Implementing the 2D square lattice Boltzmann method in Matlab

René Fink
Wismar University of Applied Sciences
Research Group Computational Engineering and Automation

September 29, 2006

Abstract

In this paper, a Matlab implementation of the 2D square lattice Boltzmann BGK method is presented. As an application example, the cavity flow problem from Hou et al is chosen. The lattice Boltzmann method is explained shortly followed by a detailed discussion of implementational aspects. The complete program code is attached in the appendix.

1 Introduction

Today, the lattice Boltzmann method (LBM) is very common in simulation domains, especially due to its high potential for parallel processing. Despite its popularity, LBM novices are often overextended since the standard literature ([1, 2]) is rather theoretical and without implementational hints. In the authors opinion, the easiest entrance to LBM can be gained by examination of reference implementations for the famous cavity flow problem by Hou et al ([1]). Such implementations are widely available for Fortran or C but hardly exist for Matlab. Since Matlab provides data parallel operations, examination of such reference implementations should be particularly easy. Existing Matlab LBM implementations are rather complex and do not concern the cavity flow benchmark.

Therefore, a Matlab reference implementation for 2D square LBM and the cavity flow problem is discussed in this paper. The implementation is based on the C reference implementation by Krafczyk ([3]). The complete program code is listed in Appendix A.

2 2D square lattice Boltzmann method

The purpose of the lattice Boltzmann method is to simulate fluid behaviours in complex geometries efficiently in parallel. Traditional fluid simulations, which are based on numerical solutions of the Navier Stokes equations have limited parallel potential and can hardly handle complex geometries. The lattice Boltzmann method is derived from the lattice gas automata (LGA), a cellular automata approach which considers single particles on lattice nodes. In contrast to LGA, LBM deals with distribution function values instead of single particles. The exact denomination for the following described method is *lattice Boltzmann BGK method* (LBGK), caused by the special collision operator being intruduced by Bhatnager, Gross and Krook in 1954.

In 2D square LBM, a square lattice with unit spacing is used. Each node has eight nearest neighbours being connected by eight links (see Fig. 1). Particles on nodes move along the axes with discrete speed $|\vec{e}| = 1$ and along the diagonals with speed $|\vec{e}| = \sqrt{2}$. Furthermore, non moving particles with speed $|\vec{e}| = 0$ are allowed. The occupation of particles is represented by the single-particle distribution function $f_i(\vec{x}, t)$, where i indicates the velocity direction of the particle. The distribution function $f_i(\vec{x}, t)$ represents the probability to find a particle at node $\vec{x}$ and time t with velocity $\vec{e}_i$. The lattice Boltzmann BGK equation is:

$$f_i(\vec{x} + \vec{e}_i, t + 1) = f_i(\vec{x}, t) - \frac{1}{\tau}[f_i(\vec{x}, t) - f_i^{(0)}(\vec{x}, t)] \tag{1}$$

where the left hand side represents the particle propagation term and the right hand side the particle collision term. For particle collision, the function $f_i^{(0)}(\vec{x}, t)$ represents the equilibrium distribution and τ the single relaxation time. The density per node ρ and the macroscopic velocity $\vec{u}$ are defined by:

$$\rho = \sum_i f_i \tag{2}$$

$$\vec{u} = \frac{1}{\rho}\sum_i f_i \vec{e}_i \tag{3}$$

The equation for the equilibrium distribution is:

$$f_i^{(0)} = a\rho[1 + 3(\vec{e}_i\vec{u}) + \frac{9}{2}(\vec{e}_i\vec{u})^2 - \frac{3}{2}|\vec{u}|^2] \tag{4}$$

whith $a = \frac{1}{9}$ for non moving particles, $a = \frac{4}{9}$ for particles moving along the axes and $a = \frac{1}{36}$ for diagonal moving particles. The relaxation time τ can be obtained from the viscosity ν by:

$$\tau = \frac{6\nu + 1}{2} \tag{5}$$

In each time step, two operations have to be performed: collision and propagation. For the collision step, the right hand side of Equation 1 is evaluated. Hereby, the type of cells (or nodes) must be considered which can be either wall cells, fluid cells or driving (evocation) cells. On *fluid cells*, the distribution function values are transformed into density and macroscopic velocity following Equation 2 and 3. Subsequently, the equilibrium distribution function value is evaluated by Equation 4. Finally, the distribution function value is updated by the right hand side of Equation 1. On *driving cells*, distribution function values are only transformed into density. Macroscopic velocities on driving cells are set to pre-defined constant values. Again, the equilibrium distribution function value is evaluated by Equation 4 but in this case, the updated

