## Fluid Simulation using Lattice Boltzmann Method (LBM) and ASCII characters

This C++ code demonstrates a simple 1D fluid simulation in the terminal using the Lattice Boltzmann Method (LBM) and ASCII characters for visualization. Please note that this is a basic example for educational purposes and may not accurately model real fluid dynamics.

### Code Overview

The code defines a class `LBMFluidSimulation` responsible for the fluid simulation. Here's a brief overview of its key components:

1. **Simulation Parameters**: The class contains variables for simulation parameters like `tau` (relaxation time), `omega` (relaxation parameter), and `rho0` (reference density).
2. **Data Representation**: The fluid simulation is represented using arrays for density `rho` and velocity components `ux` and `uy`.
3. **Lattice Directions**: The code defines lattice velocity vectors `c`, which represents the directions of particle motion in LBM.
4. **Initialization**: The function `setInitialConditions` initializes the density and velocity fields to their initial values.
5. **LBM Collision**: The function `collide` updates the distribution functions (`f`) according to the LBM collision step.
6. **LBM Streaming**: The function `stream` performs the LBM streaming step, moving the particles in the specified directions.
7. **Density and Velocity Calculation**: The function `computeDensityVelocity` calculates the density and velocity fields from the updated distribution functions.
8. **Visualization**: The function `display` prints the simulation results in the terminal using ASCII characters. The velocity magnitude is used to determine the symbol to display.

### How to Run

1. Compile the C++ code using a C++ compiler (e.g., g++).
2. Run the executable.
3. The terminal will display the fluid simulation using ASCII characters. The velocity magnitude determines the character displayed for each cell.

### Code Block

Here's the C++ code for the fluid simulation using LBM:

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <chrono>
#include <thread>

const int width = 80;          // Width of the terminal window
const int height = 20;        // Height of the terminal window
const int latticeSize = 9;    // Number of lattice directions in LBM

// Lattice velocity vectors
const int c[latticeSize][2] = {
    {0, 0}, {1, 0}, {0, 1}, {-1, 0}, {0, -1},
    {1, 1}, {-1, 1}, {-1, -1}, {1, -1}
};

class LBMFluidSimulation {
private:
    std::vector<double> f[9]; // LBM distribution functions
    double rho[height][width]; // Density
    double ux[height][width];  // x-component of velocity
    double uy[height][width];  // y-component of velocity

    // Simulation parameters
    double tau;  // Relaxation time
    double omega; // Relaxation parameter
    double rho0; // Reference density

public:
    LBMFluidSimulation() : tau(0.6), omega(1.0 / tau), rho0(1.0) {
        for (int i = 0; i < latticeSize; ++i) {
            f[i].resize(height * width, 0.0);
        }
    }

    // Set initial conditions for density and velocity
    void setInitialConditions() {
        for (int y = 0; y < height; ++y) {
            for (int x = 0; x < width; ++x) {
                rho[y][x] = rho0;
                ux[y][x] = 0.0;
                uy[y][x] = 0.0;
            }
        }
    }
```

```
// LBM collision step
void collide() {
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            double cu, u2;
            for (int i = 0; i < latticeSize; ++i) {
                cu = c[i][0] * ux[y][x] + c[i][1] * uy[y][x];
                u2 = ux[y][x] * ux[y][x] + uy[y][x] * uy[y][x];
                f[i][x + y * width] = omega * (3.0 * cu / c_sq + 4.5 * cu * cu / c_sq_sc
            }
        }
    }
}

// LBM streaming step
void stream() {
    std::vector<double> f_new[9];
    for (int i = 0; i < latticeSize; ++i) {
        f_new[i].resize(height * width, 0.0);
    }

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            for (int i = 0; i < latticeSize; ++i) {
                int x2 = (x - c[i][0] + width) % width;
                int y2 = (y - c[i][1] + height) % height;
                f_new[i][x + y * width] = f[i][x2 + y2 * width];
            }
        }
    }

    for (int i = 0; i < latticeSize; ++i) {
        f[i] = f_new[i];
    }
}

// Compute density and velocity from distribution functions
void computeDensityVelocity() {
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            double localRho = 0.0;
            double localUx = 0.0;
            double localUy = 0.0;

            for (int i = 0; i < latticeSize; ++i) {
                localRho += f[i][x + y * width];
```

```
                localUx += c[i][0] * f[i][x + y * width];
                localUy += c[i][1] * f[i][x + y * width];
            }

            rho[y][x] = localRho;
            ux[y][x] = localUx / localRho;
            uy[y][x] = localUy / localRho;
        }
    }
}

// Print the simulation results in ASCII characters
void display() {
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            char symbol = ' ';
            double velocityMagnitude = std::sqrt(ux[y][x] * ux[y][x] + uy[y][x] * uy[y]

            if (velocityMagnitude > 0.1) {
                symbol = '*';
            }
            std::cout << symbol;
        }
        std::cout << std::endl;
    }
}
};

int main() {
    LBMFluidSimulation fluidSimulation;
    fluidSimulation.setInitialConditions();

    for (int t = 0; t < 500; ++t) {
        fluidSimulation.collide();
        fluidSimulation.stream();
        fluidSimulation.computeDensityVelocity();
        fluidSimulation.display();

        // Delay to slow down the animation
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }

    return 0;
}
```