Briefly on Lattice Boltzmann Method (LBM)

Interest in the **Lattice Boltzmann Method (LBM)** has been steadily increasing since it grew out of lattice gas models in the late 1980s. While both of these methods simulate the flow of liquids and gases by imitating the basic behaviour of a gas—molecules move forwards and are scattered as they collide with each other—the lattice Boltzmann method shed the major disadvantages of its predecessor while retaining its strengths. Furthermore, it gained a stronger theoretical grounding in the physical theory of gases. These days, researchers throughout the world are attracted to the lattice Boltzmann method for reasons such as its simplicity, its scalability on parallel computers, its extensibility, and the ease with which it can handle complex geometries.

1 Introduction

The basic quantity of the LBM is the discrete-velocity distribution function $f_i(\boldsymbol{x}, t)$, often called the particle populations. Similar to the distribution functions, it represents the density of particles with velocity $\boldsymbol{c}_i = (c_{ix}, c_{iy}, c_{iz})$ at position $\boldsymbol{x}$ and time t. Likewise, the mass density ρ and momentum density $\rho\boldsymbol{u}$ at $(\boldsymbol{x}, t)$ can be found through weighted sums known as moments of f_i :

$$\rho(\boldsymbol{x}, t) = \sum_i f_i(\boldsymbol{x}, t), \quad \rho\boldsymbol{u}(\boldsymbol{x}, t) = \sum_i \boldsymbol{c}_i f_i(\boldsymbol{x}, t).$$

The major difference between f_i and the continuous distribution function f is that all of the argument variables of f_i are discrete. $\boldsymbol{c}_i$, to which the subscript i in f_i refers, is one of a small discrete set of velocities $\{\boldsymbol{c}_i\}$. The points $\boldsymbol{x}$ at which f_i is defined are positioned as a square lattice in space, with lattice spacing Δx. Additionally, f_i is defined only at certain times t, separated by a time step Δt.

By discretising the Boltzmann equation in velocity space, physical space, and time, we find the lattice Boltzmann equation

$$f_i\left(\boldsymbol{x} + \boldsymbol{c}_i\Delta t, t + \Delta t\right) = f_i(\boldsymbol{x}, t) + \Omega_i(\boldsymbol{x}, t).$$

This expresses that particles $f_i(\boldsymbol{x}, t)$ move with velocity $\boldsymbol{c}_i$ to a neighboring point $\boldsymbol{x}+\boldsymbol{c}_i\Delta t$ at the next time step $t + \Delta t$ as shown in Fig 1. At the same time, particles are affected by a collision operator Ω_i. This operator models particle collisions by redistributing particles among the populations f_i at each site. While there are many different collision operators Ω_i

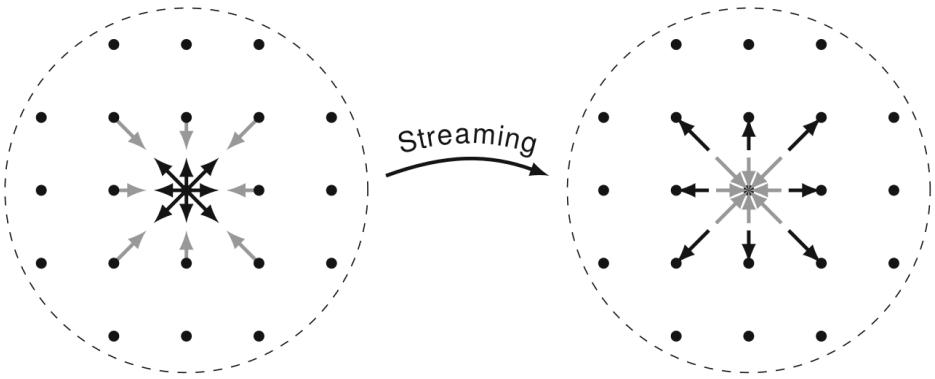

Figure 1: Particles (black) streaming from the central node to its neighbours, from which particles (grey) are streamed back. To the left we see post-collision distributions $f_i^\star$ before streaming, and to the right we see pre-collision distributions f_i after streaming

available, the simplest one that can be used for Navier-Stokes simulations is the Bhatnagar-Gross-Krook (BGK) operator:

$$\Omega_i(f) = -\frac{f_i - f_i^{\mathrm{eq}}}{\tau}\Delta t.$$

It relaxes the populations towards an equilibrium f_i^{eq} at a rate determined by the relaxation time τ

