Computational Fluid Dynamics: The Lattice Boltzmann Method

What's a fluid?

A fluid …

- … is a substance that continually deforms under stresses.
- … resists deformations only lightly, because of viscosity.
- … can adopt the shape of any container into which it flows.

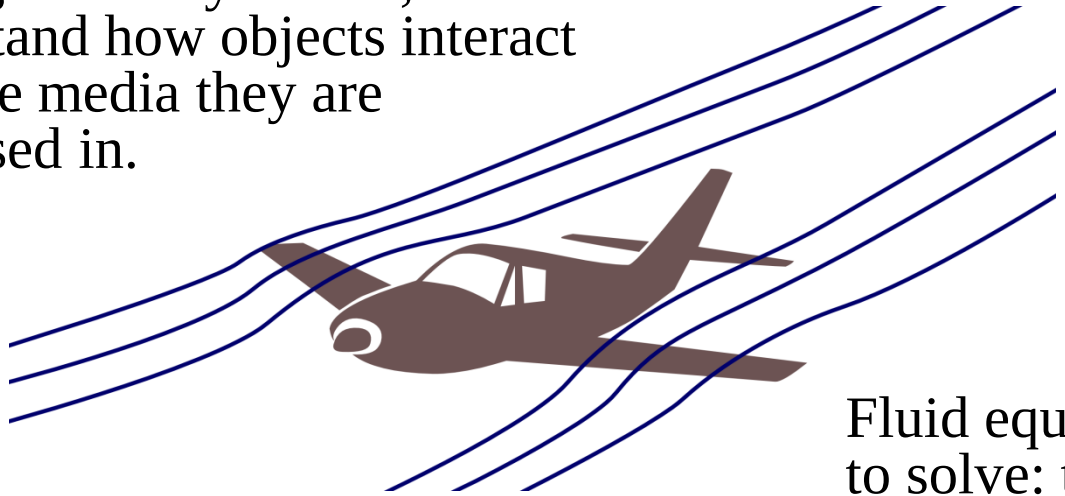
Fluids include:

- Liquids
- Gases
- Plasmas

Fluid = Continuum Model

Why fluid dynamics?

Through fluid dynamics, we understand how objects interact with the media they are immersed in.



Fluid equations are difficult to solve: this is a case for numerical simulation.

Fluid dynamics is everywhere

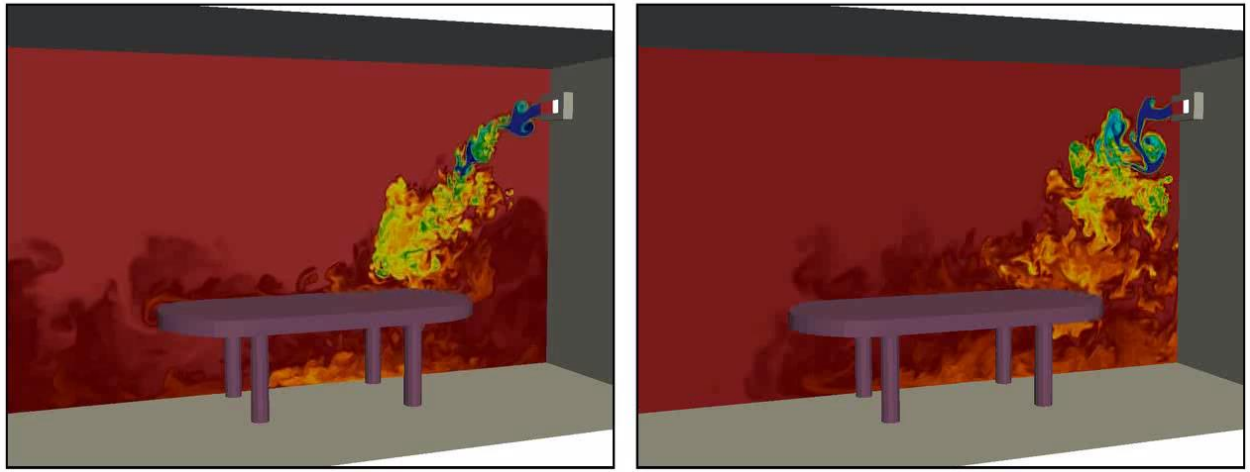
Important application areas are found

- In physics.
- In biology and medical physics.
- In chemistry.
- In geology.
- … and many more.

