 T \sim N^2$, which is considerably faster than LBM and thus does not impose any notable performance penalty on the simulations. In Appendix B a C++ implementation as used in ESPResSo is listed. The only prerequisite is that the maximum distance between membrane vertices (using a $r_{\text{shell}} = 1$) should be smaller than two times the lattice constant in order to prevent holes. As pointed out above, this condition is automatically ensured for almost all membrane vertices by the LBM-IBM algorithm. Furthermore, the vertex indices of all triangles must be ordered such that all surface normals point outwards.

In the rare case of an extremely crumpled surface, when the angle between neighboring triangles is smaller than 90° , our algorithm can sometimes fail (see Figure 6 for details). Moreover, points outside of the cell which have falsely been "activated" (believed to be inside) due to this error may stay activated if the cell has moved away and the point is out of reach of the algorithm. This can result in the cell dragging a tail of activated points behind. To remove these spurious artifacts, an additional correction step every few hundred regular steps is required as described in Section 3.3 below.

Variants of the algorithm with a wider radius $r_{\text{shell}} > 1$ of lattice points around the cell membrane have also been tested, for example using the closest 4^3 points instead the closest 2^3 points to a membrane vertex. A wider radius of lattice points vastly slows down the algorithm, as more points need to be processed. In addition, the lattice points are then further away from the surface, increasing the risk of failure due to the surface curvature. A wider radius is only useful if the cells membrane is triangulated sparsely compared to the lattice point density, in which case $r_{\text{shel}\,l} = 1$ would result in an excessive amount of holes. Given a sufficient membrane vertex density though, one can avoid the large computational overhead of a wider radius.

There is another similar variant, *update via face normals*, which in the second loop instead of membrane vertices uses the centers of the triangles with their direct normal vectors. This variant however is much more prone to errors when the surface is crumpled, since there the normal vectors are used directly and are not averaged over five to six triangles. Furthermore, although the scaling of $t \sim N^2$ is the same, it is only about half as fast compared to the *update via vertex normals* described above, because there are only about half as many membrane vertices as triangles.

3.3 | Correction step

Since the update step from Section 3.2 occasionally creates artifacts as described in Figure 6, an additional correction step is required to run once every few hundred regular steps. The correction algorithm loops through all lattice points of the inside/outside flag lattice and for each point counts the number of "activated" (believed to be inside) neighbors. Any given point can have a maximum of 26 activated neighbors. Our correction algorithm (i) detects activated lattice points with too few active neighbors and deactivates them, as well as (ii) detects deactivated points with too many active neighbors and activates them. The reason for this is the assumption that the surface – or the boundary between activated and deactivated lattice points—is locally smooth, so an activated lattice point on the boundary ideally has not much less than 13 activated neighbors and a deactivated point on the boundary has not much more than 13 activated neighbors. Figure 7 illustrates both possible corrections.

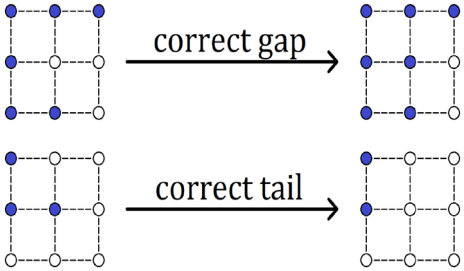

FIGURE 7 The correction step activates deactivated points with many activated neighbors (gaps) and deactivates activated points with few neighbors (tails) [Color figure can be viewed at wileyonlinelibrary.com]

Case	Max	a	d
inside	26	8(30, 8%)	18(69, 2%)
side	17	5(29, 4%)	12(70, 6%)
edge	11	3(27, 3%)	8(72, 7%)
corner	7	2(28, 6%)	5(71, 4%)

TABLE 1 Thresholds for the minimum number of activated (a) and maximum number of deactivated neighbors (d) for different lattice point locations (for details see main text)

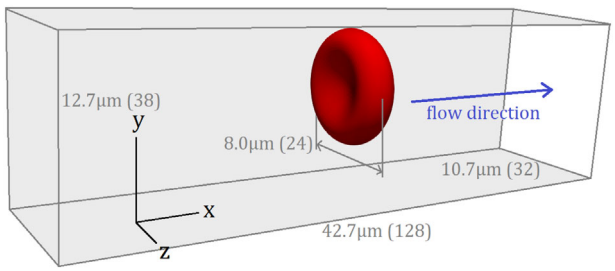

FIGURE 8 The simulation setup shown from different perspectives. A red blood cell is placed vertically in a rectangular channel. The boundaries in x direction are periodic. The numbers in brackets represent the length in number of lattice points [Color figure can be viewed at wileyonlinelibrary.com]

We impose that activated points need at least $a = 8$ activated neighbors to stay activated, which is approximately 30 % of the maximum number of neighbors. Deactivated points need less than $d = 18$ activated neighbors to stay deactivated, which is approximately 70 % of the maximum number of neighbors. The thresholds are chosen empirically with test runs so that the effectiveness of the error correcting step is maximized. If the thresholds were much lower, too few corrections would occur. If the thresholds were higher, the true cell surface would become eroded.

