

THE FHP LATTICE GAS CELLULAR AUTOMATON FOR SIMULATING FLUID FLOWS*

Anthony Austin

This work is produced by OpenStax-CNX and licensed under the
Creative Commons Attribution License 3.0[†]

Abstract

This report summarizes work done as part of the Physics of Strings PFUG under Rice University's VIGRE program. VIGRE is a program of Vertically Integrated Grants for Research and Education in the Mathematical Sciences under the direction of the National Science Foundation. A PFUG is a group of Postdocs, Faculty, Undergraduates and Graduate students formed round the study of a common problem. This module discusses the use of the FHP lattice gas cellular automaton for simulating fluid flows based on the description in (CITE). Code for simulating flow past an obstacle in a rectangular channel with periodic boundary conditions at the inflow and outflow edges is provided.

1 Introduction

Solving fluid flows is an important everyday task for engineers, physicists, and applied mathematicians. It can also be rather complex, especially when simulating flows with high Reynolds numbers. The classic approach to tackling this problem is to solve the Navier-Stokes equations directly using, for example, finite element or finite difference methods, but for domains with complicated geometries, it can be difficult to find and implement a suitable mesh over which to apply these techniques. In cases like these, it may be more desirable to accurately simulate the behavior of the flow instead of solving for it outright. To this end, scientists and mathematicians have devised lattice gas and lattice Boltzmann methods for modeling fluid flow. In our VIGRE seminar this semester, we implemented a basic 2-D version of the FHP lattice gas cellular automaton (LGCA) with an eye towards extending this model to simulate string motion in fluid flow.

2 Brief Description of the FHP Model

The LGCA approach to simulating fluid flow involves defining a lattice on which a large number of simulated particles move. Each particle has a mass and a velocity and therefore a momentum. The rules for particle interaction are chosen so that, in the macroscopic limit (i.e., when the momentum vectors are averaged over relatively large subdomains of the entire lattice), the resulting flow obeys the Navier-Stokes equations.

For the basic FHP model in 2-D, the lattice is chosen to have hexagonal symmetry. This means that a particle in the FHP LGCA can move from one node to another with six possible lattice velocities. (These

*Version 1.1: Aug 28, 2009 8:56 am +0000

[†]<http://creativecommons.org/licenses/by/3.0/>

velocities are equal in magnitude but vary with direction. So-called "multi-speed" FHP models allow for even more possibilities, but we do not discuss these here.) The reason for the hexagonal symmetry is that it has been shown that certain key tensors fail to be isotropic on any lattice with a lesser degree of symmetry (e.g., a cubic lattice), which prevents the model from yielding the Navier-Stokes equations in the macroscopic limit ([1], pgs. 38, 51)

When particles meet at a node, it is possible for a collision to occur. The basic FHP model defines only two- and three-particle collisions, but it turns out that this is sufficient to yield the desired behavior. (More sophisticated models may define more complicated collision interactions.) If two particles travelling in opposite directions meet at a node, then the particle pair is randomly rotated either clockwise or counterclockwise by sixty degrees. If three particles meet at a node in a symmetric configuration, then they collide in such a way that this configuration is "inverted." Perhaps an illustration will help clarify:

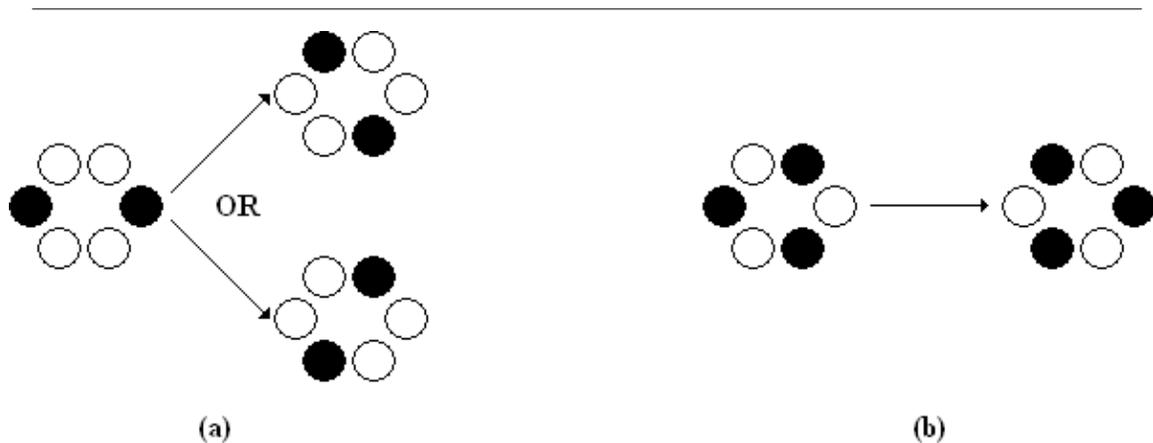


Figure 1: (a) Example of a two-particle collision in the basic FHP model. A node in the lattice is exemplified by a hexagonal arrangement of six cells, one cell for each of the six possible lattice directions. A white cell indicates the absence of a particle at the node moving in that direction, while a filled-in cell indicates the presence of such a particle. (b) Example of a symmetric three-particle collision.

For a more detailed description of the FHP model as well as descriptions of some other possible LGCAs, see [1].

3 Performing a Simulation with the FHP Model

Simulations that use this model proceed in essentially three steps. First, the simulation must be initialized with several parameters, such as the size and shape of the domain, number of timesteps, and the initial placement and velocity of the particles. In the code we provide below, we use a simple rectangular-shaped domain with walls on the upper and lower edges to simulate flow in a channel. We place a single particle at every node in the domain (except at those on walls and other flow obstacles) moving in the rightward direction.

The simulation must also have some way of knowing where the obstacles (e.g., walls) to the fluid flow are located and how the fluid ought to behave when it encounters those obstacles. That is, the boundaries of the domain must be defined, and boundary conditions must be provided. There are several possible types of boundary conditions that can be used; however, the most frequently used boundary conditions are of the no-slip type, and these are the boundary conditions used in the code below. To implement these, one simply

requires that when a particle enters a node at a channel wall or obstacle boundary, it is reflected back in the direction in which it came (i.e., its velocity vector is turned around by 180 degrees). For simplicity, at the left and right edges of our channel (where there are no walls), we have implemented periodic boundary conditions. That is, we have “linked” the left and right sides of the channel together. (In reality, this means that we are no longer simulating flow through a rectangular channel, but rather through a channel of a circular shape, but by choosing the channel dimensions to be sufficiently large, the effects of this on our simulation can be made negligible.) It is possible to implement true inflow and outflow boundary conditions, but these (especially the latter) can be quite complicated.

Next, once the simulation is initialized, it can be carried out. Performing the actual simulation consists of carrying out the appropriate collisions at each node and the propagating (or streaming) the particles to their next nodes in the lattice. This process is repeated for each timestep until all the runs are completed.

Finally, to obtain the actual flow from the simulation, the particle velocities are averaged over large subdomains. Subdomain size can vary, but due to the high susceptibility of LGCA to statistical noise (see “Notes on Performance”) section, below and [1], pg. 157), larger subdomains will yield more accurate, albeit less detailed, results.

4 Results

To test our code, we simulated flows past two different types of obstacles – a flat plate and a circular cylinder – and also examined the effects of choosing different subdomain sizes over which to average. All simulations were carried out on a grid 640 nodes by 256 nodes in dimension with 2000 timesteps. The resulting flow fields were:

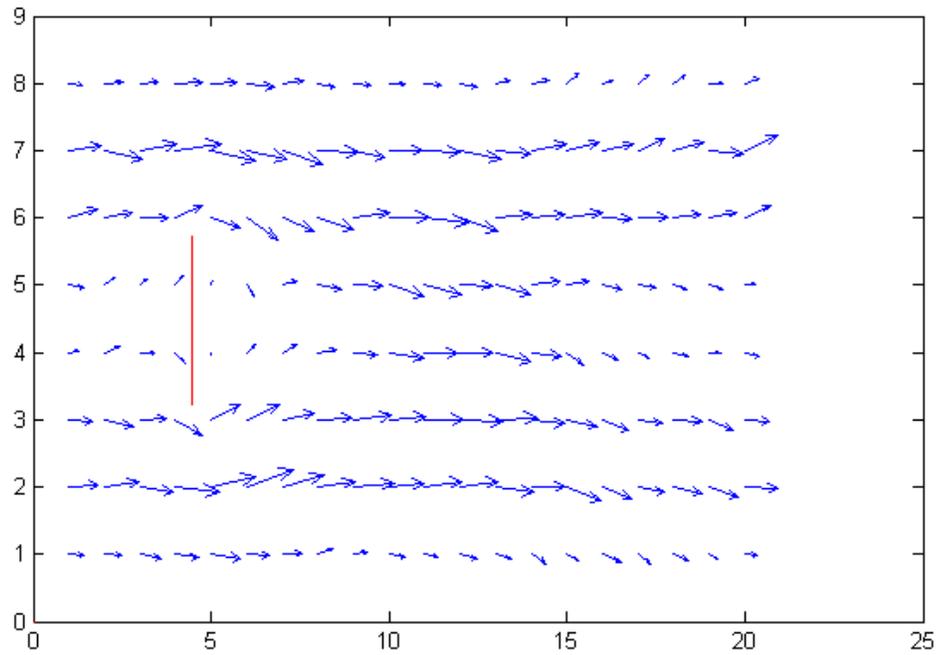


Figure 2: Flow past a flat plate. Subdomain size is 32 nodes by 32 nodes. General shape is clear, but detail is lacking.

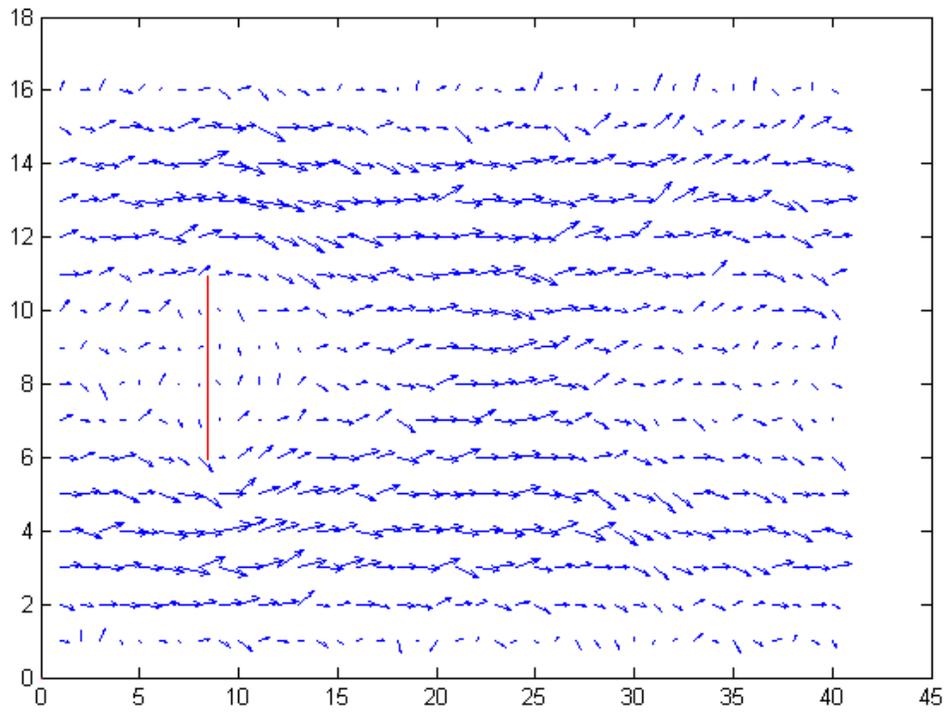


Figure 3: Flow past a flat plate. Subdomain size is 16 nodes by 16 nodes. Already it is possible to see the presence of some statistical noise, but the flow field is more detailed.