This equilibrium is given by

$$f_i^{\mathrm{eq}}(\boldsymbol{x},t) = w_i\rho\left(1 + \frac{\boldsymbol{u}\cdot\boldsymbol{c}_i}{c_{\mathrm{s}}^2} + \frac{(\boldsymbol{u}\cdot\boldsymbol{c}_i)^2}{2c_{\mathrm{s}}^4} - \frac{\boldsymbol{u}\cdot\boldsymbol{u}}{2c_{\mathrm{s}}^2}\right)$$

with the weights w_i specific to the chosen velocity set. The equilibrium is such that its moments are the same as those of f_i, i.e. $\sum_i f_i^{\mathrm{eq}} = \sum_i f_i = \rho$ and $\sum_i \boldsymbol{c}_i f_i^{\mathrm{eq}} = \sum_i \boldsymbol{c}_i f_i = \rho\boldsymbol{u}$. The equilibrium f_i^{eq} depends on the local quantities density ρ and fluid velocity $\boldsymbol{u}$ only. These are calculated from the local values of f_i, with the fluid velocity found as $\boldsymbol{u}(\boldsymbol{x},t) = \rho\boldsymbol{u}(\boldsymbol{x},t)/\rho(\boldsymbol{x},t)$.

The link between the LBE and the NSE can be determined using the Chapman-Enskog analysis. Through this, we can show that the LBE results in macroscopic behavior according to the NSE, with the kinematic shear viscosity given by the relaxation time τ as

$$v = c_{\mathrm{s}}^2\left(\tau - \frac{\Delta t}{2}\right),$$

and the kinematic bulk viscosity given as $\nu_{\mathrm{B}} = 2\nu/3$. Additionally, the viscous stress tensor can be calculated from f_i as

$$\sigma_{\alpha\beta} \approx -\left(1 - \frac{\Delta t}{2\tau}\right)\sum_i c_{i\alpha}c_{i\beta}f_i^{\mathrm{neq}},$$

where the non-equilibrium distribution $f_i^{\mathrm{neq}} = f_i - f_i^{\mathrm{eq}}$ is the deviation of f_i from equilibrium. However, computing the stress tensor explicitly in this way is usually not a necessary step when performing simulations.

1.1 The Time Step: Collision and Streaming

In full, the lattice BGK (LBGK) equation (i.e. the fully discretised Boltzmann equation with the BGK collision operator) reads

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

We can decompose this equation into two distinct parts that are performed in succession:

1. The first part is collision (or relaxation). In the collision step or relaxation step, each population $f_i(\boldsymbol{x},t)$ receives a collisional contribution and becomes

$$f_i^{\star}(\boldsymbol{x},t) = f_i(\boldsymbol{x},t) - \frac{\Delta t}{\tau}\left[f_i(\boldsymbol{x},t) - f_i^{\mathrm{eq}}(\boldsymbol{x},t)\right]. \tag{1}$$

   Collision is a purely local and algebraic operation. $f_i^{\star}$ denotes the state of the population after collision.

2. The other step is the streaming or propagation step. Here, the post-collision populations $f_i^{\star}(\boldsymbol{x},t)$ just stream along their associated direction $\boldsymbol{c}_i$ to reach a neighboring lattice site where they become $f_i\left(\boldsymbol{x}+\boldsymbol{c}_i\Delta t, t+\Delta t\right)$ as shown in Fig. 1 :

$$f_i\left(\boldsymbol{x}+\boldsymbol{c}_i\Delta t, t+\Delta t\right) = f_i^{\star}(\boldsymbol{x},t).$$

   or

$$f_i(\boldsymbol{x},t) = f_i^{\star}\left(\boldsymbol{x}-\boldsymbol{c}_i\Delta t, t+\Delta t\right). \tag{2}$$

   This is a non-local operation. Practically, one has to copy the memory content of $f_i^{\star}(\boldsymbol{x},t)$ to the lattice site located at $\boldsymbol{x}+\boldsymbol{c}_i\Delta t$ and overwrite its old information. (One has to be careful not to overwrite populations which are still required.) One common strategy is to use two sets of populations, one for reading data, the other for writing data.