Example 1: Heat convection

Air-conditioner in a meeting room

palabos.org



Fixed air-conditioner

Sweeping air-conditioner

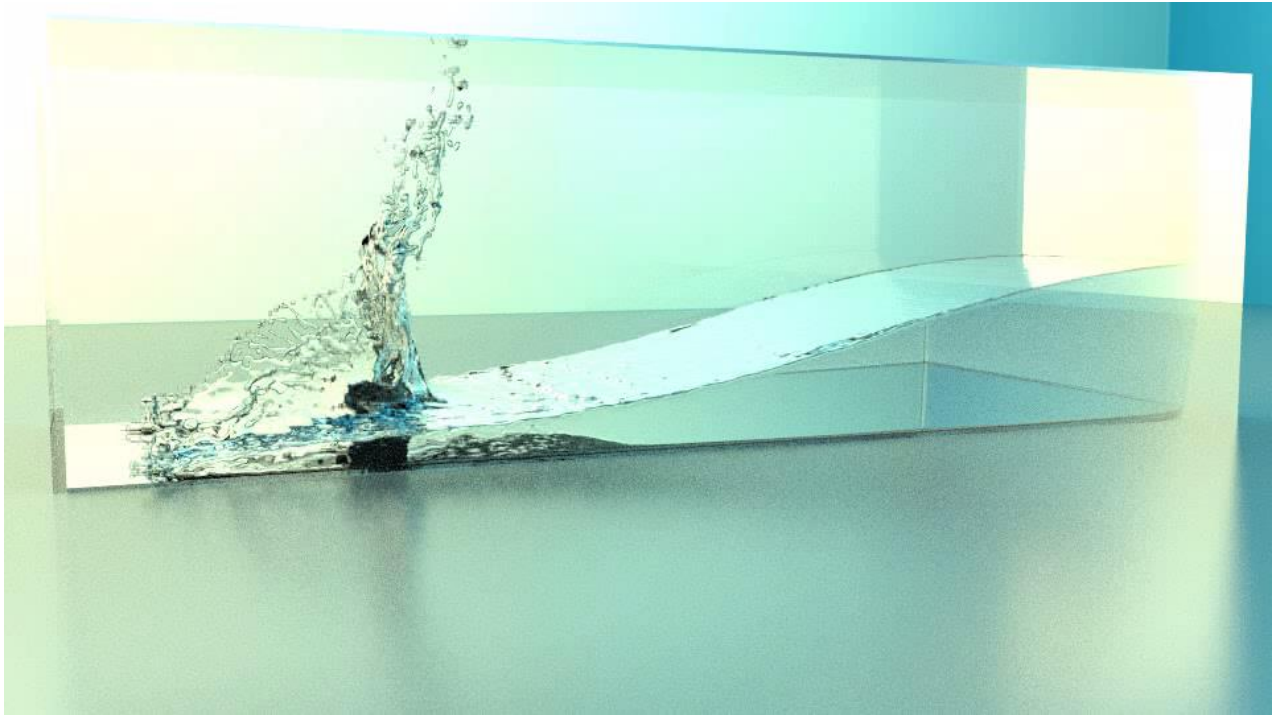
Example 2: Blood flow in an artery

Segment of a human artery with an aneurysm (balloon-like bulge).

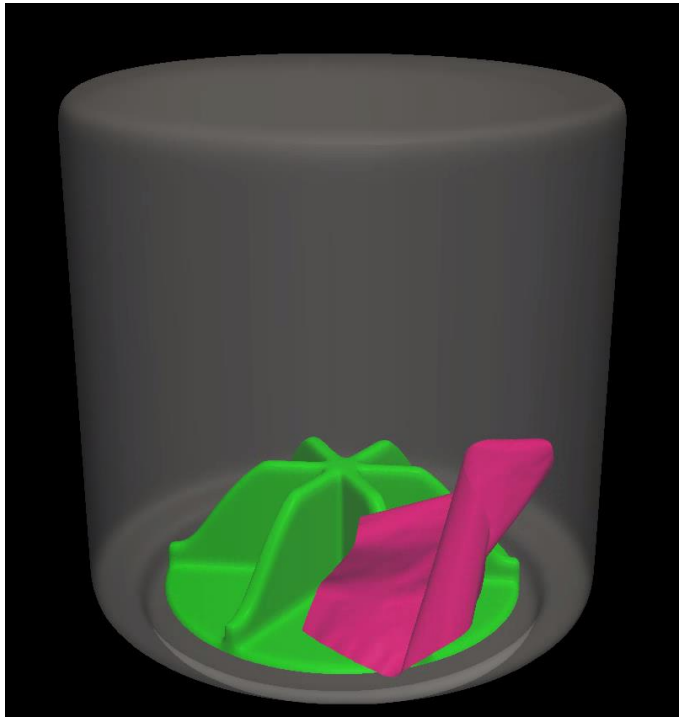
Simulation:

- Fluid dynamics of pulsatile blood flow.
- Embedded red blood cells.
- Blood clotting inside aneurysm, with change of geometry.

Example 3: Collapse of a water column



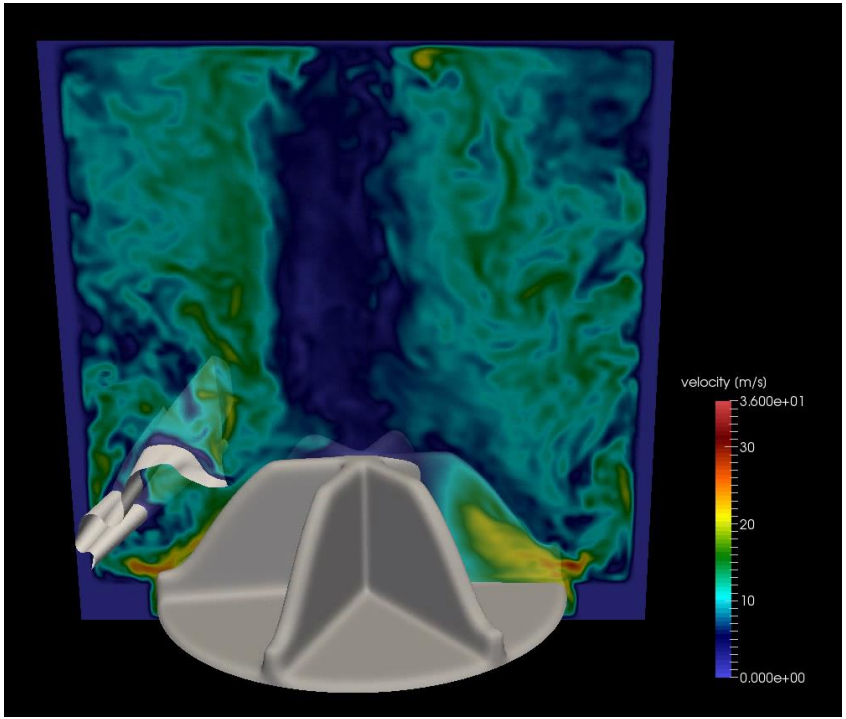
Example 4: washing machine



Simulation:

- Water fluid flow.
- Rotating agitator.
- Embedded flexible tissue, 2-way coupled with the water.