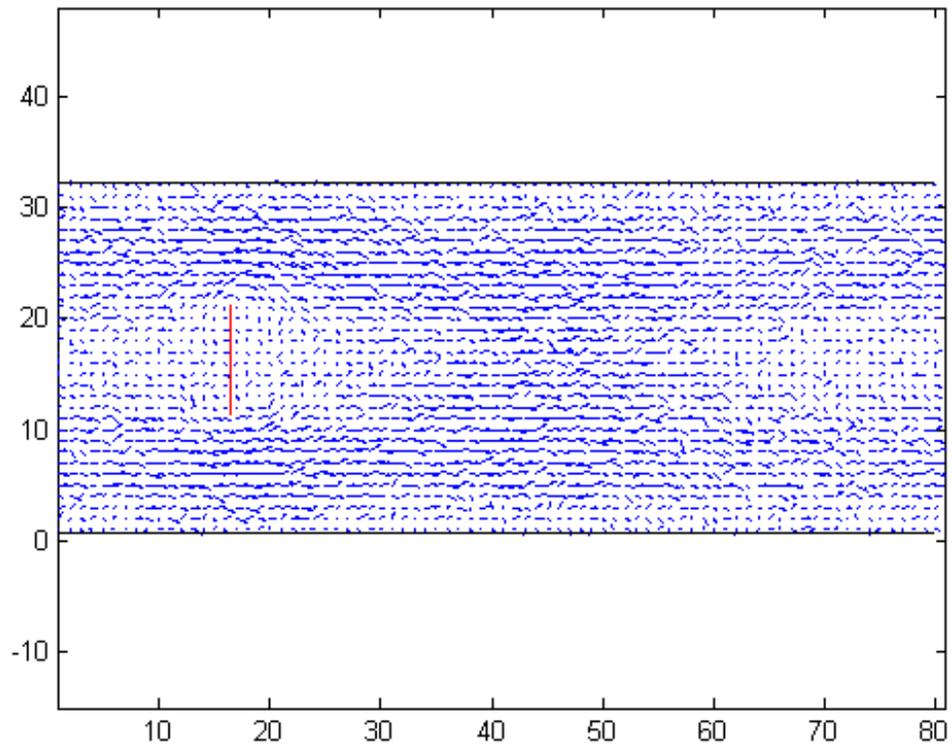


Figure 4: Flow past a flat plate. Subdomain size is 8 nodes by 8 nodes.