For lattice points on the side, edge or in the corner of the simulation box (or the local CPU domain), the maximum number of neighbors available in local memory is lower. In these cases, the thresholds are scaled down linearly with the maximum neighbor count, which is equivalent to extrapolating the missing neighbors. This avoids the overhead of having to implement a halo and communication between individual CPU nodes for the flag lattice. The thresholds are shown in Table 1.

The correction step scales with $t \sim N^3$, but since it is only executed every few hundred regular update steps, this is not of concern. The correction step also counts the number of changed lattice points in order to keep track of the errors.

4 | APPLICATION: A RBC IN A RECTANGULAR CHANNEL

4.1 | Setup

The simulation setup consists of a single RBC with $R = 4\,\mu m$ in Equations (14) and (15). The RBC is placed vertically in a rectangular channel with dimensions $L_x = 42.7\,\mu m$, $L_y = 12.0\,\mu m$ and $L_z = 10.0\,\mu m$ (periodic in x) as in our previous work[11] and as shown in Figure 8. The initial RBC position is slightly off center in the y and z direction ($y_{\text{initial}} = 1.50\,\mu m$, $z_{\text{initial}} = 0.833\,\mu m$). The fluid moves in x direction at an average flow velocity of $v_{\text{avg}} = 1.5$ mm/s by imposing a volume force (pressure gradient). The simulation time is 9.26 seconds (60 Million integration steps). Figure 9 shows snapshots of the first 0.2 % of the simulation for $\lambda = 3$, in which the cell crosses the periodic boundary for the first time.

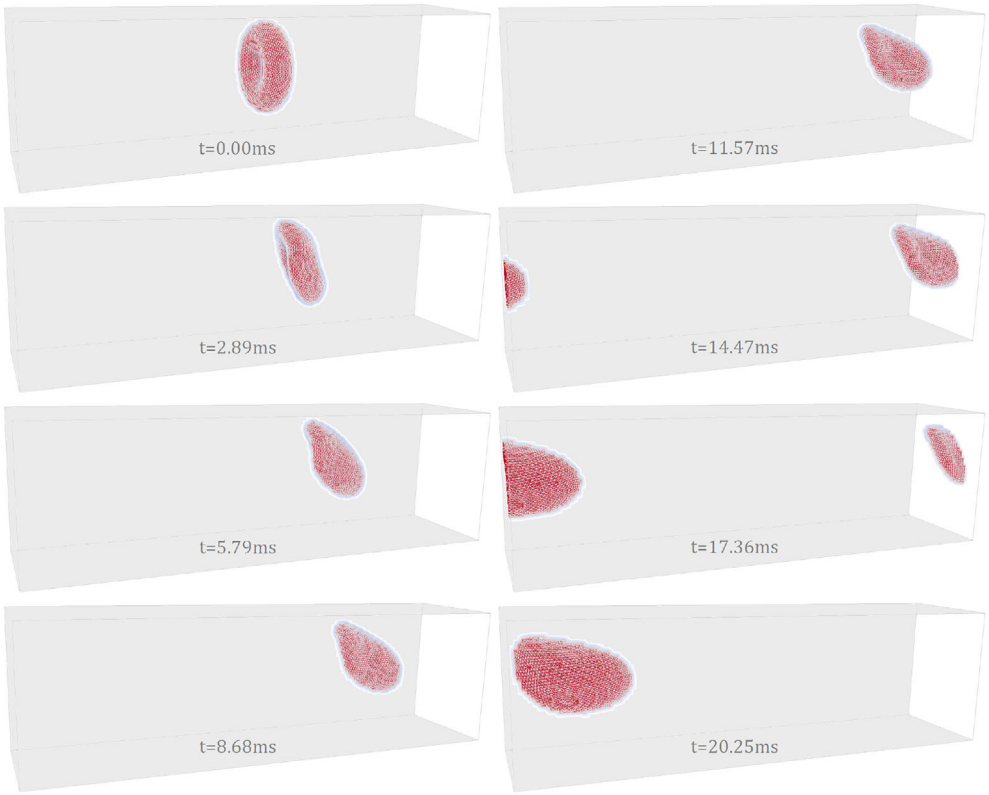

FIGURE 9 Eight snapshots of the simulation with $\lambda = 3$ at a distance of 18 750 integration steps or 2.89 ms each. The cell surface is visualized as a gray wire frame and the inside/outside data is rendered volumetrically in red. The volume tracking works very accurately, even when the cell is cut in half while crossing the periodic boundaries [Color figure can be viewed at wileyonlinelibrary.com]

Depending on the viscosity contrast λ, there is a shape preference for either the "slipper" (elongated shape, asymmetric position in channel, rotating surface) or the "croissant" (contracted shape, symmetric position in channel center, stationary). For low values of λ, croissants are preferred while large values of λ lead to slipper states. Comparing the transition point to a recent set of BIM simulations,[11] which can naturally and exactly deal with inside/outside viscosity contrasts, will validate our algorithm.

