Technische Universität München
Institut für Informatik
Prof. Dr. Hans-Joachim Bungartz
Christoph Kowitz, Martin Schreiber, Benjamin Ueckermann

Summer Term 2013
Worksheet 2

Praktikum Wissenschaftliches Rechnen
Computational Fluid Dynamics

Worksheet 2
The Lattice-Boltzmann-Method

Deadline: May 13th, 10:00 am

1 The Lattice Boltzmann Method

In this worksheet, we are going to simulate a three-dimensional lid-driven cavity scenario by means of the *Lattice Boltzmann Method* (LBM). Evolving in the last two decades, the LBM has turned out to be an alternative to traditional Navier-Stokes solvers. However, instead of considering incompressible flows, Lattice Boltzmann schemes simulate *weakly compressible* flows. These flows are characterized by a *very small Mach number* Ma which is defined as the ratio of the characteristic flow velocity u_c of our problem and the speed of sound c_s in the fluid:

$$Ma := \frac{u_c}{c_s} \tag{1}$$

Hence, it needs to hold $Ma \ll 1$ (example: $c_s \approx 1500\frac{m}{s}$ in water and $c_s \approx 350\frac{m}{s}$ in air). For the idealized case of an incompressible fluid (see worksheet 1), the Mach number completely vanishes with $c_s \to \infty$.
In contrast to the Navier-Stokes equations that are derived from the conservation laws of continuum mechanics, the LBM has its origins in statistical mechanics. It is based on the Boltzmann equation:

$$\frac{\partial f}{\partial t} + \vec{v} \cdot \nabla f = \Delta(f - f^{eq}) \tag{2}$$

where $f(\vec{x}, \vec{v}, t)$ denotes the probability density for finding fluid molecules in an infinitesimal volume around $\vec{x} \in \mathbb{R}^D$ at time t having the velocity $\vec{v} \in \mathbb{R}^D$. The operator ∇ denotes the usual spatial gradient operator, $\nabla f = (\frac{\partial f}{\partial x_1}, ..., \frac{\partial f}{\partial x_D})^\top$.
The operator $\Delta(f - f^{eq})$ on the right hand side of Eq.(2) is called *collision operator* and represents changes due to intermolecular collisions in the fluid. Here, we only consider the case where the state

of our system is very close to its equilibrium state f^{eq}. In this case, we can approximate the collision operator by the *BGK approximation*:

$$\Delta(f - f^{eq}) \approx -\frac{1}{\tau}(f - f^{eq}) \tag{3}$$

where τ is the *relaxation time* of our system, i.e. a characteristic time for the relaxation process of f towards equilibrium.
As we deal with probabilities which are related to molecular quantities, we see that Eq.(2) provides information on our flow system on the microscopic scale rather than the macroscopic scale as the Navier-Stokes equations do. However, we can obtain macroscopic quantities such as the mean velocity $\vec{u}$ or density ρ in a certain point $\vec{x}$ by integrating our probability density f over the velocity space:

$$\begin{aligned} \rho(\vec{x},t) &:= \int\limits_{\mathbb{R}^D} f \;\; dv_1...dv_D \\ \rho(\vec{x},t)\vec{u}(\vec{x},t) &:= \int\limits_{\mathbb{R}^D} f \cdot \vec{v} \;\; dv_1...dv_D \end{aligned} \tag{4}$$

Knowing our macroscopic quantities in each point $(\vec{x},t)$, we can compute our equilibrium distribution $f^{eq}(\vec{x},\vec{v},t)$ which is given by the *Maxwell-Boltzmann distribution*:

$$f^{eq}(\vec{x},\vec{v},t) = \left(\frac{m_p}{2\pi k_B T}\right)^{\frac{D}{2}} e^{-\frac{m_p(\vec{v}-\vec{u}(\vec{x},t))^2}{2k_B T}} \tag{5}$$

where $k_B \approx 1.38 \cdot 10^{-23}\frac{\mathrm{J}}{\mathrm{K}}$ is Boltzmann's Constant, m_p is the mass of a single molecule and T is the temperature of our fluid. We only consider the isothermal case where $T = const$.