Overall, the LBE concept is straightforward. It consists of two parts: collision and streaming. The collision is simply an algebraic local operation. First, one calculates the density ρ and the macroscopic velocity $\boldsymbol{u}$ to find the equilibrium distributions f_i^{eq} and the post-collision distribution $f_i^{\star}$. After collision, we stream the resulting distribution $f_i^{\star}$ to neighboring nodes. When these two operations are complete, one time step has elapsed, and the operations are repeated.

1.2 Velocity Sets and The D2Q9 Lattice

We have seen how velocity space can be discretized. This naturally leads to the question which discrete velocity set $\{\boldsymbol{c}_i\}$ to choose. On the one hand, an appropriate set has to be sufficiently well-resolved to allow for consistent solutions of the Navier-Stokes or even Navier-Stokes-Fourier equations. On the other hand, the numerical cost of the algorithm

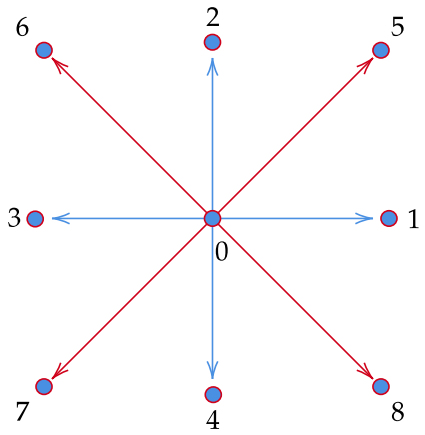

Figure 2: D2Q9 lattice

scales with the number of velocities. It is therefore an important task to find a set with a minimum number of velocities, yet the ability to capture the desired physics.

The most commonly used velocity sets to solve the Navier-Stokes equation are D1Q3, D2Q9, D3Q15, D3Q19 and D3Q27. In this work, D2Q9 velocity set will be used. The velocity components $\{c_{i\alpha}\}$ and weights $\{w_i\}$ are also collected below and the lattice is shown in the Fig. 2.

$$\mathbf{c}_i = \begin{cases} (0,0), & i=0, \\ (\pm 1,0),(0,\pm 1), & i=1,\ldots,4,, \\ (\pm 1,\pm 1), & i=5,\ldots,8. \end{cases} \quad w_i = \begin{cases} \frac{4}{9}, & i=0, \\ \frac{1}{9}, & i=1,\ldots,4, \\ \frac{1}{36}, & i=5,\ldots,8. \end{cases}$$

As given before, the local density ρ and velocity $\boldsymbol{u}$ are (weighted) sums of the populations f_i. It is advisable to unroll these summations, which means writing out the full expressions rather than using loops. Most velocity components are zero, and we do not want to waste CPU time by summing zeros. For example,for the D2Q9 velocity set as defined above, we would implement:

$$\begin{aligned} \rho &= f_0 + f_1 + f_2 + f_3 + f_4 + f_5 + f_6 + f_7 + f_8 \\ u_x &= \left[(f_1 + f_5 + f_8) - (f_3 + f_6 + f_7)\right]/\rho \\ u_y &= \left[(f_2 + f_5 + f_6) - (f_4 + f_7 + f_8)\right]/\rho \end{aligned} \tag{3}$$

