

Lattice Boltzmann Equation: D2Q9 Model Simulation

The following program `lbe.cpp` simulates two-dimensional flow in a channel with a linear obstacle using the Lattice Boltzmann Equation in the Bhatnagar-Gross-Krook approximation. The program is based on `lbe.f` by Dr. Sauro Succi.

```
// Lattice Boltzmann Equation: D2Q9 Model Simulation
// Flow in a channel with a linear obstacle

#include <cmath>
#include <cstdlib>
#include <iostream>

using namespace std;

#include <GL/glut.h>

int Nx = 64;           // number of cells in x direction
int Ny = 32;           // number of cells in y direction

int Nc = 9;            // number of discrete velocities
enum direction {
    East = 1, North, West, South,
    NorthEast, NorthWest, SouthWest, SouthEast
};

double ***n;           // number densities
double ***neq;         // equilibrium number densities
double ***u;           // fluid velocity
```

```

void allocate() {
    static int oldNx = 0, oldNy = 0;
    if (Nx != oldNx) {
        if (n != 0) {
            for (int x = 0; x < oldNx+2; x++) {
                for (int y = 0; y < oldNy+2; y++) {
                    delete [] n[x][y]; delete [] neq[x][y]; delete [] u[x][y];
                }
                delete [] n[x]; delete [] neq[x]; delete [] u[x];
            }
        }
        delete [] n; delete [] neq; delete [] u;
        oldNx = Nx;
        oldNy = Ny;
        n = new double** [Nx+2];
        neq = new double** [Nx+2];
        u = new double** [Nx+2];
        for (int x = 0; x < Nx+2; x++) {
            n[x] = new double* [Ny+2];
            neq[x] = new double* [Ny+2];
            u[x] = new double* [Ny+2];
            for (int y = 0; y < Ny+2; y++) {
                n[x][y] = new double [Nc];
                neq[x][y] = new double [Nc];
                u[x][y] = new double [2];
            }
        }
    }
}

```

```
double rho;                // fluid density
double u0[2];              // initial fluid velocity
double uf[2];              // final fluid velocity
double tau;                // relaxation time
int nObstacle;             // extent of obstacle in y direction

void getInput() {

    cout << " Enter initial fluid velocity u0_x: ";
    cin >> u0[0]; u0[1] = 0;
    cout << " Enter fluid density rho: ";
    cin >> rho;
    cout << " Enter final fluid velocity uf_x: ";
    cin >> uf[0]; uf[1] = 0;
    cout << " Enter relaxation time tau: ";
    cin >> tau;
    cout << " Enter extent of obstacle in y direction: ";
    cin >> nObstacle;
}

const double w0 = 16 / 36.0,
              w1 = 4 / 36.0,
              w2 = 1 / 36.0;
const double w[9] = {      // directional weights
    w0,                    // zero
    w1, w1, w1, w1,        // east, north, west, south
    w2, w2, w2, w2        // north-east, north-west, south-west, south-east
};
```

```
const double c[9][2] = {           // particle velocities
    {0, 0},                        // zero
    {1, 0}, {0, 1},                // east, north
    {-1, 0}, {0, -1},              // west, south
    {1, 1}, {-1, 1},               // north-east, north-west
    {-1, -1}, {1, -1}              // south-west, south-east
};

const double csSqd = 1 / 3.0; // speed of sound squared

void initialize_u() {
    for (int x = 0; x < Nx; x++)
        for (int y = 0; y < Ny; y++) {
            for (int d = 0; d < 2; d++)
                u[x][y][d] = u0[d];
        }
}

void compute_neq() {
    for (int x = 0; x < Nx+2; x++)
        for (int y = 0; y < Ny+2; y++) {