4.2 | Results

Figure 10 depicts the different behavior of the RBC when λ is varied between 1 and the physiological value of approximately 5. Besides visual inspection, there are two quantitative indicators for the cell shape: center of mass radial displacement d and asphericity a. Both are scalar values that change over time. Figure 11A shows the radial displacement $d := \sqrt{\Delta y^2 + \Delta z^2}$ of the center of mass from the middle of the rectangular channel over time for two different values of λ. The two distinct stable cell shapes are represented by either the graph dropping to zero (croissant) or the graph oscillating around an offset (slipper). Figure 11B shows the asphericity—a scalar value indicating how nonspherical the surface is. The asphericity is, according to Fedosov et al,[5] defined as follows: First, the center of mass $\vec{\bar{x}}$ is calculated from all N membrane vertices at locations $\vec{x_i}$.

$$\vec{\bar{x}} := \frac{1}{N} \sum_{i=0}^{N} \vec{x_i}. \tag{17}$$

Then, we define the gyration tensor S by Equation (18).

$$S := \frac{1}{N} \sum_{i=0}^{N} (\vec{x_i} - \vec{\bar{x}})(\vec{x_i} - \vec{\bar{x}})^T. \tag{18}$$

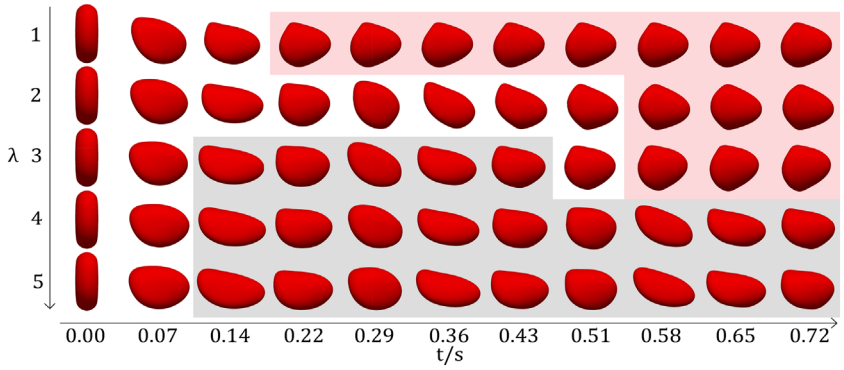

FIGURE 10 The cell geometries resulting from different values of λ depending on the simulated time. The cells with pink background are marked as converged croissants while a gray background indicates the oscillating slipper state. In the small time frame shown here, the cells at $\lambda \in \{4, 5\}$ converge to a stable oscillation which is caused by the cell *tank-treading* in the flow [Color figure can be viewed at wileyonlinelibrary.com]

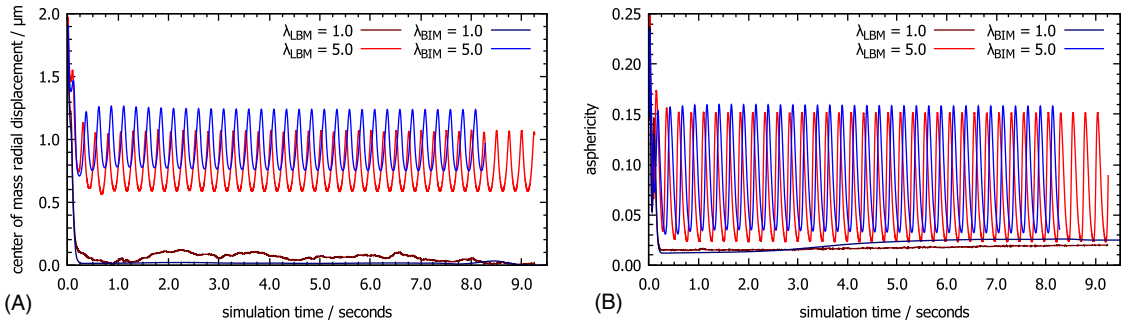

FIGURE 11 (A) Radial displacement of the cell center of mass from the middle of the rectangular channel and (B) asphericity over time for different values of λ . Cells with low values of λ migrate to the channel center (croissant) while at $\lambda = 5$ the cells maintain stable *tank-treading* which is a characteristic pattern for the slipper state. The datasets for boundary-integral methods show good matching with our inside/outside method, except for a small difference in phase for the slippers. Note that for the croissant shape the asphericity converges to a nonzero value, since the cell shape is not completely spherical [Color figure can be viewed at wileyonlinelibrary.com]

Finally, with λ_x^2, λ_y^2, and λ_z^2 being the eigenvalues of S, the asphericity a is defined by Equation (19).

$$a := \frac{(\lambda_x^2 - \lambda_y^2)^2 + (\lambda_y^2 - \lambda_z^2)^2 + (\lambda_z^2 - \lambda_x^2)^2}{2\,(\lambda_x^2 + \lambda_y^2 + \lambda_z^2)}. \tag{19}$$

As can be seen, both the LBM and BIM graphs for $\lambda = 1$ quickly converge to zero for the radial displacement and to a small constant offset for the asphericity. The LBM and BIM graphs for $\lambda = 5$ show pronounced oscillations which are caused by the cell membrane continuously rotating around the cell interior (so-called *tank-treading*) and the offset from zero indicates that the cell is located asymmetrically in the channel. LBM and BIM differ only slightly in offset and phase of the oscillation, while the oscillation frequency and amplitude are almost the same. Possible explanations for this difference are that the exact flow rate in LBM might mismatch by a few percent compared to BIM or that in the BIM simulations a different cell surface tessellation algorithm with only 2048 triangles is utilized.