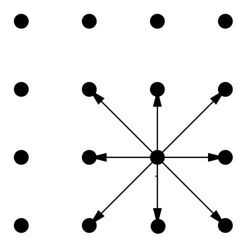
Figure 1: Nearest neighbour links of a lattice node

distribution function value corresponds to the equilibrium value. On *wall cells*, a simple bounce-back rule is applied in which every particle is reflected to its opposite direction.

For the propagation step, distribution function values move to their linked cells by keeping their velocities. The simulation loop is terminated when a steady state is reached, which can be indiced by certain criteria.

3 Matlab Implementation

For the Matlab implementation of the LBM, the lid-driven cavity flow problem defined by Hou et al ([1]) was chosen. In this benchmark case, the flow of an incompressible liquid enclosed in a square cavity is simulated. The flow is driven by a constant stream on the top boundary.

Since a certain example was implemented, there are two program parts: model part and simulator part. The model part contains geometry settings of the cavity, velocity settings of driving cells and definition of initial conditions. The simulator part contains definitions of simulator-specific variables and the LBM code itself being enclosed in the simulation loop. In further sections, there is no distinction between model and simulator part since the programmer's view is in focus there.

The implementation is not focussed on performance but on understandability. Therefore, some operations are performed which are not necessary but increase both readability and compactness of the code.

3.1 Data structures and initialization

In line 13–27, experiment parameters are set. Hereby, the number of nodes and iterations can be specified in `nx`, `ny` and `iterations`. The geometry of the cavity is given by a matrix `geometry` with size `[ny,nx]`. The type of cells is specified in `geometry` by either `0` (fluid cells), `1` (wall cells) or `2` (driving cells). The initial density and the driving velocity on the top boundary are given by the scalars `rho_0` and `u_0`. Finally, the Reynolds number is specified by `Re` affecting viscosity and relaxation which are stored in `viscosity` and `tau`. Taken from [3], the formula for viscosity in dependency on lattice size n ($n = nx = ny$), driving velocity u_0

and Reynolds number Re is:

$$\nu = \frac{(n-1)u_0}{Re} \tag{6}$$

In line 30, helper variables for distribution function value access are defined. These definitions imply the following discrete microscopic velocities $\vec{e}_i$:

$$\left(\begin{array}{ccccccccc} \vec{e}_1 & \vec{e}_2 & \vec{e}_3 & \vec{e}_4 & \vec{e}_5 & \vec{e}_6 & \vec{e}_7 & \vec{e}_8 & \vec{e}_9 \end{array} \right) = \left(\begin{array}{ccccccccc} 0 & 1 & 0 & -1 & 0 & 1 & 1 & -1 & -1 \\ 0 & 0 & -1 & 0 & 1 & 1 & -1 & -1 & 1 \end{array} \right) \tag{7}$$

In line 36–41, variables used in the simulation loop are allocated. Distribution function values are arranged in a matrix `f` with size `[nx*ny,9]` where each row represents one cell while each column represents one discrete velocity or link to neighboured node. In the same way, the matrix `feq` stores equilibrium distribution function values. Macroscopic density and velocities are stored in vectors `rho`, `ux` and `uy`, each with size `[nx*ny,1]`, so each row represents one cell. Finally, a variable `usqr` is defined, storing helper values for later equilibrium evaluations.

In line 44–46, distribution function values are set to initial values following Equation 4, assuming zero macroscopic velocities on $t = 0$. The last step before the simulation loop (line 49–51) is to store the indices of wall cells, fluid cells and driving cells in vectors `WALL`, `FL` and `DR` for data parallel evaluations during the simulation loop.

3.2 Simulation Loop

Line 53–109 contain the simulation loop. The loop runs for a fixed number of iterations, so no steady state criteria are checked. As described in section 2, the simulation loop is divided into two steps: collision and propagation, which are discussed in the following subsections.