            double uSqd = u[x][y][0] * u[x][y][0] + u[x][y][1] * u[x][y][1];
            uSqd /= 2 * csSqd;
            for (int i = 0; i < Nc; i++) {
                double uci = u[x][y][0] * c[i][0] + u[x][y][1] * c[i][1];
                uci /= csSqd;
                neq[x][y][i] = rho * w[i] * (1 + uci * (1 + uci / 2) - uSqd);
            }
        }
}
```

```
void initialize_n() {
    for (int x = 0; x < Nx + 2; x++)
        for (int y = 0; y < Ny + 2; y++)
            for (int i = 0; i < Nc; i++)
                n[x][y][i] = neq[x][y][i];
}

void initialize() {
    getInput();
    allocate();
    initialize_u();
    compute_neq();
    initialize_n();
}

void boundaryConditions() {

    // periodic boundary conditions for east moving particles at east/west ends
    for (int y = 1; y <= Ny; y++) {
        n[0][y][East] = n[Nx][y][East];
        n[0][y][NorthEast] = n[Nx][y][NorthEast];
        n[0][y][SouthEast] = n[Nx][y][SouthEast];
    }

    // periodic boundary conditions for west moving particles at east/west ends
    for (int y = 1; y <= Ny; y++) {
        n[Nx + 1][y][West] = n[1][y][West];
        n[Nx + 1][y][NorthWest] = n[1][y][NorthWest];
    }
}
```

```
    n[Nx + 1][y][SouthWest] = n[1][y][SouthWest];
}

// no-slip boundary conditions for north moving particles at the top end
for (int x = 1; x <= Nx; x++) {
    n[x][Ny + 1][South] = n[x][Ny][North];
    n[x][Ny + 1][SouthEast] = n[x][Ny][NorthWest];
    n[x][Ny + 1][SouthWest] = n[x][Ny][NorthEast];
}

// no-slip boundary conditions for south moving particles at the bottom end
for (int x = 1; x <= Nx; x++) {
    n[x][0][North] = n[x][1][South];
    n[x][0][NorthWest] = n[x][1][SouthEast];
    n[x][0][NorthEast] = n[x][1][SouthWest];
}
}

void freeStream() {

    // start at NW corner and stream north and north-west moving particles
    for (int x = 1; x <= Nx; x++)
    for (int y = Ny; y >= 1; y--) {
        n[x][y][North] = n[x][y - 1][North];
        n[x][y][NorthWest] = n[x + 1][y - 1][NorthWest];
    }

    // start at NE corner and stream east and north-east moving particles
    for (int x = Nx; x >= 1; x--)
```

```
for (int y = Ny; y >= 1; y--) {
    n[x][y][East] = n[x - 1][y][East];
    n[x][y][NorthEast] = n[x - 1][y - 1][NorthEast];
}

// start at SE corner and stream south and south-east moving particles
for (int x = Nx; x >= 1; x--)
for (int y = 1; y <= Ny; y++) {
    n[x][y][South] = n[x][y + 1][South];
    n[x][y][SouthEast] = n[x - 1][y + 1][SouthEast];
}

// start at SW corner and stream west and south-west moving particles
for (int x = 1; x <= Nx; x++)
for (int y = 1; y <= Ny; y++) {
    n[x][y][West] = n[x + 1][y][West];
    n[x][y][SouthWest] = n[x + 1][y + 1][SouthWest];
}
}

void compute_u() {

    for (int x = 1; x <= Nx; x++)
    for (int y = 1; y <= Ny; y++)
    for (int d = 0; d < 2; d++) {
        u[x][y][d] = 0;
        for (int i = 1; i < Nc; i++)
            u[x][y][d] += n[x][y][i] * c[i][d];
        u[x][y][d] /= rho;
    }
}
```

```
    }  
}  
  
void collide() {  
  
    for (int x = 1; x <= Nx; x++)  
    for (int y = 1; y <= Ny; y++)  
    for (int i = 0; i <= Nc; i++)  
        n[x][y][i] -= (n[x][y][i] - neq[x][y][i]) / tau;  
}  
  
void force() {  
  
    double nu = csSqd * (tau - 0.5);  
    double f = 8 * nu * uf[0] * rho / (6.0 * Ny * Ny);  
  
    for (int x = 1; x <= Nx; x++)  
    for (int y = 1; y <= Ny; y++) {  
        n[x][y][East] += f;  
        n[x][y][NorthEast] += f;  