The D2Q9 equilibrium distribution for the chosen velocity vectors is given by (using $\boldsymbol{u} = (u_x, u_y)^\top$, $\boldsymbol{u}^2 = u_x^2 + u_y^2$ and $c_s^2 = 1/3$):

$$\begin{aligned} f_0^{\mathrm{eq}} &= \tfrac{2\rho}{9}\left(2 - 3\boldsymbol{u}^2\right), \\ f_1^{\mathrm{eq}} &= \tfrac{\rho}{18}\left(2 + 6u_x + 9u_x^2 - 3\boldsymbol{u}^2\right), & f_5^{\mathrm{eq}} &= \tfrac{\rho}{36}\left[1 + 3\left(u_x + u_y\right) + 9u_xu_y + 3\boldsymbol{u}^2\right], \\ f_2^{\mathrm{eq}} &= \tfrac{\rho}{18}\left(2 + 6u_y + 9u_y^2 - 3\boldsymbol{u}^2\right), & f_6^{\mathrm{eq}} &= \tfrac{\rho}{36}\left[1 - 3\left(u_x - u_y\right) - 9u_xu_y + 3\boldsymbol{u}^2\right], \\ f_3^{\mathrm{eq}} &= \tfrac{\rho}{18}\left(2 - 6u_x + 9u_x^2 - 3\boldsymbol{u}^2\right), & f_7^{\mathrm{eq}} &= \tfrac{\rho}{36}\left[1 - 3\left(u_x + u_y\right) + 9u_xu_y + 3\boldsymbol{u}^2\right], \\ f_4^{\mathrm{eq}} &= \tfrac{\rho}{18}\left(2 - 6u_y + 9u_y^2 - 3\boldsymbol{u}^2\right), & f_8^{\mathrm{eq}} &= \tfrac{\rho}{36}\left[1 + 3\left(u_x - u_y\right) - 9u_xu_y + 3\boldsymbol{u}^2\right]. \end{aligned} \tag{4}$$

1.3 Initialization

The simplest approach to initializing the populations at the start of a simulation is to set them to $f_i^{\mathrm{eq}}(\boldsymbol{x}, t=0) = f_i^{\mathrm{eq}}(\rho(\boldsymbol{x}, t=0), \boldsymbol{u}(\boldsymbol{x}, t=0))$. Often, the values $\rho(\boldsymbol{x}, t=0) = 1$ and $\boldsymbol{u}(\boldsymbol{x}, t=0) = \boldsymbol{0}$ are used.

1.4 Boundary Conditions

In the LBE, the boundary conditions apply at boundary nodes $\boldsymbol{x}_{\mathrm{b}}$ which are sites with at least one link to a solid and a fluid node. The formulation of LB boundary conditions is typically a non-trivial task. Rather than specifying the macroscopic variables of interest, such as ρ and $\boldsymbol{u}$, LB boundary conditions apply to the mesoscopic populations f_i, giving more degrees of freedom than the set of macroscopic variables. Althougn many different boundary conditions can be trivially implemented LBM, in this project we need anly the followings.

- **Bounce-back (no-slip wall):** During propagation, if a particle meets a rigid boundary, it will be reflected back to its original location with its velocity reversed. This mechanism is implemented quite simply as

  $$f_i\left(\boldsymbol{x}_{\mathrm{b}}, t+\Delta t\right) = f_i^{\star}\left(\boldsymbol{x}_{\mathrm{b}}, t\right),$$

  where $f_{\bar{i}}$ refers to the f for the direction $\bar{i}$ such that $c_{\bar{i}} = -c_i$. The BB method leads to a macroscopic no-slip velocity condition for a resting wall located midway on the link between lattice nodes. This can be simply achieved by setting a map between the populations:

  $$f_{[0,1,2,3,4,5,6,7,8]}\left(\boldsymbol{x}_{\mathrm{b}}, t+\Delta t\right) = f_{[0,3,4,1,2,7,8,5,6]}\left(\boldsymbol{x}_{\mathrm{b}}, t\right);$$

- **Velocity outlet:** The outflow condition can be applied with adjusting unavailable incoming distributions to node distribution.

2 Implementation of the Lattice Boltzmann Method

The core LBM algorithm consists of a cyclic sequence of substeps, with each cycle corresponding to one time step.