Figure 12 shows the averages over the last 0.2 s econds, which is approximately the period of cell rotation, of the radial positions from Figure 11A. The resulting diagram represents the phase change of the RBC at $\lambda \approx 4.75$, where the RBC changes from croissant ($\lambda < 4.75$) to slipper ($\lambda > 4.75$) in good agreement between BIM and our LBM tracking algorithm.

Every LBM data point in Figure 12 corresponds to approximately 2 weeks of compute time on 16 cores of two Intel Xeon E5-2680 CPUs with the fast tracking algorithm of Section 3.2. When instead using the ray-cast algorithm for every lattice point (N^5 scaling) in every simulation time step, the compute time for the same simulation would be approximately three years. With the ray-casting algorithm only applied for the points close to the surface (N^4 scaling) in every time step, compute time would be 4 months. However due to the parallelization of the IBM in the ESPResSo simulation package, in multi-CPU parallelization for any CPU core only part of the cell membrane is known, making the ray-cast-based algorithms very difficult to parallelize. The comparison of compute times instead is done with the simulation executed on only a single CPU core and the compute time is extrapolated to what it would be on 16 cores.

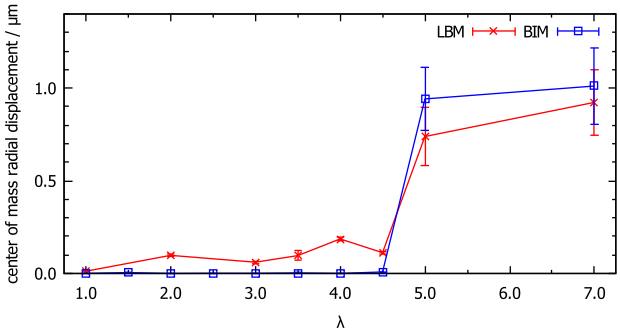

FIGURE 12 The center of mass radial displacement averaged over the last 0.2 seconds for different values of λ. Lattice Boltzmann method with our inside/outside tracking and local viscosity change reproduces the λ -phase-transition from boundary-integral simulations quite accurately [Color figure can be viewed at wileyonlinelibrary.com]

5 | CONCLUSION

We presented an efficient tracking algorithm to distinguish the interior fluid of a dynamically deforming RBC from the outside fluid during a lattice-Boltzmann-immersed-boundary simulation. By calculating the scalar products of area-weighted surface vertex normals with local distance vectors between the surface vertices and the LBM lattice points, we track the enclosed cell volume. As our algorithm treats only those LBM lattice points which are in immediate vicinity to the RBC membrane, it is capable of very accurate discrete volume tracking without significantly impacting simulation performance.

As one particular application, we examined a RBC with viscosity contrast λ flowing through a microchannel. The results demonstrated good agreement between the LBM-IBM approach and BIM simulations including viscosity contrast. Finally, our method is not restricted to LBM simulations but can be employed equally well in combination with other grid-based approaches such as finite-difference or finite-volume methods.

ACKNOWLEDGEMENTS

M.L. acknowledges funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - Project number 391977956 - SFB 1357 “Microplastics” (subproject B04). S.J.M. acknowledges funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - Project number 326998133 - TRR 225 “Biofabrication” (subproject B07). S.G. acknowledges funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - Project number 417989940 - FOR 2688 “Instabilities, Bifurcations and Migration in Pulsatile Flows” subproject B3 “Pulsating flows in the microcirculation”. We gratefully acknowledge computing time provided by the SuperMUC system of the Leibniz Rechenzentrum, Garching. We further acknowledge support through the computational resources provided by the Bavarian Polymer Institute.

Conflict of interest

The authors declare no potential conflict of interests.