3.2.1 Collision step

Collision step evaluations start in line 58–60, where the density and macroscopic velocities for all cells are calculated following Equation 2 and 3. In line 63 and 64, constant velocities for driving cells are set to `u_0` for horizontal direction and `0` for vertical direction. In line 65, values for the helper variable `usqr` are evaluated, representing the term $|\vec{u}|^2$ in Equation 4.

In line 68–76, equilibrium distribution function values of all cells are evaluated following Equation 4. Since discrete velocities $\vec{e}_i$ are implicit, expressions for individual velocities (or directions) must be given explicit instead of using Eq. 4 once.

In line 80–88, distribution function values are updated according to cell types. Wall cells are updated in line 80 by application of the bounce back rule. The usage of helper variables for directions clearly shows the reflection of distribution function values in opposite directions. Driving cells are updated in line 84 by just replacing the old distribution function values by equilibrium values. Fluid cells are updated in line 88 following the right hand side of Equation 1.

3.2.2 Propagation Step

For the propagation step, the matrix of distribution function values (`f`) is transformed into an array of size `[ny,nx,9]` by the Matlab function `reshape()` (line 94). The transformation allows easy and understandable programming of propagation in certain directions. In line 97–104, propagation in each direction is performed by data parallel expressions. For example, distribution function values for particles which move in east direction (line 97) are right shifted for one column. In line 107, distribution function values are rearranged into a matrix of size `[nx*ny,9]` for the next iteration step.

3.3 Visualization

Visualization operations are placed in line 112–118. In line 112, the magnitute of latest macroscopic velocity values is evaluated and scaled by driving velocity `u_0`. The result of the operation is stored in a vector `u` of size `[nx*ny,1]`. In line 113, the vector `u` is transformed into a matrix of size `[ny,nx]` by usage of `reshape()`. The purpose of this transformation is the necessity of 2D data representation used as input for subsequent visualization operations.

In line 114, macroscopic velocity values are displayed in a picture using the Matlab function `imagesc()`. This function constructs a bitmap picture from a double matrix, where the matrix values are scaled to use the full colormap. In line 115, the aspect ratio of the plot is corrected, while in line 116, a colorbar for reading magnitude values is added to the plot. In line 117, a plot title is added containing the number of iterations performed. A visualization plot for a lattice with 257x257 cells after 350000 iterations and Reynolds number 1000 is shown in Figure 2.

4 Conclusion

In this paper, a Matlab reference implementation for 2D square lattice Boltzmann BGK method was presented. As an application example, the famous lid-driven cavity flow problem by Hou et al was used. Using this implementation, Matlab users can gain an easy entrance to the lattice Boltzmann method beside the standard literature.

Containing only 59 lines of code, this implemementation is very compact. Reasons for this compactness are partly data parallel operations but also the acceptance of unnecessary evaluations. For example, evaluation of equilibrium distribution function values in wall cells is not necessary and keeps some potential for speed optimizations.

References

[1] S. Hou, Q. Zou, S. Chen, G. D. Doolen, A. C. Cogley: Simulation of Cavity Flow by the Lattice Boltzmann Method. Journal of Computational Physics, Vol. 118, Issue 2, p. 329-347, 05/1995.