2 Discretization

As our probability density $f(\vec{x},\vec{v},t)$ lives in a $(2D+1)$-dimensional space, a direct numerical simulation based on Eq.(2) would be much too expensive. Therefore, the Lattice Boltzmann scheme shrinks the infinite number of velocities $v \in \mathbb{R}^D$ to a finite set $\vec{c}_i$, $i = 0,...,Q-1$. The particles in LBM consequently are only allowed to move with one of the Q velocities $\vec{c}_i$.
Besides, space is discretized in cubic cells with meshsize dx. The velocity set $\{\vec{c}_i\}_{i=0,...,Q-1}$ is now chosen such that fluid molecules can either stay in their current cell or move to the center of direct neighbour cell in a single timestep.
In our implementation, we will restrict ourselves to the *D3Q19* model (three-dimensional model with 19 *lattice velocities* $\vec{c}_i$), see Fig.1, and enumerate the velocities lexicographically with respect to the cells that their vectors point to, see Tab.1.
So, instead of searching for one function $f(\vec{x},\vec{v},t) : \mathbb{R}^{2D} \times [0,t_{end}) \to \mathbb{R}$, we now search for Q functions $f_i(\vec{x},t) : \mathbb{R}^D \times [0,t_{end}) \to \mathbb{R}$ which fulfill

$$\frac{\partial f_i}{\partial t} + \vec{c}_i \cdot \nabla f_i = -\frac{1}{\tau}(f_i - f_i^{eq}), \quad i = 0,...,Q-1. \tag{6}$$

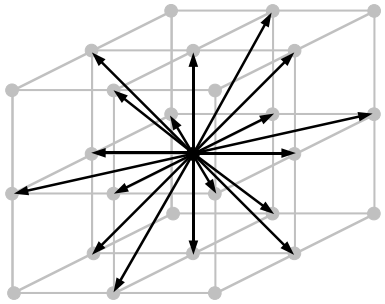

Figure 1: D3Q19 disretization scheme.

$\vec{c}_0$	(0,-1,-1)
$\vec{c}_1$	(-1, 0,-1)
$\vec{c}_2$	(0, 0,-1)
$\vec{c}_3$	(1, 0,-1)
$\vec{c}_4$	(0, 1,-1)

$\vec{c}_5$	(-1,-1, 0)
$\vec{c}_6$	(0,-1, 0)
$\vec{c}_7$	(1,-1, 0)
$\vec{c}_8$	(-1, 0, 0)
$\vec{c}_9$	(0, 0, 0)

$\vec{c}_{10}$	(1, 0, 0)
$\vec{c}_{11}$	(-1, 1, 0)
$\vec{c}_{12}$	(0, 1, 0)
$\vec{c}_{13}$	(1, 1, 0)
$\vec{c}_{14}$	(0,-1, 1)

$\vec{c}_{15}$	(-1, 0, 1)
$\vec{c}_{16}$	(0, 0, 1)
$\vec{c}_{17}$	(1, 0, 1)
$\vec{c}_{18}$	(0, 1, 1)

Table 1: Lattice velocities of the D3Q19 model on a unit lattice ($dx = 1$).

Using the Euler approximation for the timestepping, a time-implicit, forward Euler approximation for the spatial gradient expression $\vec{c}_i \cdot \nabla f_i$ and a Crank-Nicolson-like form for the discrete collision operator representation, we end up with the Lattice Boltzmann update rule:

$$f_i(\vec{x} + \vec{c}_i dt, t + dt) := f_i(\vec{x}, t) - \frac{1}{\tau}\left(f_i(\vec{x}, t) - f_i^{eq}\right), \quad i = 0, ..., Q - 1. \tag{7}$$

From now on, we will consider the dimensionless case, where $dx = 1$, $dt = 1$. The relaxation time τ is restricted to the interval $(0.5, 2.0)$ for stability reasons and is linked to the kinematic fluid viscosity via

$$\nu = c_s^2 \left(\tau - \frac{1}{2}\right) \tag{8}$$

where $c_s := \frac{1}{\sqrt{3}}$ is the speed of sound on our lattice.
The density $\rho(\vec{x}, t)$ and flow velocity $\vec{u}(\vec{x}, t)$ can be computed locally in each grid cell from the discrete analogons of the integral expressions from Eq.(4):

$$\begin{aligned} \rho(\vec{x}, t) &= \sum_{i=0}^{Q-1} f_i \\ \rho(\vec{x}, t)\vec{u}(\vec{x}, t) &= \sum_{i=0}^{Q-1} f_i \vec{c}_i \end{aligned} \tag{9}$$