ORCID

Moritz Lehmann https://orcid.org/0000-0002-4652-8383
Sebastian Johannes Müller https://orcid.org/0000-0002-6020-4991
Stephan Gekle https://orcid.org/0000-0001-5597-1160

REFERENCES

1. Kaoui B, Biros G, Misbah C. Why do red blood cells have asymmetric shapes even in a symmetric flow? *Phys Rev Lett*. 2009;103(18):188101.
2. Shi L, Pan T-W, Glowinski R. Numerical simulation of lateral migration of red blood cells in Poiseuille flows. *Int J Numer Methods Fluids*. 2012;68(11):1393-1408.
3. Vlahovska PM, Barthès-Biesel D, Misbah C. Flow dynamics of red blood cells and their biomimetic counterparts. *C R Physique*. 2013;14(6):451-458.
4. Aouane O, Thiébaud M, Benyoussef A, Wagner C, Misbah C. Vesicle dynamics in a confined Poiseuille flow: from steady state to chaos. *Phys Rev E*. 2014;90(3):033011.
5. Fedosov DA, Peltomäki M, Gompper G. Deformation and dynamics of red blood cells in flow through cylindrical microchannels. *Soft Matter*. 2014;10(24):4258-4267.
6. Geislinger TM, Franke T. Hydrodynamic lift of vesicles and red blood cells in flow — from Fåhræus & Lindqvist to microfluidic cell sorting. *Adv Colloid Interf Sci*. 2014;208:161-176.

7. Lanotte L, Mauer J, Mendez S, et al. Red cells' dynamic morphologies govern blood shear thinning under microcirculatory flow conditions. *Proc Natl Acad Sci U S A*. 2016;113(47):13289-13294.
8. Clavería V, Aouane O, Thiébaud M, et al. Clusters of red blood cells in microcapillary flow: hydrodynamic versus macromolecule induced interaction. *Soft Matter*. 2016;12(39):8235-8245.
9. Cordasco D, Bagchi P. On the shape memory of red blood cells. *Phys Fluids*. 2017;29(4):041901.
10. Secomb TW. Blood flow in the microcirculation. *Ann Rev Fluid Mech*. 2017;49:443-461.
11. Guckenberger A, Kihm A, John T, Wagner C, Gekle S. Numerical-experimental observation of shape bistability of red blood cells flowing in a microchannel. *Soft Matter*. 2018;14(11):2032-2043.
12. Losserand S, Coupier G, Podgorski T. Migration velocity of red blood cells in microchannels. *Microvasc Res*. 2019;124:30-36.
13. Tomaiuolo G, Simeone M, Martinelli V, Rotoli B, Guido S. Red blood cell deformation in microconfined flow. *Soft Matter*. 2009;5(19):3736-3740.
14. Cluitmans JCA, Chokkalingam V, Janssen AM, Brock R, Huck WTS, Bosman GJCGM. Alterations in red blood cell deformability during storage: a microfluidic approach. *Biomed Res Int*. 2014;2014:764268.
15. Prado G, Farutin A, Misbah C, Bureau L. Viscoelastic transient of confined red blood cells. *Biophys J*. 2015;108(9):2126-2136.
16. Kihm A, Kaestner L, Wagner C, Quint S. Classification of red blood cell shapes in flow using outlier tolerant machine learning. *PLoS Comput Biol*. 2018;14(6):e1006278.
17. Quint S, Christ AF, Guckenberger A, et al. 3D tomography of cells in micro-channels. *Appl Phys Lett*. 2017;111(10):103701.
18. Freund JB. Numerical simulation of flowing blood cells. *Annu Rev Fluid Mech*. 2014;46:67-95.
19. Danker G, Vlahovska P, Misbah C. Vesicles in Poiseuille flow. *Phys Rev Lett*. 2009;102(14):148102.
20. Kaoui B, Krüger T, Harting J. How does confinement affect the dynamics of viscous vesicles and red blood cells? *Soft Matter*. 2012;8(35):9246-9252.
21. Tahiri N, Biben T, Ez-Zahraouy H, Benyoussef A, Misbah C. On the problem of slipper shapes of red blood cells in the microvasculature. *Microvasc Res*. 2013;85:40-45.
22. Farutin A, Misbah C. Symmetry breaking and cross-streamline migration of three-dimensional vesicles in an axial Poiseuille flow. *Phys Rev E*. 2014;89(4):042709.
23. Ye H, Shen Z, Li Y. Interplay of deformability and adhesion on localization of elastic micro-particles in blood flow. *J Fluid Mech*. 2018;861:55-87.
24. Mauer J, Mendez S, Lanotte L, et al. Flow-induced transitions of red blood cell shapes under shear. *Phys Rev Lett*. 2018;121(11):118103.
25. Zhu Q, Asaro RJ. Response modes of erythrocytes in high-frequency oscillatory shear flows. *Phys Fluids*. 2019;31(5):051901.
26. Takeishi N, Rosti ME, Imai Y, Wada S, Brandt L. Haemorheology in dilute, semi-dilute and dense suspensions of red blood cells. *J Fluid Mech*. 2019;872:818-848.
27. Pozrikidis C. Numerical simulation of the flow-induced deformation of red blood cells. *Ann Biomed Eng*. 2003;31(10):1194-1205.
28. Peng Z, Asaro RJ, Zhu Q. Multiscale modelling of erythrocytes in Stokes flow. *J Fluid Mech*. 2011;686:299-337.
29. Barakat JM, Shaqfeh ESG. Stokes flow of vesicles in a circular tube. *J Fluid Mech*. 2018;851:606-635.
30. Guckenberger A, Gekle S. A boundary integral method with volume-changing objects for ultrasound-triggered margination of microbubbles. *J Fluid Mech*. 2018;836:952-997.
31. Ye T, Shi H, Peng L, Li Y. Numerical studies of a red blood cell in rectangular microchannels. *J Appl Phys*. 2017;122(8):084701.
32. Balogh P, Bagchi P. A computational approach to modeling cellular-scale blood flow in complex geometry. *J Comput Phys*. 2017;334:280-307.
33. Krüger T, Kusumaatmaja H, Kuzmin A, Shardt O, Silva G. Viggen erlend magnus. *The Lattice Boltzmann Method*. New York, NY: SpringerInternational Publishing; 2017.
34. Peskin CS. The immersed boundary method. *Anuario*. 2003;11:479-517.
35. Seol Y, Hu W-F, Kim Y, Lai M-C. An immersed boundary method for simulating vesicle dynamics in three dimensions. *J Comput Phys*. 2016;322:125-141.
36. Závodszky G, Rooij B, Azizi V, Hoekstra A. Cellular level in-silico modeling of blood rheology with an improved material model for red blood cells. *Front Phys*. 2017;8:563.
37. Tian F-B, Dai H, Luo H, Doyle JF, Rousseau B. Fluid–structure interaction involving large deformations: 3D simulations and applications to biological systems. *J Comput Phys*. 2014;258:451-469.
38. Kaoui B, Harting J. Two-dimensional lattice Boltzmann simulations of vesicles with viscosity contrast. *Rheol Acta*. 2016;55(6):465-475.
39. Frijters S, Krüger T, Harting J. Parallelised Hoshen-Kopelman algorithm for lattice-Boltzmann simulations. *Comput Phys Commun*. 2015;189:92-98.
40. Takeishi N, Imai Y, Nakaaki K, Yamaguchi T, Ishikawa T. Leukocyte margination at arteriole shear rate. *Phys Rep*. 2014;2(6):e12037.
41. Krüger T. Effect of tube diameter and capillary number on platelet margination and near-wall dynamics. *Rheol Acta*. 2016;55(6):511-526.
42. Haan M, Závodszky G, Azizi V, Hoekstra A. Numerical investigation of the effects of red blood cell cytoplasmic viscosity contrasts on single cell and bulk transport behaviour. *Appl Sci*. 2018;8(9):1616.
43. Aidun CK, Clausen JR. Lattice-Boltzmann method for complex flows. *Annu Rev Fluid Mech*. 2010;42:439-472.
44. Limbach HJ, Arnold A, Mann BA, Holm C. ESPResSo—an extensible simulation package for research on soft matter systems. *Comput Phys Commun*. 2006;174(9):704-727.
45. Arnold A, Lenz O, Kesselheim S, et al. *ESPResSo 3.1: Molecular Dynamics Software for Coarse-Grained Models*. Berlin Heidelberg / Germany: Springer; 2013.