        n[x][y][SouthEast] += f;  
  
        n[x][y][West] -= f;  
        n[x][y][NorthWest] -= f;  
        n[x][y][SouthWest] -= f;  
    }  
}  
  
void obstacle() {
```

```
int x = Nx / 4;
int y1 = Ny / 2 - nObstacle / 2;
int y2 = Ny / 2 + nObstacle / 2;

for (int y = y1 + 1; y <= y2; y++) {
    n[x + 1][y][East] = n[x + 1][y][West];
    n[x][y][West] = n[x][y][East];
}

for (int y = y1; y <= y2 + 1; y++) {
    n[x + 1][y][NorthEast] = n[x + 1][y][SouthWest];
    n[x + 1][y][SouthEast] = n[x + 1][y][NorthWest];
    n[x][y][SouthWest] = n[x][y][NorthEast];
    n[x][y][NorthWest] = n[x][y][SouthEast];
}
}

int step = 0;

void takeStep() {

    boundaryConditions();
    freeStream();
    compute_u();
    compute_neq();
    collide();
    force();
    if (nObstacle > 1)
```

```
        obstacle();

    glutPostRedisplay();
}

const double toDegrees = 180.0 / (4 * atan(1.0));

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3ub(0, 0, 255);
    glRectd(0, - 0.05 * (Ny + 2), Nx + 2, 0);
    glRectd(0, Ny + 1, Nx + 2, (Ny + 1) * 1.05);
    if (nObstacle > 1) {
        glColor3ub(0, 255, 0);
        int x = Nx / 4, y1 = Ny / 2 - nObstacle / 2,
            y2 = Ny / 2 + nObstacle / 2;
        glRectd(x, y1, x + 1, y2);
    }

    double uMax = 0;
    for (int x = 0; x < Nx + 2; x++)
        for (int y = 0; y < Ny + 2; y++) {
            double uMag = sqrt(u[x][y][0] * u[x][y][0] + u[x][y][1] * u[x][y][1]);
            if (uMag > uMax)
                uMax = uMag;
        }
    if (uMax == 0) uMax = 1;

    glColor3ub(255, 0, 0);
```

```
for (int x = 0; x < Nx + 2; x++)
for (int y = 0; y < Ny + 2; y++) {
    glPushMatrix();
        double uMag = sqrt(u[x][y][0] * u[x][y][0] +
                           u[x][y][1] * u[x][y][1]);
        double scale = uMag / uMax;
        double angle = atan2(u[x][y][1], u[x][y][0]) * toDegrees;
        glTranslated(x, y, 0);
        glScaled(scale, scale, 1);
        glRotated(angle, 0, 0, 1);
        glBegin(GL_LINES);
            glVertex2d(0, 0); glVertex2d(1, 0);
            glVertex2d(1, 0); glVertex2d(0.8, 0.2);
            glVertex2d(1, 0); glVertex2d(0.8, -0.2);
        glEnd();
    glPopMatrix();
}

glutSwapBuffers();
}

void reshape(int w, int h) {
    int x0 = 0, y0 = 0, dx = w, dy = h;
    double aspect = (w * (Ny + 2) * 1.1) / (h * (Nx + 2));
    if (aspect > 1) {
        dx = int(dx / aspect);
        x0 = (w - dx) / 2;
    } else {
        dy = int(dy * aspect);
    }
}
```

```
        y0 = (h - dy) / 2;
    }
    glViewport(x0, y0, dx, dy);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, Nx + 2, - 0.05 * (Ny + 2), (Ny + 2) * 1.05);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

}

void mouse(int button, int state, int x, int y) {
    switch (button) {
    case GLUT_LEFT_BUTTON:
        if (state == GLUT_DOWN)
            ;
        break;
    default:
        break;
    }
}

int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    if (argc > 1) {
        Nx = atoi(argv[1]);
        Ny = atoi(argv[2]);
    }
    initialize();
}
```

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);  
glutInitWindowSize(600, 330);  
glutInitWindowPosition(100, 100);  
glutCreateWindow("Lattice Boltzmann Equation: D2Q9 Model");  
glClearColor(1.0, 1.0, 1.0, 0.0);  
glShadeModel(GL_FLAT);  
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);  
glutDisplayFunc(display);  
glutReshapeFunc(reshape);  
glutIdleFunc(takeStep);  
glutMouseFunc(mouse);  
glutMainLoop();  
}
```