Example 4: washing machine



Washing machine: velocity field.

Credits

I'd like to thank

- **Orestis Malaspinas** (University of Geneva) for the blood flow example.
- **Dimitrios Kontaxakis** (FlowKit Ltd, Lausanne) for the example of a washing machine.
- **Andrea Parmigiani** (University of Geneva) and **Andrea Di Blasio** (FlowKit Ltd, Lausanne) for water column example.

Computational Fluid Dynamics: Equations and challenges

The Navier-Stokes equations

The basis: Navier-Stokes equations for incompressible flow.
They don’t account for

- Fluid compressibility in gases.
- Thermal effects.
- Chemical reactions.
- … many more.

And yet, they are very frequently and successfully used, for both liquids and gases.

The Navier-Stokes equations

$$\partial_t \mathbf{u} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho_0}\nabla p + \nu \Delta \mathbf{u}$$

$$\nabla \cdot \mathbf{u} = 0$$

Flow acceleration

Convective acceleration

Pressure Gradient

Viscous dissipation

Continuity equation:
Fluid incompressibility

$\mathbf{u}(\mathbf{x}, t)$	Velocity
$p(\mathbf{x}, t)$	Pressure
ν	Viscosity

Dimensionless formulation

$$\partial_t \mathbf{u} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho_0}\nabla p + \nu \Delta \mathbf{u}$$

Choose a…

Characteristic velocity: U

Characteristic length: L

$$\vec{u}^* = \frac{\vec{u}}{U} \qquad \partial_t^* = \frac{L}{U}\partial_t$$

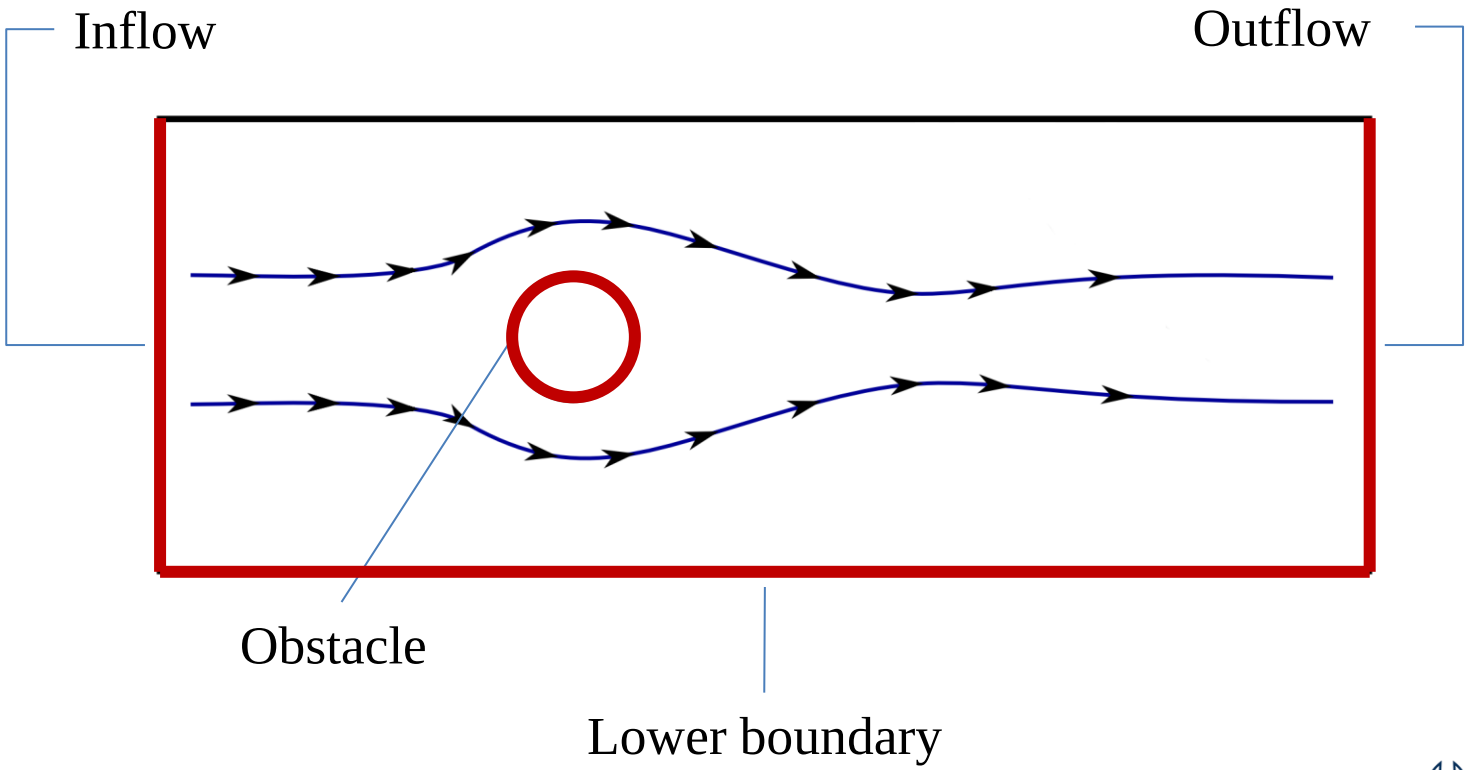
$$p^* = \frac{p}{\rho_0 U^2} \qquad \vec{\nabla}^* = L\vec{\nabla}$$