We still need a discrete representation of the equilibrium distribution from Eq.(5). In the low Mach

number limit, we can approximate f_i^{eq} by:

$$f_i^{eq}(\rho, \vec{u}) = w_i \rho \left(1 + \frac{\vec{c}_i \cdot \vec{u}}{c_s^2} + \frac{(\vec{c}_i \cdot \vec{u})^2}{2c_s^4} - \frac{\vec{u} \cdot \vec{u}}{2c_s^2}\right) \tag{10}$$

where w_i are lattice specific weights; for the D3Q19 model, they are given by:

$$w_i := \begin{cases} \frac{12}{36} & \|\vec{c}_i\| = 0 \\ \frac{2}{36} & \text{if} \quad \|\vec{c}_i\| = 1 \\ \frac{1}{36} & \|\vec{c}_i\| = \sqrt{2} \end{cases} \tag{11}$$

3 Collide and streaming step

Let's go back to Eq.(7). Considering the right hand side, we observe that it only depends on the local quantities $f_i(\vec{x}, t)$, $\rho(\vec{x}, t)$, $u(\vec{x}, t)$ and $f_i^{eq}(\rho(\vec{x}, t), u(\vec{x}, t))$. So, we can split the timestepping of our algorithm into two parts:

$$\begin{aligned} f_i^*(\vec{x}, t) &= f_i(\vec{x}, t) - \tfrac{1}{\tau}\left(f_i(\vec{x}, t) - f_i^{eq}(\rho(\vec{x}, t), u(\vec{x}, t))\right) \quad \textit{Collide step} \\ f_i(\vec{x} + \vec{c}_i dt, t + dt) &= f_i^*(\vec{x}, t) \qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{Streaming step} \end{aligned} \tag{12}$$

An alternative formulation arises when switching the collide and the streaming step in the time loop:

$$\begin{aligned} f_i(\vec{x}, t) &= f_i^*(\vec{x} - \vec{c}_i dt, t - dt) \qquad\qquad\qquad\qquad\quad \textit{Streaming step} \\ f_i^*(\vec{x}, t) &= f_i(\vec{x}, t) - \tfrac{1}{\tau}\left(f_i(\vec{x}, t) - f_i^{eq}(\rho(\vec{x}, t), u(\vec{x}, t))\right) \quad \textit{Collide step} \end{aligned} \tag{13}$$

In our implementation, we will stick to the formulation of Eq.(13).
We will refer to the f_i^* populations as *post-collision distributions* from now on.
Eq.(12) shows several features of LBM:

- The collision operator only involves local adaptions of the distribution functions f_i.
- Unlike the Navier-Stokes approach, we only need to communicate between two neighboured cells once per timestep when we do the streaming. There is no additional equation system to be solved in each timestep!
- As the whole scheme works very locally via direct neighbour-neighbour relations, it is very well suited for parallel computations.

4 Initial and boundary conditions

Similar to the Navier-Stokes approach, we need to provide valid initial and boundary conditions for our system. In case of the cavity scenario, we will initialise the distributions by