1. Compute the macroscopic moments $\rho(\boldsymbol{x}, t)$ and $\boldsymbol{u}(\boldsymbol{x}, t)$ using the equation 3 from $f_i(\boldsymbol{x}, t)$.
2. Obtain the equilibrium distribution $f_i^{\mathrm{eq}}\left(\boldsymbol{x}, t\right)$ using the equation 4.
3. If desired, write the macroscopic fields $\rho(\boldsymbol{x}, t), \boldsymbol{u}(\boldsymbol{x}, t)$ and/or $\boldsymbol{\sigma}(\boldsymbol{x}, t)$ to the hard disk for visualization or post-processing.
4. Perform collision (relaxation) using the equation 1.
5. Perform streaming (propagation) using the equation 2.
6. Increase the time step, setting t to $t + \Delta t$, and go back to step 1 until the last time step or convergence has been reached.

2.1 Grid

The computational domain is discretized using an equally spaced, uniform lattice consisting of $N_{\text{nodes}} = (N+2) \times (M+2)$ nodes. The lattice indices are defined as $n = 0, 1, \ldots, N+1$ in the horizontal direction and $m = 0, 1, \ldots, M+1$ in the vertical direction. Each node in the domain is therefore identified by the integer pair (n, m).

By construction, the rows $m = 0$ and $m = M+1$ correspond to wall boundary nodes, while the column $n = N+1$ represents outflow boundary nodes.

Instead of working with the two–index notation (n, m), the nodes may also be addressed using a single serial index defined as

$$\text{index}(n, m) = n + m\,(N+2),$$

which maps the two–dimensional lattice coordinates onto a one–dimensional memory layout.

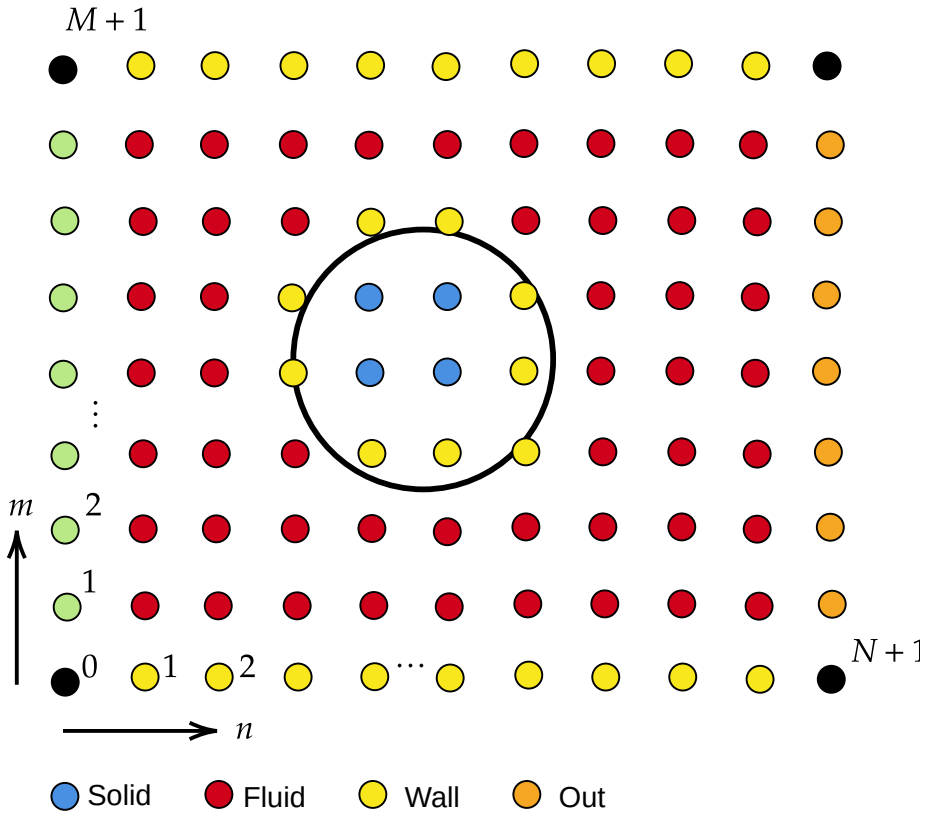

Figure 3: Computational grid with boundaries and node types.