Reynolds number:

$$Re = \frac{UL}{\nu}$$

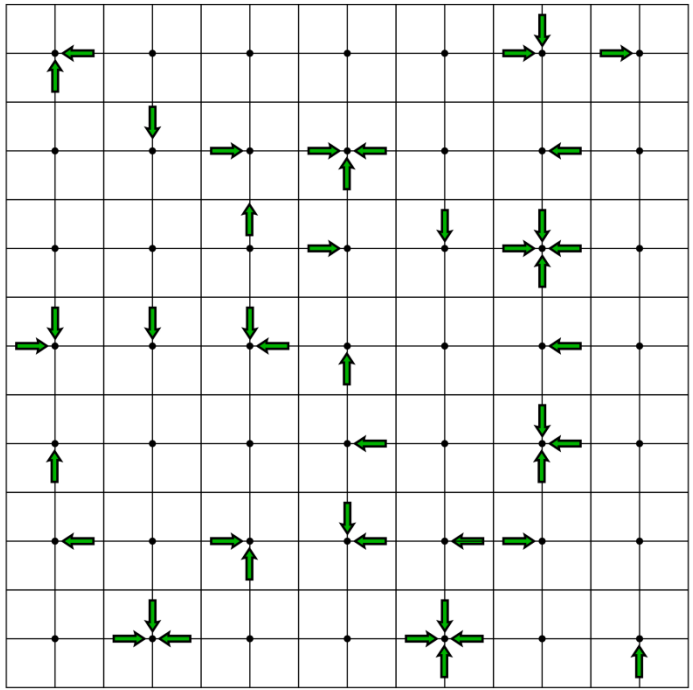
$$\partial_t^* \mathbf{u}^* + (\mathbf{u}^* \cdot \nabla^*)\mathbf{u}^* = -\nabla p^* + \frac{\nu}{UL}\Delta \mathbf{u}^*$$

Boundary Conditions



The predecessor of Lattice Boltzmann: Lattice Gas Cellular Automata

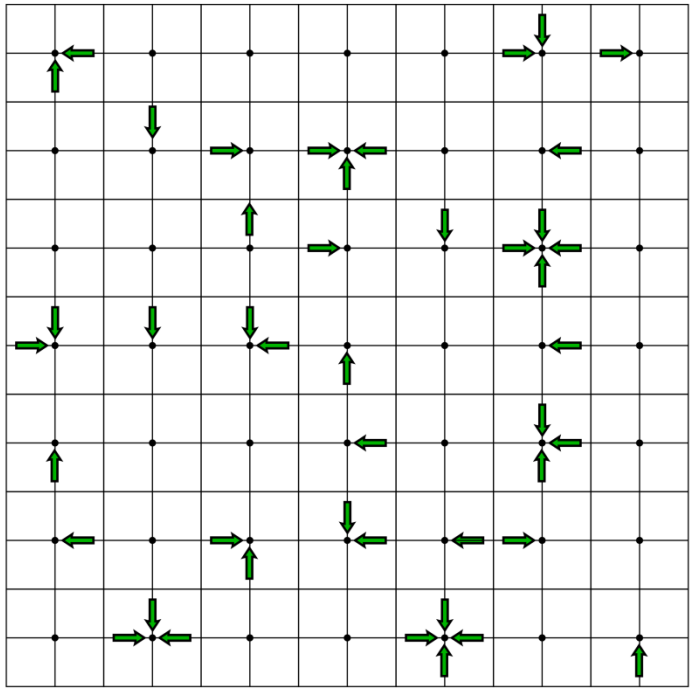
Lattice Gas Automata



A lattice gas cellula automaton is a lattice of cells, in which:

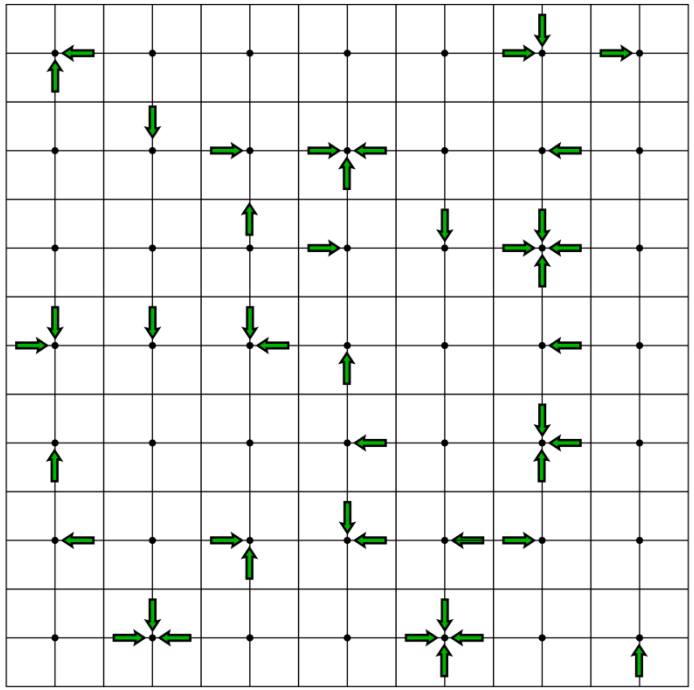
- The state of the system is described by discrete Boolean states.
- The cells are represented by dots.
- The degrees of freedom (the particles) are labeled as $n_i \in \{0,1\}$. Each cell contains z quantities $n_i(\boldsymbol{x}, t), i = 1 \cdots z$. In the example on the left, z=4.