$$f_i(\vec{x}, t = 0) := f_i^{eq}(1.0, \vec{0.0}) = w_i. \tag{14}$$

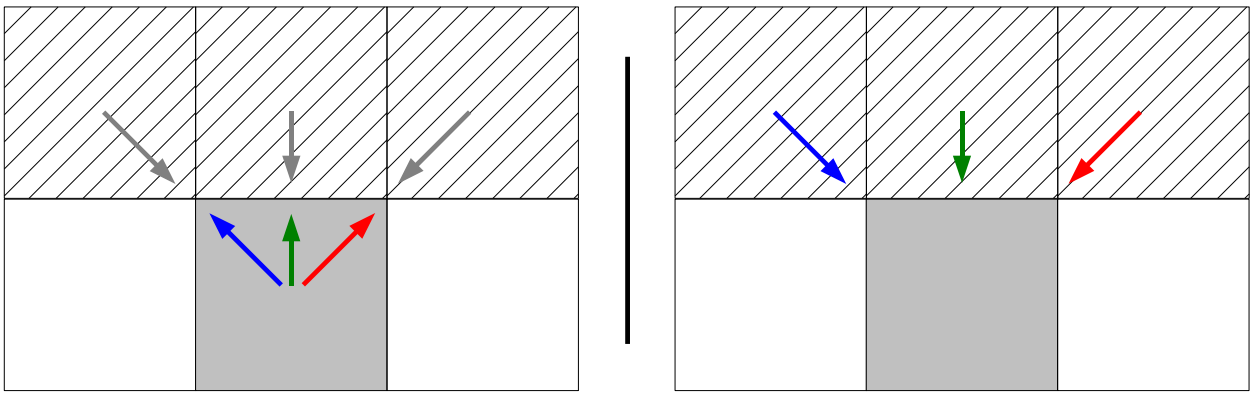

Figure 2: Half-way bounce back scheme. The dashed cells mark cells outside the fluid domain, which are separated from the fluid domain by a no-slip wall. The gray colored distribution functions $f_{inv(i)}$ that would be streamed into the light gray colored cell at position $\vec{x}$ are reconstructed by the red, green and blue post-collision distributions f_i^*.

For the boundary conditions, consider Fig.2. Let $inv(i)$ denote the index of the inverse lattice velocity to $\vec{c}_i$, that is $\vec{c}_{inv(i)} = -\vec{c}_i$. If a distribution function f_i is connected to a lattice velocity $\vec{c}_i$ that points towards a wall (see red, green and blue colored arrows on the left picture of Fig.2), there will be particle populations $f_{inv(i)}$ connected with the lattice velocity $\vec{c}_{inv(i)}$ which would travel into the fluid domain in the streaming step and would originate from a lattice cell outside the computational domain (gray colored distributions on the left picture in Fig.2). Thus, we need to reconstruct these distributions with respect to the type of boundaries that the distributions would cross when entering the domain. In the cavity scenario, we have two types of boundaries: no-slip and moving walls.

No-slip Let us consider the no-slip case first. If we have a no-slip wall (with zero velocity at its surface), it is clear that no particles can cross this wall, i.e. there is no mass passing through the wall. As we would have particles going out of the computational domain from the last fluid cell inside the domain (red, blue and green colored particle populations on the left picture in Fig.2), this means that the same amount of particles needs to enter the computational domain again to guarantee that the mass flux vanishes at the wall. So, we can reconstruct the missing distribution functions by setting:

$$f^*_{inv(i)}(\vec{x} + \vec{c}_i dt, t) := f^*_i(\vec{x}, t) \tag{15}$$

where $\vec{x}$ denotes the fluid cell next to the boundary. This boundary condition is known as the half-way bounce back scheme. It can also be formulated from the boundary cells point of view $\vec{y}$ as:

$$f^*_i(\vec{y}, t) := f^*_{inv(i)}(\vec{y} + \vec{c}_i \mathrm{d}t, t) \tag{16}$$

Moving-wall A similar boundary rule can be derived for moving walls where an additional acceleration term needs to be taken into account. In this case the condition reads for a fluid cell next to

a boundary at position $\vec{x}$:

$$f^*_{inv(i)}(\vec{x} + \vec{c}_i dt, t) := f^*_i(\vec{x}, t) + 2w_i\rho(\vec{x}, t)\frac{\vec{c}_i \cdot \vec{u}_{\text{wall}}}{c_s^2} \tag{17}$$

where u_{wall} is the velocity at the moving wall. The same boundary condition formulated from the point of view of a boundary cell at position $\vec{y}$:

$$f^*_i(\vec{y}, t) := f^*_{inv(i)}(\vec{y} + \vec{c}_i \mathrm{d}t, t) + 2w_i\rho(\vec{y} + \vec{c}_i \mathrm{d}t, t)\frac{\vec{c}_i \vec{u}_{\text{wall}}}{c_s^2} \tag{18}$$

Treating the Boundary in the Code In the implementation of the boundary treatment one has to take care to only take into account the boundary cells. For each boundary cell, the directions i pointing to a neighboring fluid cell have to be found. Then only the distribution functions f^*_i pointing into the same directions have to be updated according to (16) and (18). The other distribution functions in the cell will not be used. In the next streaming step, which is only done for fluid cells anyway, these post-collision distributions are then streamed into the respective fluid cells. For the case of the lid-driven cavity, finding the boundary cells can be done easily, since all boundary cells are at the edge of the domain and they can be easily accessed by a set of `for` loops.