2.2 Data Structure

In the D2Q9 velocity discretization, each lattice node stores $N_f = 9$ distribution values. Consequently, the total number of degrees of freedom for the phase–space distribution function f is given by $N_{\text{total}} = N_f \times N_{\text{nodes}}$. The distribution value corresponding to direction i

at a node located at (n, m), denoted $f_i(n, m)$, can be accessed in memory as

$$f_i(n, m) = f\big[(\, n + m \times (N + 2)\,) + i \times N_{\text{nodes}}\big].$$

In this storage layout, all distributions for different velocity directions are separated in blocks of size N_{nodes}, while node indices are stored contiguously along the n direction.

2.3 Image Read and Preparing Compuational Domain

The following function will be given to you.

```
void LBMInput(const char *imageFileName,
          dfloat threshold,
          int *outN, int *outM,
          unsigned char **rgb,
          unsigned char **alpha,
          int **nodeType)
```

First, the PNG image given by `imageFileName` is loaded:

```
read_png(imageFileName, &N, &M, rgb, alpha);
```

Here,

- N and M are the image dimensions in the horizontal (n) and vertical (m) directions,
- `rgb` contains red–green–blue color channels,
- `alpha` contains transparency values.

To ensure space for upstream inflow and downstream wake development, the original domain is padded:

$$N_{\text{pad}} = \alpha_n N, \qquad M_{\text{pad}} = \alpha_m M,$$

with $\alpha_n = 4$ and $\alpha_m = 2$. The padded dimensions are then clamped to a maximum of 8192 grid points for memory safety.

2.3.1 Allocate Node Types

An integer array `nodeType` of size $(N_{\text{pad}} + 2)(M_{\text{pad}} + 2)$ is allocated:

```
*nodeType = (int*) calloc((Npad+2)*(Mpad+2), sizeof(int));
```

This stores either `FLUID` or `WALL`, enabling easy classification throughout the solver.

2.3.2 Padding and Centering the Image

The original pixel data are copied into the padded RGB and alpha arrays:

```
unsigned char *rgbPad   = ...
unsigned char *alphaPad = ...
```

Each pixel is shifted so that the original geometry appears centered inside the larger computational grid, leaving space for inflow/outflow. A serial index is computed using a macro `idx`:

$$\texttt{id} = \texttt{idx}(N_{\text{pad}}, n, m)$$

2.3.3 Wall Detection by Threshold

The grayscale magnitude of a pixel is estimated from the RGB channels:

$$g = \sqrt{r^2 + g^2 + b^2}$$

A pixel is marked as a wall if:

$$g < \texttt{threshold}$$

Otherwise, it is considered fluid:

```
(*nodeType)[id] = WALL * ( sqrt(r*r+g*g+b*b) < threshold );
```

Transparent pixels (alpha = 0) are always assumed fluid. The variable `wallCount` accumulates the total number of wall nodes for reporting.

2.3.4 Boundary Enforcement

To guarantee solid walls at the top and bottom boundaries, the following rows are forcibly set as walls:

```
for (n=1; n<=Npad; ++n) {
    nodeType(idx(Npad,n,1))   = WALL;
    nodeType(idx(Npad,n,Mpad)) = WALL;
}
```

This enforces no-slip walls on the horizontal boundaries along with the detected walls from the image.