Lattice Gas Automata



- Each cell in this example is populated by at most four cells, represented by a green arrow.
- The arrow indicates the direction in which a particle is moving.
- The constant z is the coordination number: all the neighbors of site $\boldsymbol{x}$ are obtained as $\boldsymbol{x} + \boldsymbol{v}_i$.
- The lattice directions are $\boldsymbol{v}_1 = (1,0)$, $\boldsymbol{v}_2 = (0,1)$, $\boldsymbol{v}_3 = (-1,0)$, and $\boldsymbol{v}_4 = (0,-1)$

Lattice Gas Automata



The evolution rule has two steps:

- **Interaction step**: The quantities n_i «collide» locally and new values $n_i{}'$ are computed, according to a collision operator $\Omega_i(n)$.
- **Streaming step**: The quantity $n_i{}'(\boldsymbol{x})$is sent to the neighboring site along the lattice direction $\boldsymbol{v}_i$.

Comment

- This mesoscopic model describes the dynamics of a **gas**, not a liquid.

Lattice Gas Automata

Exclusion principle

- The fact that we choose the n_i as Boolean variables (1 or 0) implies an **exclusion principle**. There is no more than one particle per cell traveling in a given direction.
- The quanity n_i is also called the occupation number. It indicates the presence ($n_i = 1$) or absence ($n_i = 0$) of a physical particle entering a site along the direction $\boldsymbol{v}_i$.
- There is a finite number (2^z) of possible input and output configurations. The collision operator can be pre-computed for all input configurations, and represented as a lookup table with 2^z entries.

Lattice Gas Automata

Microdynamics

- Collision: $n_i^{\text{out}}(\boldsymbol{x}, t) = n_i^{\text{in}}(\boldsymbol{x}, t) + \Omega_i(\boldsymbol{x}, t)$

- Streaming: $n_i^{\text{in}}(\boldsymbol{x}, t) = n_i^{\text{out}}(\boldsymbol{x} - \boldsymbol{v}_i \delta t, t - \delta t)$

- The constant δt carries units of time and $\boldsymbol{v}_i$ units of velocity.
- During the streaming step, the particle n_i travels in direction of the velocity $\boldsymbol{v}_i$, reaches the cell at position $\boldsymbol{x} + \boldsymbol{v}_i$, and keeps its direction $\boldsymbol{v}_i$.

Lattice Gas Automata

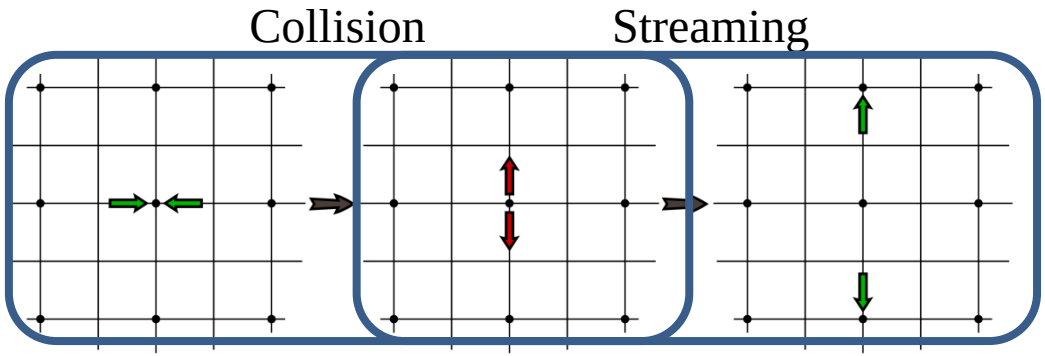
Some simple lattice-gas fluid models

- **HPP**: Hardy, Pomeau, de Pazzis, 1971: kinetic theory of point particles on a rectangular lattice.
- **FHP**: Frisch, Hasslacher and Pomeau, 1986: first LGA reproducing (almost) correct hydrodynamic behaviour (i.e. the Navier-Stokes equations.

Lattice Gas Automata

Example: collision and streaming in the HPP model

- During collision, the occupation numbers of the post-collision states (red) are constructed from the pre-collision states. We don't provide the full HPP collision rule here, but just show one example: head-on collision.
- Streaming maps post-collision states(red) to pre-collision states (green) on neighboring nodes.



Our color scheme:

Green: pre-collision

Red: post-collision

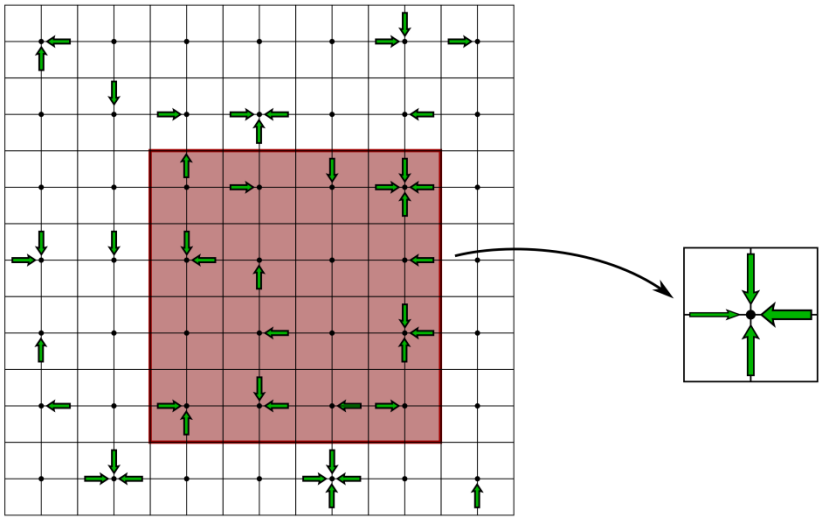
From Lattice Gas to Lattice Boltzmann

Why aren’t we doing Navier-Stokes with Lattice Gas Automata?