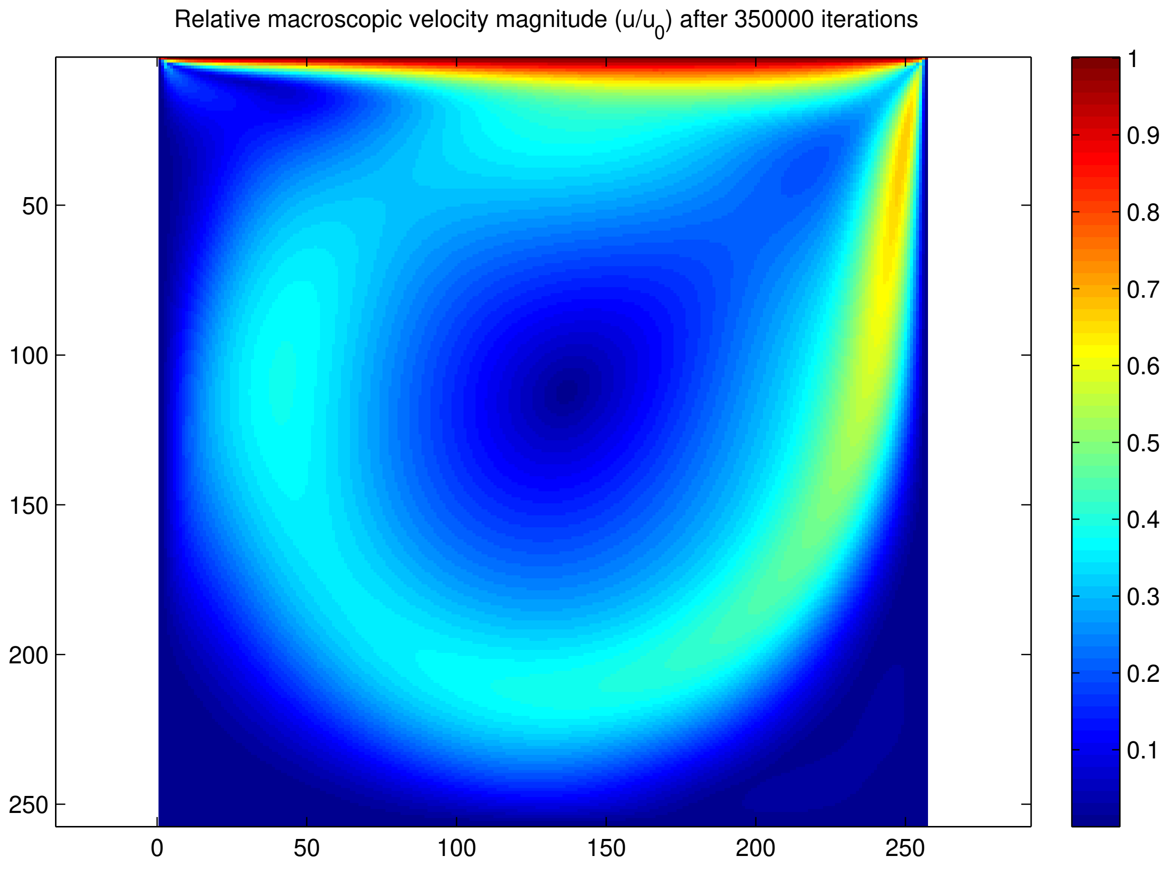

Figure 2: Simulation results visualization

[2] S. Chen, G. D. Doolen: Lattice Boltzmann method for fluid flows. Annual Review of Fluid Mechanics 30, p. 329-364, 1998.

[3] M. Krafczyk: Lattice-BGK TUTORIAL. Version 1.0, 03/2001. `http://www.lstm.uni-erlangen.de/lbm2001/download/LBGK_tutorial.tgz`