5 Data structures

We consider a cubic domain with $xlength$ fluid cells in each direction. As we need additional information on the boundary nodes, we are going to store the first layer of boundary nodes as well, so that we store the information for $(xlength + 2)^D$ discrete cells.

So, we arrange the following data structures:

- 2 fields with $Q \cdot (xlength+2)^D$ doubles for the pdf field. We store two fields as we want to do a simple copy operation in the streaming step and thus need to avoid overwriting existing data. In order to access the distributions, we arrange them in a lexicographic order, that is the i-th distribution function in the cell (x, y, z) can be found at $Q \cdot (z \cdot xlength^2 + y \cdot xlength + x) + i$.
- 1 field with $(xlength+2)^D$ integers as flag field. This field stores information on the geometry of our problem and is also sorted lexicographically. Currently, we introduce three states for a cell: FLUID=0, NO_SLIP=1 and MOVING_WALL=2. The cells on the very top – that is inside the lid of our cavity – are flagged as MOVING_WALL whereas all other boundary cells are set to NO_SLIP. The inner cells are set to the state FLUID.

6 The algorithm

We now look at the overall algorithm of our program:

```
int main (int argc, char *argv[]){
  double *collideField=NULL;
  double *streamField=NULL;
  int *flagField=NULL;
  int xlength;
  double tau;
  double velocityWall[3];
  int timesteps;
  int timestepsPerPlotting;

  readParameters(
    &xlength,&tau,&velocityWall,timesteps,timestepsPerPlotting,argc, argv
  );
  // TODO: initialise pointers here!

  initialiseFields(collideField,streamField,flagField,xlength);

  for(int t = 0; t < timesteps; t++){
    double *swap=NULL;
    doStreaming(collideField,streamField,flagfield,xlength);
    swap = collideField;
    collideField = streamField;
    streamField = swap;

    doCollision(collideField,flagfield,&tau,xlength);
    treatBoundary(collideField,flagfield,velocityWall,xlength);

    if (t%timestepsPerPlotting==0){
      writeVtkOutput(collideField,flagfield,argv,t,xlength);
    }
  }
}
```

7 Our program

Our simulation code is to be based on the algorithm above. We need to implement the following functions:

- `void readParameters(int *xlength, double *tau, double *velocityWall, int *timesteps, int *timestepsPerPlotting,int argc,`

`char *argv[])`:
Checks that there is exactly one argument handed over to the program and interpretes this parameter as filename. From this file, the domain size $xlength$, the relaxation time τ, the wall velocity $\vec{u}_{wall}$, the number of timesteps and the interval between subsequent VTK outputs are read. For this purpose, we can use the helper functions `read_int(...)` and `read_double(...)` from `init.h`. Implement the function within `initLB.c`.

- `void initialiseFields(double *collideField, double *streamField, int *flagField, int xlength)`:
  Initialises the particle distribution function fields `collideField` and `streamField`. The distributions in both fields are initialised according to the descriptions in Sec.4. The distributions in the boundary layer shall also be set to these values. Implement the function within `initLB.c`.
- `void doStreaming(double *collideField,double *streamField,int *flagField, int xlength)`:
  Carries out the streaming step. For each FLUID cell, the distributions f_i from ALL neighbouring cells $\vec{x}+\vec{c}_i$ are copied from the `collideField` to the i-th position in the `streamingField`. After `doStreaming(...)` is called, the pointers to the `collideField` and the `streamField` need to be switched (see algorithm above) since now, we have our current distribution field $f_i(\vec{x},t)$ stored in the array of `streamField`. Implement the function within `streaming.c`.
- `void doCollision(double *collideField,int *flagField,const double * const tau, int xlength)`:
  Carries out the collision (should be implemented within `collision.c`). Therefore, we loop over all inner cells, that is all cells except for the outer boundary layer, and perform the following steps in each cell (let `currentCell` be a pointer to the first distribution function within the respective cell):

```
double density;
double velocity[3];
double feq[Q];

computeDensity (currentCell, &density);
computeVelocity(currentCell, &density, velocity);
computeFeq     (density,velocity,feq);
computePostCollisionDistributions(currentCell,tau,feq);
```