Issues with Lattice Gas Automata:

- Lattice gas is lower-level («mesoscopic») than macroscopic Navier-Stokes models («macroscopic»), and represents more details of the physics. But most often, these details are not needed.
- Molecular noise: you need many particles to reach reasonable statistical averages.
- Computer hardware: modern computers are efficient with floating-point calculations. The discrete lattice gas models loose their advantage.

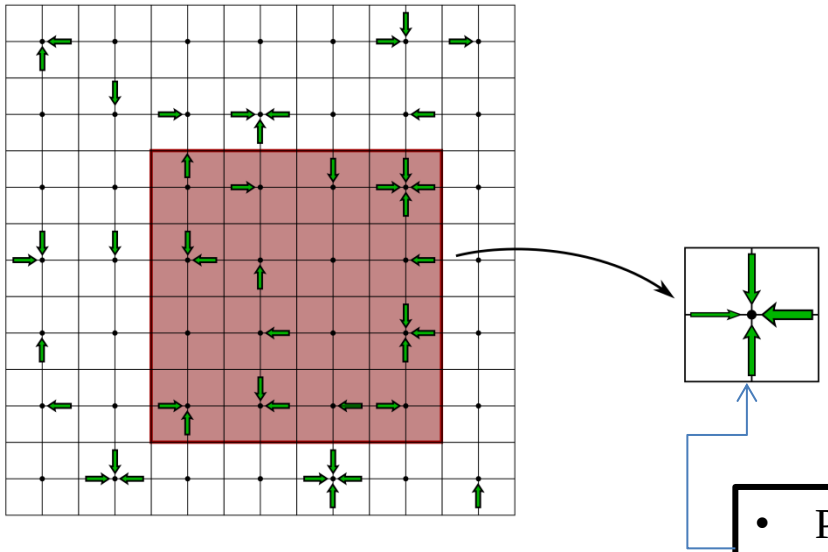
From discrete to continuum variables



In a lattice gas automaton:

- Run the simulation with discrete variables.
- At the end, extract macroscopic variables by taking averages.

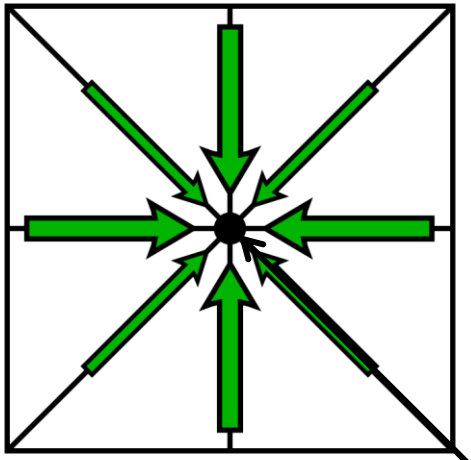
Lattice Boltzmann Method: Idea



- Run simulation with continuum variables right away, to save space.
- Continuum model is derived from discrete method by means of statistics.

- Particle density: real-valued.
- Arrow thickness stands for particle density.

Lattice Boltzmann Method: details

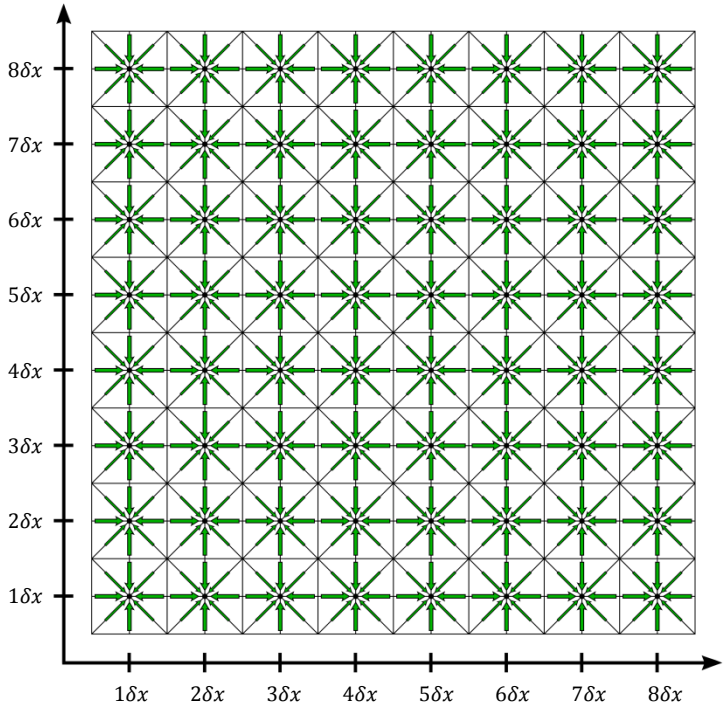


Lattice Boltzmann 2D model D2Q9:

- Four directions aren't quite enough.
- Nine directions: eight connections to nearest neighbors + «rest population».
- Variables representing particle densities are called «populations».

«Rest population» particles stay on a cell, don't travel to neighbors during streaming.