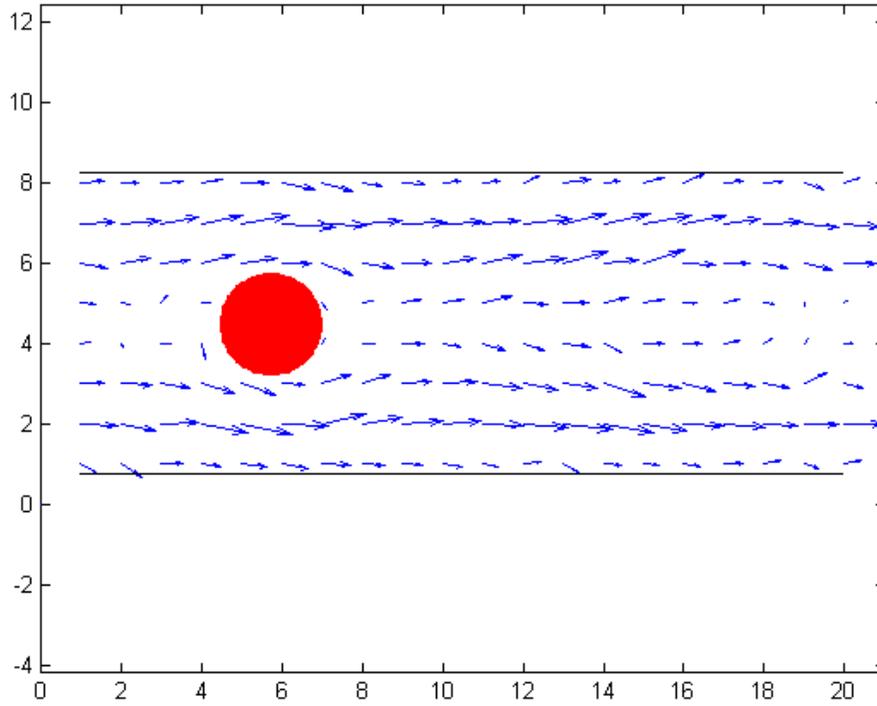


Figure 5: Flow past a circular cylinder. Subdomain size is 32 nodes by 32 nodes.

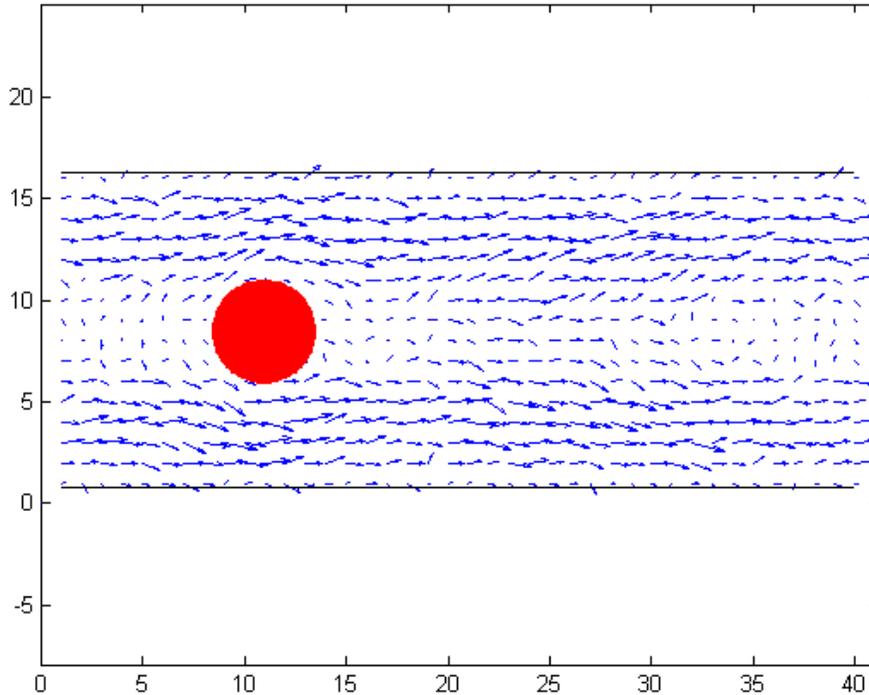


Figure 6: Flow past a circular cylinder. Subdomain size is 16 nodes by 16 nodes.

5 Notes on Performance

Though lattice gas models are convenient, they suffer from several drawbacks, the most notable of which is statistical noise. As the figures in the previous section illustrate, there is a significant tradeoff between flow field accuracy and model detail for a fixed domain size. As with any type of simulation, it is possible to get better, more accurate and detailed results by simply enlarging the domain and taking a greater number of timesteps. Unfortunately, this means a larger consumption of computing resources. Generating each of the above figures using the code below took between three and four hours of time on a Dell Latitude D410 laptop running MATLAB on Windows XP with 1 GB RAM and a 2 GHz Intel processor. To enlarge the domain by 10 times in each dimension and take 10 times as many timesteps, as is done in generating the figures on pages 83-84 of [1], would require an enormous amount of memory and runtime. To make the simulation usable, it will be necessary in the future to improve the simulation algorithm (e.g., by implementing it on a bitwise level as discussed on pg. 42 of [1]). Additionally, this model lends itself easily to parallelization, providing another possibility for performance improvement.