2.3.5 Memory Cleanup and Reassignment

After processing, the old `rgb` and `alpha` arrays are freed, and replaced by the padded versions. Finally, the padded dimensions are returned to the caller.These updated dimensions are assigned N and M, and determine the simulation lattice size.

```
free(*rgb);
free(*alpha);
*rgb = rgbPad;
*alpha = alphaPad;
*outN = Npad;
*outM = Mpad;
```

```
void LBMInput(const char *imageFileName,
        dfloat threshold,
        int *outN,
        int *outM,
        unsigned char **rgb,
        unsigned char **alpha,
        int **nodeType){
        int n,m, N,M;
// read png file and size of box
read_png(imageFileName, &N, &M, rgb, alpha);
// pad to guarantee space around obstacle and extend the wake
// int Npad = 4*N; // Donwind side
dfloat alpha_n = 4;
dfloat alpha_m = 2;
int Npad = alpha_n*N; // Downwind side
int Mpad = alpha_m*M; // Upwind side

// Round the size of domain
if(Npad>8192) Npad = 8192;
if(Mpad>8192) Mpad = 8192;

// threshold walls based on gray scale
*nodeType = (int*) calloc((Npad+2)*(Mpad+2), sizeof(int));

// mark pixels by gray scale intensity
unsigned char *rgbPad=(unsigned char*)calloc(3*(Npad+2)*(Mpad+2),sizeof(unsigned char));
unsigned char *alphaPad=(unsigned char*)calloc((Npad+2)*(Mpad+2),sizeof(unsigned char));
int wallCount = 0;
for(m=1;m<=M;++m){
    for(n=1;n<=N;++n){
```

```
        int offset = ((n-1)+(m-1)*N);
        dfloat r = (*rgb)[3*offset+0];
        dfloat g = (*rgb)[3*offset+1];
        dfloat b = (*rgb)[3*offset+2];
        dfloat a = (*alpha) ? (*alpha)[offset]:255;
      // Center image in padded region (including halo zone)
        int id = idx(Npad,n+(N/2),m+(M/2));
        if(a==0){
            (*nodeType)[id] = FLUID;
        }else{
            (*nodeType)[id] = WALL*(sqrt(r*r+g*g+b*b)<threshold);
        }
        wallCount += (*nodeType)[id];
        rgbPad[3*id+0] = r;
        rgbPad[3*id+1] = g;
        rgbPad[3*id+2] = b;
        alphaPad[id] = 255;
    }
}
for(n=1;n<=Npad;++n){
    (*nodeType)[idx(Npad,n,1)] = WALL;
    (*nodeType)[idx(Npad,n,Mpad)] = WALL;
}
free(*rgb); free(*alpha);
*rgb = rgbPad;
*alpha = alphaPad;

printf("wallCount = %d (%g percent of %d x %d nodes)\n",
      wallCount, 100.*((dfloat)wallCount/((Npad+2)*(Mpad+2))), Npad, Mpad);

fclose(test);
*outN = Npad;
*outM = Mpad;
}
```

2.4 Output

The function `lbmOutput` is responsible for converting the microscopic lattice–Boltzmann distribution function f_i into macroscopic flow variables, computing derived diagnostic quantities (such as vorticity), mapping them into RGB pixel values, and exporting the result into a PNG image. It operates on a $(N+2) \times (M+2)$ lattice that includes ghost (halo) nodes.

The function accepts the lattice geometry, the node–type map (`FLUID` vs. `WALL`), the distribution function f_i, and arrays used to store RGBA pixel values. At output, the `rgb`

and `alpha` arrays are overwritten to visualize field information.

```
void lbmOutput(const char *fname,
               const int *nodeType,
               unsigned char *rgb,
               unsigned char *alpha,
               const dfloat c,
               const dfloat dx,
               int N,
               int M,
               const dfloat *f)
```

Opening Output File

A file handle is opened for writing the output image: `write_png` will use this stream to generate a PNG file.

```
FILE *bah = fopen(fname, "w");
```

Velocity Field Allocation

Two temporary arrays of length $(N+2)(M+2)$ are allocated:

- `Ux`: stores the macroscopic velocity component u_x at each node,
- `Uy`: stores the macroscopic velocity component u_y at each node.

They are initialized to zero via `calloc`.

```
dfloat *Ux = (dfloat*) calloc((N+2)*(M+2), sizeof(dfloat));
dfloat *Uy = (dfloat*) calloc((N+2)*(M+2), sizeof(dfloat));
```

Reconstruction of Macroscopic Velocity

A nested loop traverses the interior computational nodes ($n = 1, \ldots, N$ and $m = 1, \ldots, M$). For each lattice node, the nine distribution components f_i (for D2Q9) are gathered from memory into a temporary buffer. The macroscopic density is then recovered via

$$\rho = \sum_{i=0}^{8} f_i. \tag{5}$$

The macroscopic velocity vector (u_x, u_y) is computed from the discrete velocity moments. For the D2Q9 model, the expressions are

$$u_x = \frac{c}{\rho}(f_1 - f_3 + f_5 - f_6 - f_7 + f_8), \tag{6}$$

$$u_y = \frac{c}{\rho}(f_2 - f_4 + f_5 + f_6 - f_7 - f_8), \tag{7}$$

where c is the lattice scaling parameter. The resulting velocities are stored in `Ux` and `Uy`.

```
for(m=1;m<=M;++m){
  for(n=1;n<=N;++n){
    int base = idx(N, n, m);
    for(s=0;s<NSPECIES;++s)
      fnm[s] = f[base + s*(N+2)*(M+2)];
    const dfloat rho = sum_{s} fnm[s];
    Ux[base] = (...) * c / rho;
    Uy[base] = (...) * c / rho;
  }
}
```