Lattice Boltzmann Variables



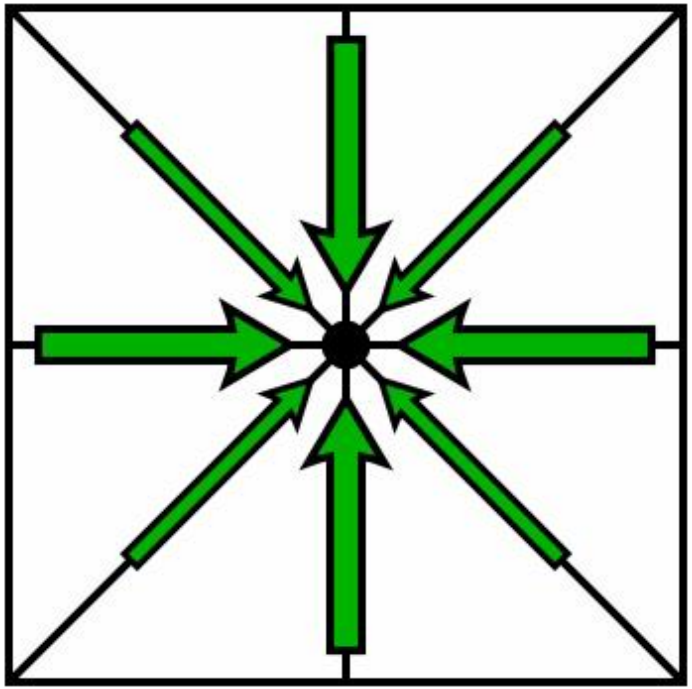
Space Discretization:

- Every cell represents flow variables at a point in space.
- Equal distance δx between cells in x- and y-direction.
- To represent space with 8x8 points, we need a total of 8x8x9 = 576 floating point variables.

Python code to allocate memory:

```
f = zeros(9, 8, 8)
```

Collision step



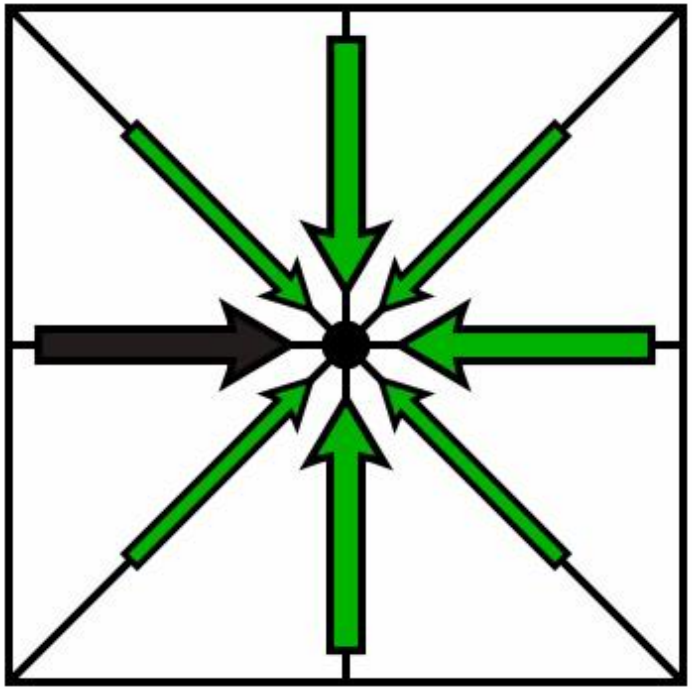
- Like in lattice gas: collision maps pre-collision populations to post-collision populations.
- All populations are modified during collision. Changes are not visible on video, because they are small perturbations.

Our color scheme:

Green: pre-collision

Red: post-collision

Collision step



Model:

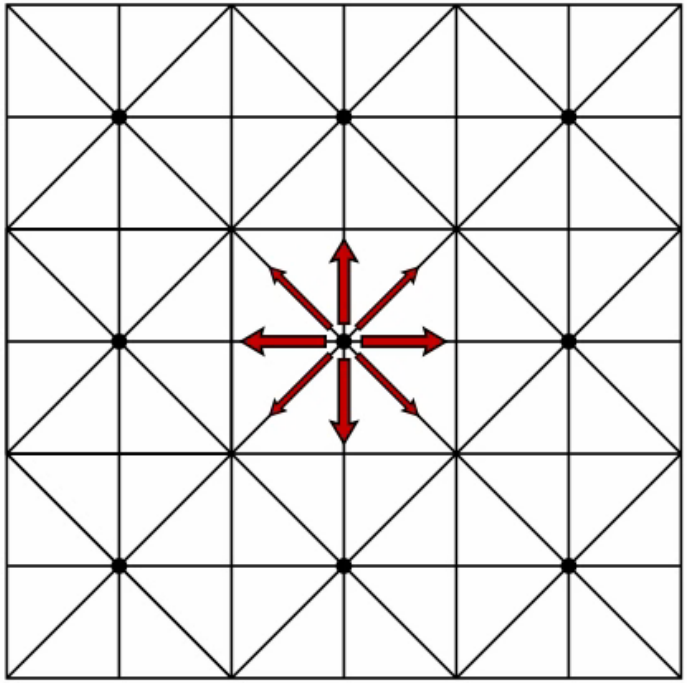
- Collision is instantaneous.
- Collision is localized at cell coordinates.

Our color scheme:

Green: pre-collision

Red: post-collision

Streaming step



Model:

- Streaming takes the system from time t to time $t + \delta t$.
- Streaming includes nearest-neighbor access.