46. Weik F, Weeber R, Szuttor K, et al. ESPResSo 4.0 — an extensible software package for simulating soft matter systems. *Eur Phys J Spec Top*. 2019;227(14):1789-1816.
47. Guo Z, Zheng C, Shi B. Discrete lattice effects on the forcing term in the lattice Boltzmann method. *Phys Rev E*. 2002;65(4):046308.
48. Mittal R, Iaccarino G. Immersed boundary methods. *Ann Rev Fluid Mech*. 2005;37:239-261.
49. Krüger T, Varnik F, Raabe D. Efficient and accurate simulations of deformable particles immersed in a fluid using a combined immersed boundary lattice Boltzmann finite element method. *Comput Math Appl*. 2011;61(12):3485-3505.
50. Loop Charles Teorell. Smooth Subdivision Surfaces Based on Triangles (Master Thesis); University of Utah.
51. Skalak R, Tozeren A, Zarda RP, Chien S. Strain energy function of red blood cell membranes. *Biophys J*. 1973;13(3):245-264.
52. Guckenberger A, Schraml MP, Chen PG, Leonetti M, Gekle S. On the bending algorithms for soft objects in flows. *Comput Phys Commun*. 2016;207:1-23.
53. Guckenberger A, Gekle S. Theory and algorithms to compute Helfrich bending forces: a review. *J Phys Condens Matter*. 2017;29(20):203001.
54. Gompper G, Kroll DM. Random surface discretizations and the renormalization of the bending rigidity. *Journal de Physique I*. 1996;6(10):1305-1320.
55. Bächer C, Gekle S. Computational modeling of active deformable membranes embedded in three-dimensional flows. *Phys Rev E*. 2019;99(6):062418.
56. Helfrich W. Elastic properties of lipid bilayers, theory and possible experiments. *Z Naturforsch*. 1973;28(11-12):693-703.
57. Evans E, Fung YC. Improved measurements of the erythrocyte geometry. *Microvasc Res*. 1972;4(4):335-347.
58. Shimrat M. Algorithm 112: position of point relative to polygon. *Commun ACM*. 1962;1:434.
59. Möller T, Trumbore B. Fast, minimum storage ray/triangle intersection. *SIGGRAPH '05*. New York, NY: ACM; 2005;2(1):21-28.