A Program code

```
% 2D square Lattice Boltzmann Method reference implementation
%
% Problem description can be found in:
% Hou et al.: Simulation of Cavity Flow by the Lattice Boltzmann Method.
% Journal of Computational Physics, Vol. 118, Issue 2, 05/1995.
%
% Author: Rene Fink, Wismar University of Applied Sciences, Research Group
% Computational Engineering and Automation
%
% Date: 2006-09-26

% begin experiment parameter settings
```

```
nx=33; % number of grid points in x direction
ny=nx; % number of grid points in y direction

iterations=2000; % number of iterations

geometry=ones(ny,nx); % wall cells (geometry==1)
geometry(2:ny-1,2:nx-1)=0; % fluid cells (geometry==0)
geometry(1,:)=2; % driving cells on the top boundary (geometry==2)

rho_0=1; % initial density
u_0=0.1; % driving velocity on the top boundary

Re=1000; % Reynolds number
viscosity=(ny-1)*u_0/Re % kinematic viscosity (0.005 <= viscosity <= 0.2)
tau=(6*viscosity+1)/2; % relaxation time
% end experiment parameter settings

C=1;E=2;S=3;W=4;N=5;NE=6;SE=7;SW=8;NW=9; % helper variables for directions
% directions are indiced as follows:
% 9 5 6
% 4 1 2
% 8 3 7

f=zeros(nx*ny,9); % distribution function values of each cell
feq=zeros(nx*ny,9); % equilibrium disribution function value
rho=zeros(nx*ny,1); % macroscopic density
ux=zeros(nx*ny,1); % macroscopic velocity in x direction
uy=zeros(nx*ny,1); % macroscopic velocity in y direction
usqr=zeros(nx*ny,1); % helper variable

% begin set initial distribution function values
f(:,C)=rho_0*4/9;
f(:,[E S W N])=rho_0/9;
f(:,[NE SE SW NW])=rho_0/36;
% end set initial distribution function values

FL=find(geometry==0); % create indices of all fluid cells
WALL=find(geometry==1); % create indices of all wall cells
DR=find(geometry==2); % create indices of all driving cells

for i=1:iterations

    % begin collision step ----------------------------------------------------

    % begin distribution function value transformation to macroscopic values
    rho(:)=sum(f,2); % macroscopic density
    ux(:)=(f(:,E)-f(:,W)+f(:,NE)+f(:,SE)-f(:,SW)-f(:,NW))./rho; % x velocity
    uy(:)=(f(:,N)-f(:,S)+f(:,NE)+f(:,NW)-f(:,SE)-f(:,SW))./rho; % y velocity
    % end distribution function value transformation to macroscopic values

    ux(DR)=u_0; % set x velocity for driving cells
    uy(DR)=0; % set y velocity for driving cells
    usqr(:)=ux.*ux+uy.*uy; % calculate helper variable value

    % begin equilibrium distribution function value calculation
    feq(:,C)=(4/9)*rho.*(1-1.5*usqr);
    feq(:,E)=(1/9)*rho.*(1+3*ux+4.5*ux.^2-1.5*usqr);
    feq(:,S)=(1/9)*rho.*(1-3*uy+4.5*uy.^2-1.5*usqr);
    feq(:,W)=(1/9)*rho.*(1-3*ux+4.5*ux.^2-1.5*usqr);
    feq(:,N)=(1/9)*rho.*(1+3*uy+4.5*uy.^2-1.5*usqr);
    feq(:,NE)=(1/36)*rho.*(1+3*(ux+uy)+4.5*(ux+uy).^2-1.5*usqr);
    feq(:,SE)=(1/36)*rho.*(1+3*(ux-uy)+4.5*(ux-uy).^2-1.5*usqr);
    feq(:,SW)=(1/36)*rho.*(1+3*(-ux-uy)+4.5*(-ux-uy).^2-1.5*usqr);
    feq(:,NW)=(1/36)*rho.*(1+3*(-ux+uy)+4.5*(-ux+uy).^2-1.5*usqr);
    % end equilibrium distribution function value calculation

    % begin wall cell f calculation (bounce back)
    f(WALL,[C E S W N NE SE SW NW])=f(WALL,[C W N E S SW NW NE SE]);
    % end wall cell f calculation (bounce back)

    % begin driving cell f calculation
    f(DR,:)=feq(DR,:); % distribution function value = equilibrium value
    % end driving cell f calculation

    % begin fluid cell f calculation
    f(FL,:)=f(FL,:)*(1-1/tau)+feq(FL,:)/tau;
    % end fluid cell f calculation

    % end collision step --------------------------------------------------

    % begin propagation step ------------------------------------------------
    f=reshape(f,[ny,nx,9]); % transform f for easy propagation

    % begin particle propagation
    f(:,2:nx,E)=f(:,1:nx-1,E);
    f(2:ny,:,S)=f(1:ny-1,:,S);
    f(:,1:nx-1,W)=f(:,2:nx,W);
    f(1:ny-1,:,N)=f(2:ny,:,N);
    f(1:ny-1,2:nx,NE)=f(2:ny,1:nx-1,NE);
    f(2:ny,2:nx,SE)=f(1:ny-1,1:nx-1,SE);
    f(2:ny,1:nx-1,SW)=f(1:ny-1,2:nx,SW);
    f(1:ny-1,1:nx-1,NW)=f(2:ny,2:nx,NW);
    % end particle propagation

    f=reshape(f,[nx*ny,9]); % re-transform f for next iteration step
    % end propagation step --------------------------------------------------
end

% begin display
u=sqrt(ux.^2+uy.^2)/u_0; % calculate relative macroscopic velocity magnitude
u=reshape(u,ny,nx); % reshape u to 2D for plotting
imagesc(u); % plot macroscopic velicity magnitude
axis('equal'); % make display square
colorbar; % show color index
title(['Relative macroscopic velocity magnitude (u/u_0) after ',...
    num2str(iterations),' iterations']); % show plot title
% end display
```