Plotting Range

Two user–defined constants are selected:

$$\texttt{plotMin} = -10, \qquad \texttt{plotMax} = 10,$$

which are later used to normalize scalar values into the $[0, 1]$ range for color mapping. A second nested loop again iterates over the interior nodes. Only `FLUID` nodes are visualized; `WALL` nodes preserve their previous pixel values.

The two–dimensional vorticity (scalar curl) is computed using central finite differences:

$$\omega = \frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y}. \tag{8}$$

With discrete spacing Δx, the derivatives are approximated as

$$\frac{\partial u_y}{\partial x} \approx \frac{u_y(n+1, m) - u_y(n-1, m)}{2\Delta x}, \tag{9}$$

$$\frac{\partial u_x}{\partial y} \approx \frac{u_x(n, m+1) - u_x(n, m-1)}{2\Delta x}. \tag{10}$$

The computed vorticity is linearly normalized into $[0, 1]$:

$$\omega_{\text{norm}} = \frac{\omega - \texttt{plotMin}}{\texttt{plotMax} - \texttt{plotMin}}. \tag{11}$$

In the current configuration of the code, a grayscale color map is used:

$$R = G = B = 255\,\omega_{\text{norm}}, \qquad A = 255,$$

resulting in lighter pixels for regions of high vorticity magnitude. An alternative diverging red–blue colormap is present in the source, guarded by a preprocessor directive, and can be enabled to visualize both sign and magnitude of vorticity.

```
dfloat plotMin = -10, plotMax = 10;
for(m=1;m<=M;++m){
    for(n=1;n<=N;++n){
        int id = idx(N,n,m);
        // over write pixels in fluid region
        if(nodeType[id]==FLUID){
            unsigned char r,g,b,a;
            // reconstruct macroscopic density
            dfloat rho = 0;
            for(s=0;s<NSPECIES;++s)
                rho += f[id+s*(N+2)*(M+2)];

            rho          = ((rho-plotMin)/(plotMax-plotMin)); // rescale
            dfloat dUxdy = (Ux[idx(N,n,m+1)]-Ux[idx(N,n,m-1)])/(2.*dx);
            dfloat dUydx = (Uy[idx(N,n+1,m)]-Uy[idx(N,n-1,m)])/(2.*dx);

            dfloat curlU = dUydx-dUxdy;
            curlU = ((curlU-plotMin)/(plotMax-plotMin));
    #if GRAYSCALE
            r = 255*curlU;
            g = 255*curlU;
            b = 255*curlU;
            a = 255;
    #else
          a = 255;
          if(curlU>.55){r = 255*(curlU-.55)/.45; g = 0; b = 0;}
          else if(curlU<.45){r = 0; g = 0; b = 255*(.45-curlU)/.45;}
          else{r = 255;g = 255;b = 255;}
    #endif
            rgb[idx(N,n,m)*3+0] = r;
            rgb[idx(N,n,m)*3+1] = g;
            rgb[idx(N,n,m)*3+2] = b;
            alpha[idx(N,n,m)] = a;
        }
    }
}
```

For each fluid node, the red, green, and blue components are assigned to the `rgb` array at positions associated with the node index. The opacity (alpha channel) is also stored.

PNG Output and Cleanup

After all pixels are updated, the function calls `write_png`, passing dimensions $(N+2) \times (M+2)$ along with the computed RGBA arrays. The file handle is then closed, and the temporary velocity arrays are freed to prevent memory leaks.

```
write_png(bah, N+2, M+2, rgb, alpha);
fclose(bah);
free(Ux);
free(Uy);
```