AUTHOR BIOGRAPHY

Moritz Lehmann was born in 1997 in Bavaria, Germany. During school he taught himself multiple programming languages and developed his first software project called *PhysX3D*, a real-time 3D *n*-body simulation with a custom graphics engine for orbit plotting. After the Abitur he studied physics at the University of Bayreuth. In addition to studying biophysics, he is currently doing his PhD in theoretical physics – specializing in high-performance GPU programming with OpenCL – on an efficient GPU implementation of the LBM named *FluidX3D*, which speeds up complex simulations with free surfaces, particles and thermal convection from days to minutes of compute time while at the same time visualizing results in real time.

How to cite this article: Lehmann M, Müller SJ, Gekle S. Efficient viscosity contrast calculation for blood flow simulations using the lattice Boltzmann method. *Int J Numer Meth Fluids*. 2020;1–15. https://doi.org/10.1002/fld.4835

APPENDIX A: C++ IMPLEMENTATION OF THE INITIALIZATION WITH RAY-CASTING

```
void inout_initialize_raycast() { // initialize by ray-casting
	double x, y, z;
	for(int i=0; i<gLocalLatticeSizeX; i++) {
		for(int j=0; j<gLocalLatticeSizeY; j++) {
			for(int k=0; k<gLocalLatticeSizeZ; k++) {
				x = (i+gLocalLatticeOffsetX+gGlobalLatticeOffsetX)*gLatticeConstant;
				y = (j+gLocalLatticeOffsetY+gGlobalLatticeOffsetY)*gLatticeConstant;
				z = (k+gLocalLatticeOffsetZ+gGlobalLatticeOffsetZ)*gLatticeConstant;
				int intersections = 0;
				for(int n=0; n<gCurrentNumberOfTriangles; n++) {
					if(inout_intersectRayTriangle( // geometric algorithm to detect
						inout_Ray(                  // intersection between a line
							inout_Vector(x, y, z),     // segment and a triangle in 3D
							inout_Vector(x+0.01, y+0.03, z+1.04)),
						inout_Triangle(
							inout_Vector(gPoints[gTriangles[n][0]][0], gPoints[gTriangles[n][0]][1], gPoints[gTriangles[n]
[0]][2]),
							inout_Vector(gPoints[gTriangles[n][1]][0], gPoints[gTriangles[n][1]][1], gPoints[gTriangles[n]
[1]][2]),
							inout_Vector(gPoints[gTriangles[n][2]][0], gPoints[gTriangles[n][2]][1], gPoints[gTriangles[n]
[2]][2])
						)
					)) intersections++; // increment number of intersections
				}
				mData[i][j][k] = intersections%2!=0; // true if intersections is odd
			}
		}
	}
}
```

Listing 1: C++ implementation of the initialization with ray casting.