6 Conclusion

We have successfully implemented the basic version of the FHP lattice gas cellular automaton for simulating fluid behavior. Though our present implementation is a bit inefficient, it appears to give the expected results, and we know ways that it may be improved. Assembling this model has allowed us to take a step in the direction of our ultimate goal is of investigating the motion of vibrating strings in fluids.

7 Acknowledgements

This Connexions module describes work conducted as part of Rice University's VIGRE program, supported by National Science Foundation grant DMS-0739420.

8 Appendix: FHP LGCA Code

```

%
% fhp.m -- Uses the FHP LGCA model to simulate the flow of a fluid
%         past a plate in a wide channel with no-slip
%         boundary conditions. This code aims to implement the FHP
%         LGCA as described in_ Lattice Gas Cellular Automata and
%         Lattice Boltzmann Models_ by Wolf-Gladrow. Periodic
%         boundary conditions are assumed at the channel's left
%         and right edges.
%
% WRITTEN BY: Anthony P. Austin, February 11, 2009

function fhp
    tic; % Time program execution.

    % Number of nodes in each direction. These must be multiples of 32
    % for the coarse graining to work.
    numnodes_x = 6400/10;
    numnodes_y = 2560/10;

    % Number of timesteps over which to simulate.
    t_end = 5;

    % 3D array of nodes to store the vectors that represent the occupied
    % cells at each node.
    % 0 - Cell unoccupied.
    % 1 - Cell occupied.
    %
    % 1st Index -- Node x-coordinate.
    % 2nd Index -- Node y-coordinate.
    % 3rd Index -- Cell number
    %
    % The elements of the occupancy vectors correspond to the cells in the
    % following way:
    %
    %           3   2
    %           \ /
    %         4 - 0 - 1

```

```

%           / \
%           5  6
%
% Observe that this convention differs slightly from Wolf-Gladrow's.
%
nodes = zeros(numnodes_x, numnodes_y, 6);

% Define the lattice velocities.
c1 = [1; 0];
c2 = [cos(pi/3); sin(pi/3)];
c3 = [cos(2*pi/3); sin(2*pi/3)];
c4 = [-1; 0];
c5 = [cos(4*pi/3); sin(4*pi/3)];
c6 = [cos(5*pi/3); sin(5*pi/3)];

% Define a matrix that indicates where the flow obstacles are.
% 0 - No obstacle present at that node.
% 1 - Obstacle at the node.
%
% Don't forget to put 1's at the interior points, too!
obstacle = zeros(numnodes_x, numnodes_y);

% Insert a flat plate as the obstacle.
for (j = 880/10:1:1680/10)
    obstacle(1280/10, j) = 1;
end

% Insert a circular cylinder as the obstacle.
%{
theta = 0:0.001:2*pi;
xc = round(168 + 40*cos(theta));
yc = round(128 + 40*sin(theta));

for (i = 1:length(theta))
    obstacle(xc(i), yc(i)) = 1;
end

for (i = 1:numnodes_x)
    currrow = obstacle(i, :);
    n = find(currrow, 1, 'first');
    m = find(currrow, 1, 'last');

    if (~isempty(n))
        for (j = n:1:m)
            obstacle(i, j) = 1;
        end
    end
end
end
%}

% Set up the simulation.