  The functions used above are described in the following:
  - `void computeDensity(const double * const currentCell, double *density)`:
    Computes the density within `currentCell` according to Eq.(9) and stores the result in `density`. Implement the function within `computeCellValues.c`.
  - `void computeVelocity(const double * const currentCell, const double * const density, double *velocity)`:
    Computes the velocity within `currentCell` according to Eq.(9) and stores the result in `velocity`. Implement the function within `computeCellValues.c`.
  - `void computeFeq(const double * const density, const double * const velocity, double *feq)`:

Computes the equilibrium distribution according to Eq.(10) from the density and the velocity and stores the result in `feq`. Implement the function within `computeCellValues.c`.

  - `void computePostCollisionDistributions(double *currentCell, const double * const tau, const double * const feq)`:
    Computes the post-collision distributions f_i^* according to the BGK update rule from Eq.(13) using the equilibrium distribution values and the relaxation parameter τ. The result is again stored on the `currentCell`. Implement the function within `collision.c`.

- `void treatBoundary(double *collideField, int *flagField, const double * const wallVelocity,int xlength)`:
  Carries out the boundary treatment. Therefore, we loop over the outer boundary cells, check each cell for its state (NO_SLIP or MOVING_WALL), and set the respective distribution functions inside this cell according to Eq.(16) and Eq.(18). Implement the function within `boundary.c`.
- `void writeVtkOutput(const double * const collideField, const int * const flagField,const char *filename,int t,int xlength)`:
  Write density and velocity from the collision field to a file. Implement the function within `visualLB.c`.
  Remark: For this task, we can copy and adapt parts of the `write_vtkFile(...)`-function of worksheet 1.

Besides, we the file `LBDefinitions.h` shall contain static const definitions for the lattice velocities, the lattice weights etc:

```
#ifndef _LBDEFINITIONS_H_
#define _LBDEFINITIONS_H_
  static const int LATTICEVELOCITIES[19][3] = {....};
  static const double LATTICEWEIGHTS[19] = {...};
  static const double C_S = 1.0/sqrt(3.0);
#endif
```

Include the respective definitions within this file.

8 Problems

1. Theory/ Questions to the code:
   a) In Sec.3, some advantages of the LBM are listed. Can you also think of some disadvantages (compared to Navier-Stokes solvers)?
   b) In Tab.1, we introduced an ordering of the lattice velocities within each cell. How can we easily compute the index of the inverse direction $c_{inv(i)}$?
   c) The Chapman-Enskog expansion is an asymptotic theory, relating the Lattice Boltzmann approach to Navier-Stokes. It points out that in the continuum limit, the LBM asymptotically approaches the Navier-Stokes equations. However, consider the linear update rule from Eq.(7). How is it possible that we can approximate the nonlinear Navier-Stokes equations by the Lattice Boltzmann update rule?

d) Storing two times Q variables per cell implies high storage demands. Can you think of special cases where our memory demand might get comparable to the one of Navier-Stokes solvers?

2. Implement all functions from Sec.7 and put together the algorithm from Sec.6 in your `main.c`. Your code needs to compile with the flags `-pedantic`, `-Wall` and `-Werror`.
3. Solve the cavity problem for different Reynolds numbers $Re = \frac{\|u_{wall}\| \cdot xlength}{\nu}$ (Re =0.1, 1.0, 10.0, 100.0, 1000.0). What do you observe (performance, memory demands, number of timesteps to reach steady state)? Visualize the results with ParaView.

9 General/ debugging hints

1. In order to evaluate the performance of your LB code, you can measure the *M*ega *L*attice *U*pdates *P*er *S*econd, that is the number of cells that you handle per timestep divided by 10^6 and the computational time in seconds. A reasonable LB code should be in the range of 6-8 MLUPS. However, we did not start tuning our code yet!
2. Instabilities in LB simulations are encountered when particle distribution functions become negative. This should never happen!
3. The density ρ should always be very close to unity. If deviations occur that are higher than a few percent, something is going wrong!

Please hand in your solution that is your code including makefile *before* the given deadline (see first page)! The answers/ observations to the questions above do not have to be handed in; nevertheless, they can be part of the exam!