APPENDIX B: C++ IMPLEMENTATION OF THE *UPDATE VIA VERTEX NORMALS* FOR ESPRESSO

```
// flag array that contains the boolean inside/outside information:
//     bool    mData[mLocalLatticeSizeX][mLocalLatticeSizeY][mLocalLatticeSizeZ]
// four additional auxiliary arrays need to be allocated in system memory at simulation startup:
// (A) double mNormalVectorOnPoint[mMaxNumberOfPoints][3][3]
// (B) int    mUpdateList[mLocalLatticeSizeX*mLocalLatticeSizeY*mLocalLatticeSizeZ][3]
// (C) int    mClosestPoint[mLocalLatticeSizeX][mLocalLatticeSizeY][mLocalLatticeSizeZ]
// (D) double mClosestDistance[mLocalLatticeSizeX][mLocalLatticeSizeY][mLocalLatticeSizeZ]
void InoutLatticeLocal::update() { // update cell boundary (closest membrane vertex to lattice point preferred)
        fetch_surface_data(); // fill mPoints and mTriangles with new data
        const int r = 1; // radius of lattice points around membrane vertex
        double x, y, z; // temporary variables
        int i, j, k; // discrete lattice point indices
        int u, v, w; // discrete lattice point indices for the 8 cube corner points
        double px, py, pz; // vector from membrane vertex to lattice point
        double ux, uy, uz; // span vector 1 of triangle
        double vx, vy, vz; // span vector 2 of triangle
        double nx, ny, nz; // normal vector on surface triangele
        // reset mUpdateNumber and mNormalVectorOnPoint from last step
        mUpdateNumber = 0; // reset update number
        for(int n=0; n<mCurrentNumberOfPoints; n++) {
                mNormalVectorOnPoint[n][0] = 0; // reset corner normal vectors
                mNormalVectorOnPoint[n][1] = 0;
                mNormalVectorOnPoint[n][2] = 0;
        }
        // calculate mNormalVectorOnPoint for every membrane vertex as the average over the normal vectors on all adjacent triangles (corner normal
           vectors)
        for(int n=0; n<mCurrentNumberOfTriangles; n++) {
                // span vectors of triangle
                ux = mPoints[mTriangles[n][1]][0]-mPoints[mTriangles[n][0]][0];
                uy = mPoints[mTriangles[n][1]][1]-mPoints[mTriangles[n][0]][1];
                uz = mPoints[mTriangles[n][1]][2]-mPoints[mTriangles[n][0]][2];
                vx = mPoints[mTriangles[n][2]][0]-mPoints[mTriangles[n][0]][0];
                vy = mPoints[mTriangles[n][2]][1]-mPoints[mTriangles[n][0]][1];
                vz = mPoints[mTriangles[n][2]][2]-mPoints[mTriangles[n][0]][2];
                nx = uy*vz-uz*vy; // n = u x v
                ny = uz*vx-ux*vz;
                nz = ux*vy-uy*vx;
                // add normal vector to all three points
                mNormalVectorOnPoint[mTriangles[n][0]][0] += nx;
                mNormalVectorOnPoint[mTriangles[n][0]][1] += ny;
                mNormalVectorOnPoint[mTriangles[n][0]][2] += nz;
                mNormalVectorOnPoint[mTriangles[n][1]][0] += nx;
                mNormalVectorOnPoint[mTriangles[n][1]][1] += ny;
                mNormalVectorOnPoint[mTriangles[n][1]][2] += nz;
                mNormalVectorOnPoint[mTriangles[n][2]][0] += nx;
                mNormalVectorOnPoint[mTriangles[n][2]][1] += ny;
                mNormalVectorOnPoint[mTriangles[n][2]][2] += nz;
        }
        // determine all lattice points that need to be processed, save them in mUpdateList and the closest membrane vertex to them in mClosestPoint
        for(int n=0; n<mCurrentNumberOfPoints; n++) {
                x = mPoints[n][0];
                y = mPoints[n][1];
                z = mPoints[n][2];
                // discrete lattice point indices of the bottom left point
                i = (int)(x/mLatticeConstant-mLocalLatticeOffsetX-mGlobalLatticeOffsetX);
                j = (int)(y/mLatticeConstant-mLocalLatticeOffsetY-mGlobalLatticeOffsetY);
                k = (int)(z/mLatticeConstant-mLocalLatticeOffsetZ-mGlobalLatticeOffsetZ);
                for(int a=1-r; a<=r; a++) {
                        for(int b=1-r; b<=r; b++) {
                                for(int c=1-r; c<=r; c++) {
                                        // discrete lattice point indices for the 8 cube corner points
                                        u = i+a;
                                        v = j+b;
                                        w = k+c;
                                        // check if lattice point is out of bounds
                                        if(u<0 || u>=mLocalLatticeSizeX || v<0 || v>=mLocalLatticeSizeY || w<0 || w>=mLocalLatticeSizeZ) continue;
                                        // vector from membrane vertex to lattice point
                                        px = mLatticeConstant*(u+mLocalLatticeOffsetX+mGlobalLatticeOffsetX) - x;
                                        py = mLatticeConstant*(v+mLocalLatticeOffsetY+mGlobalLatticeOffsetY) - y;
                                        pz = mLatticeConstant*(w+mLocalLatticeOffsetZ+mGlobalLatticeOffsetZ) - z;
                                        // mark closest triangle ID to lattice point
                                        const double sqd = d_sq(px)+d_sq(py)+d_sq(pz);
                                        if(sqd < mClosestDistance[u][v][w]) {
                                                mClosestDistance[u][v][w] = sqd;
                                                mClosestPoint[u][v][w] = n;
                                                mUpdateList[mUpdateNumber][0] = u; // point needs to be updated
                                                mUpdateList[mUpdateNumber][1] = v;
                                                mUpdateList[mUpdateNumber][2] = w;
                                                mUpdateNumber++;
                                        }
                                }
                        }
                }
        }
        // finally, update all lattice points previously saved in mUpdateList
        for(int u=0; u<mUpdateNumber; u++) {
                // get lattice point that needs to be updated
                i = mUpdateList[u][0];
                j = mUpdateList[u][1];
                k = mUpdateList[u][2];
                // get ID of closest membrane vertex
                const int n = mClosestPoint[i][j][k];
                nx = mNormalVectorOnPoint[n][0];
                ny = mNormalVectorOnPoint[n][1];
                nz = mNormalVectorOnPoint[n][2];
                // vector from closest membrane vertex to lattice point
                px = mLatticeConstant*(i+mLocalLatticeOffsetX+mGlobalLatticeOffsetX) - mPoints[n][0];
                py = mLatticeConstant*(j+mLocalLatticeOffsetY+mGlobalLatticeOffsetY) - mPoints[n][1];
```

```
                    pz = mLatticeConstant*(k+mLocalLatticeOffsetZ+mGlobalLatticeOffsetZ) - mPoints[n][2];
                    mData[i][j][k] = px*nx+py*ny+pz*nz < 0.0; // activate if scalar product < 0
                    mClosestDistance[i][j][k] = 1E10; // reset closest distance to lattice point
            }
            if((mStepCounter++)
    }
```

Listing 2: C++ implementation of the algorithm in Section 3.2.