```

```

for (i = 1:1:numnodes_x)
    for (j = 2:1:(numnodes_y - 1)) % Don't include the top and bottom walls.
        % Skip points on the obstacle boundary
        if (obstacle(i, j) ~= 1)
            curr_cell = nodes(i, j, :); % Get the cell for the current node.

            curr_cell(1) = 1;           % Put a particle in the cell flowing in the
                                       % rightward direction.

            nodes(i, j, :) = curr_cell; % Reinsert the cell into the array.
        end
    end
end

% Carry out the simulation.
for (t = 1:1:t_end)

    % Carry out collisions at non-boundary nodes.
    for (i = 1:1:numnodes_x)
        for (j = 2:1:(numnodes_y - 1)) % Don't include the top and bottom walls.
            % Ensure that there's no obstacle in the way.
            if (obstacle(i, j) ~= 1)

                % Extract the current cell.
                cell_oc = nodes(i, j, :);

                % Determine how many particles are in the cell.
                numparts = sum(cell_oc);

                % Determine and execute appropriate collision.
                if ((numparts ~= 2) && (numparts ~= 3)) % No collision occurs.
                    nodes(i, j, :) = cell_oc;
                elseif (numparts == 3) % Three-particle collisions.
                    % We require a symmetric configuration.
                    if ((cell_oc(1) == cell_oc(3)) && (cell_oc(3) == cell_oc(5)))
                        % Invert the cell contents.
                        nodes(i, j, :) = ~cell_oc;
                    else
                        nodes(i, j, :) = cell_oc;
                    end
                else % Two-particle collisions.
                    % Find the cell of one of the particles.
                    p1 = find(cell_oc, 1);

                    % We need its diametric opposite to be occupied as well.
                    if ((p1 > 3) || (cell_oc(p1 + 3) ~= 1))
                        nodes(i, j, :) = cell_oc;
                    else
                        % Randomly rotate the particle pair clockwise or
                        % counterclockwise.
                        r = rand;
                    end
                end
            end
        end
    end
end

```



```
neighbor_x = 0;
neighbor_y = 0;

% Propagation in the 1-direction.
neighbor_y = j;

if (i == numnodes_x)
    neighbor_x = 1;
else
    neighbor_x = i + 1;
end

n_cell_oc = n_nodes(neighbor_x, neighbor_y, :);
n_cell_oc(1) = cell_oc(1);
n_nodes(neighbor_x, neighbor_y, :) = n_cell_oc;

% Propagation in the 2-direction.
if (j ~= numnodes_y)
    neighbor_y = j + 1;

    if (mod(j, 2) == 0)
        if (i == numnodes_x)
            neighbor_x = 1;
        else
            neighbor_x = i + 1;
        end
    else
        neighbor_x = i;
    end

    n_cell_oc = n_nodes(neighbor_x, neighbor_y, :);
    n_cell_oc(2) = cell_oc(2);
    n_nodes(neighbor_x, neighbor_y, :) = n_cell_oc;
end

% Propagation in the 3-direction.
if (j ~= numnodes_y)
    neighbor_y = j + 1;

    if (mod(j, 2) == 1)
        if (i == 1)
            neighbor_x = numnodes_x;
        else
            neighbor_x = i - 1;
        end
    else
        neighbor_x = i;
    end

    n_cell_oc = n_nodes(neighbor_x, neighbor_y, :);
    n_cell_oc(3) = cell_oc(3);
```

```
        n_nodes(neighbor_x, neighbor_y, :) = n_cell_oc;
    end

    % Propagation in the 4-direction.
    neighbor_y = j;

    if (i == 1)
        neighbor_x = numnodes_x;
    else
        neighbor_x = i - 1;
    end

    n_cell_oc = n_nodes(neighbor_x, neighbor_y, :);
    n_cell_oc(4) = cell_oc(4);
    n_nodes(neighbor_x, neighbor_y, :) = n_cell_oc;

    % Propagation in the 5-direction.
    if (j ~= 1)
        neighbor_y = j - 1;

        if (mod(j, 2) == 1)
            if (i == 1)
                neighbor_x = numnodes_x;
            else
                neighbor_x = i - 1;
            end
        else
            neighbor_x = i;
        end

        n_cell_oc = n_nodes(neighbor_x, neighbor_y, :);
        n_cell_oc(5) = cell_oc(5);
        n_nodes(neighbor_x, neighbor_y, :) = n_cell_oc;
    end

    % Propagation in the 6-direction.
    if (j ~= 1)
        neighbor_y = j - 1;

        if (mod(j, 2) == 0)
            if (i == numnodes_x)
                neighbor_x = 1;
            else
                neighbor_x = i + 1;
            end
        else
            neighbor_x = i;
        end

        n_cell_oc = n_nodes(neighbor_x, neighbor_y, :);
        n_cell_oc(6) = cell_oc(6);
    end
end
```

```

                n_nodes(neighbor_x, neighbor_y, :) = n_cell_oc;
            end
        end
    end

    % Propagate the particles to their next nodes.
    nodes = n_nodes;

    % Print the current time step every so often so we know that the
    % program hasn't frozen or crashed.
    if (mod(t, 5) == 0)
        disp(t);
    end
end

% Subdivide the total domain into subdomains of size 32x32 for the
% purposes of coarse-graining. See pg. 51.
grain_size = 8;
grain_x = numnodes_x / grain_size;
grain_y = numnodes_y / grain_size;

% Pre-allocate vectors for the averaged velocities.
av_vel_x_coords = zeros(1, grain_x * grain_y);
av_vel_y_coords = zeros(1, grain_x * grain_y);
av_vel_x_comps = zeros(1, grain_x * grain_y);
av_vel_y_comps = zeros(1, grain_x * grain_y);

% Iterate over the entire domain, averaging and storing the results as
% we go.
currval = 1;
for (i = 1:1:grain_x)
    % Calculate the lower and upper x-boundaries.
    x_bd_l = (i - 1)*grain_size + 1;
    x_bd_u = i*grain_size;
    for (j = 1:1:grain_y)
        % Calculate the lower and upper y-boundaries.
        y_bd_l = (j - 1)*grain_size + 1;
        y_bd_u = j*grain_size;

        % Get the number of particles moving in each direction in the
        % current subdomain.
        np = zeros(1, 6);
        np(1) = sum(sum(nodes(x_bd_l:1:x_bd_u, y_bd_l:1:y_bd_u, 1)));
        np(2) = sum(sum(nodes(x_bd_l:1:x_bd_u, y_bd_l:1:y_bd_u, 2)));
        np(3) = sum(sum(nodes(x_bd_l:1:x_bd_u, y_bd_l:1:y_bd_u, 3)));
        np(4) = sum(sum(nodes(x_bd_l:1:x_bd_u, y_bd_l:1:y_bd_u, 4)));
        np(5) = sum(sum(nodes(x_bd_l:1:x_bd_u, y_bd_l:1:y_bd_u, 5)));
        np(6) = sum(sum(nodes(x_bd_l:1:x_bd_u, y_bd_l:1:y_bd_u, 6)));

        % Compute the average velocity.
        av_vel = (1/(grain_size.^2))*(np(1)*c1 + np(2)*c2 + np(3)*c3 + np(4)*c4 + np(5)*c5 + np(6)*

```

```

        % Store the velocity components.
        av_vel_x_comps(currval) = av_vel(1);
        av_vel_y_comps(currval) = av_vel(2);

        % Store the positional coordinates.
        av_vel_x_coords(currval) = i;
        av_vel_y_coords(currval) = j;

        currval = currval + 1;
    end
end

% Plot the average velocity field.
quiver(av_vel_x_coords, av_vel_y_coords, av_vel_x_comps, av_vel_y_comps);

% Plot the channel boundaries.
hold on;
plot([1; grain_x], [0.75; 0.75], 'k-');
hold on;
plot([1; grain_x], [grain_y + 0.25; grain_y + .25], 'k-');

% Display the flow obstacle.
obstacle_x = zeros(1, nnz(obstacle));
obstacle_y = zeros(1, nnz(obstacle));
k = 1;

for (i = 1:1:numnodes_x)
    for (j = 1:1:numnodes_y)
        if (obstacle(i, j) == 1)
            obstacle_x(k) = 0.5 + (numnodes_x ./ (grain_size .* (numnodes_x - 1))) .* (i - 1);
            obstacle_y(k) = 0.5 + (numnodes_y ./ (grain_size .* (numnodes_y - 1))) .* (j - 1);
            k = k + 1;
        end
    end
end

hold on;
plot(obstacle_x, obstacle_y, 'r-');
axis equal;

toc; % Print the time it took to execute.
end

```

References

- [1] Dieter A. Wolf-Gladrow. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models 8211; An Introduction*. Springer, Berlin, 2005.