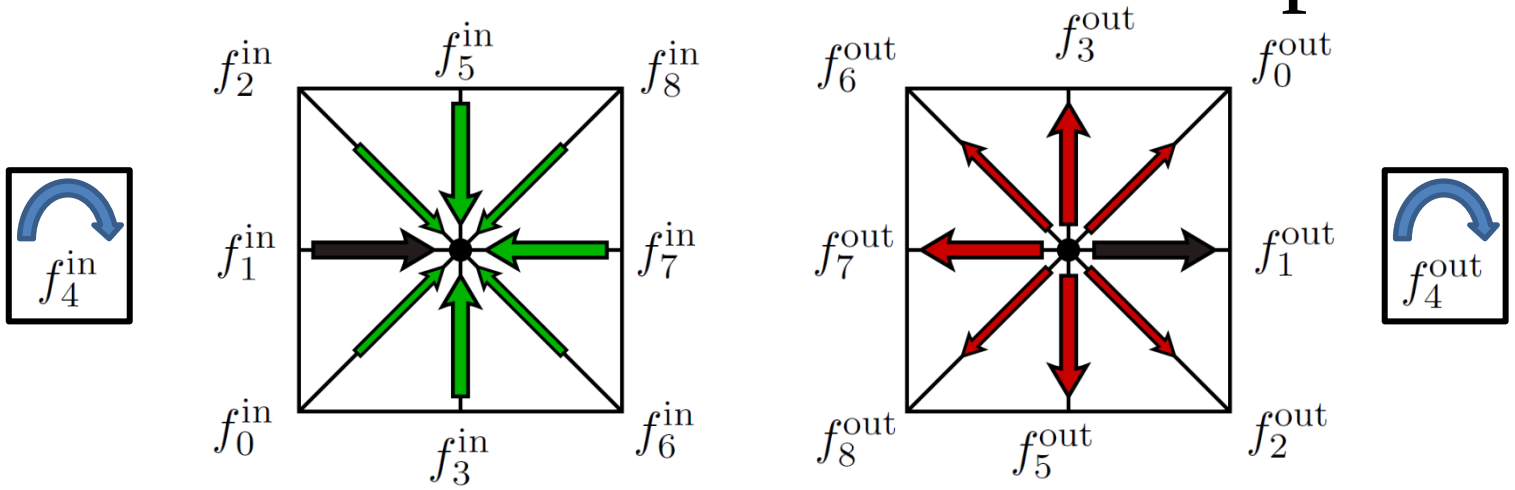
Our color scheme:

Green: pre-collision

Red: post-collision

Macroscopic Variables

Back to the collision step



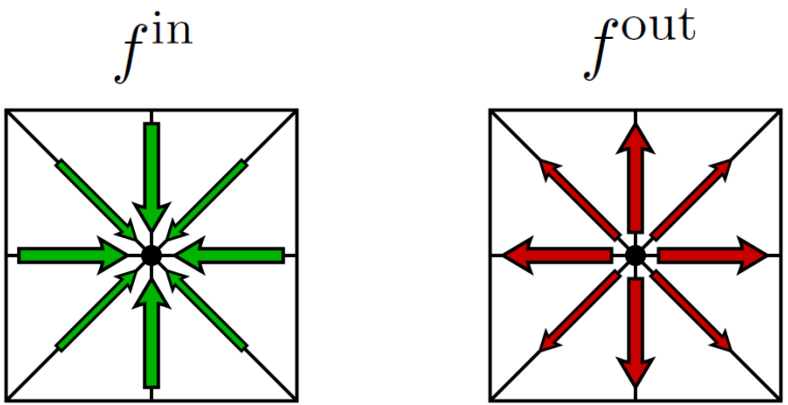
Pre-collision: $f_i^{\text{in}}(\mathbf{x}, t)$
«Incoming populations»

Post-collision: $f_i^{\text{out}}(\mathbf{x}, t)$
«Outgoing populations»

The index i runs from 0 to 8.

Pre- and post-collision: variables

In our Python code, we save incoming and outgoing populations in separate matrices.



Python code:

```
# assign some size
nx, ny = ...
fin = zeros(9, nx, ny)
fout = zeros(9, nx, ny)
```

Density

Particle density is defined the same as in a lattice gas automaton:

$$\rho(\boldsymbol{x}, t) = \sum_{i=0}^{8} f_i^{\text{in}}(\boldsymbol{x}, t)$$

Each cell has its own density, which is the sum of the nine populations.

Density

$$\rho(\boldsymbol{x}, t) = \sum_{i=0}^{8} f_i^{\text{in}}(\boldsymbol{x}, t)$$

Python code:

```
rho = zeros((nx, ny))
for ix in range(nx):
    for iy in range(ny):
        rho[ix, iy] = 0
        for i in range(9):
            rho[ix,iy] += fin[i, ix, iy]
```

Simpler and faster, with NumPy array-based syntax:

```
rho = sum(fin, axis=0)
```

Pressure

Pressure is proportional to density, according to the ideal gas law, at constant temperature:

$$p = c_s^2 \rho$$

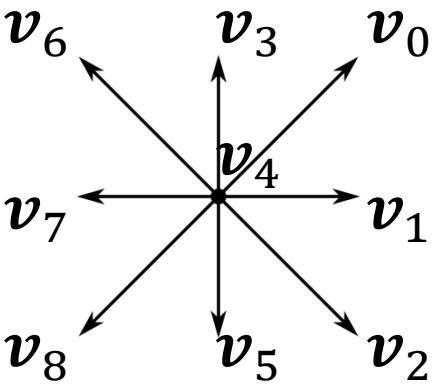
The speed of sound c_s is, in our D2Q9 model, a lattice constant:

$$c_s^2 = \frac{1}{3}\frac{\delta x^2}{\delta t^2}$$

Units: Say that δx has units of meters $[m]$ and δt has units of seconds$[s]$. Then pressure has units of $\frac{m^2}{s^2}$. Multiply this by the physical density of our fluid $\left(\text{e.g. } 1\,\frac{kg}{m^3}\right)$ to obtain units of Pascal $\left(\frac{kg}{ms^2}\right)$.

Velocity

Let's introduce a set of 9 vectors:



$\boldsymbol{v}_0 = (1,1),$
$\boldsymbol{v}_1 = (1,0),$
$\boldsymbol{v}_2 = (1,-1),$
...
$\boldsymbol{v}_4 = (0,0),$
...

They are called lattice velocities: If there's a cell at position $\boldsymbol{x}$, there's also one at $\boldsymbol{x} + \boldsymbol{v}_i \delta t$.

Again, velocity is defined just the same as in a lattice gas automaton:

$$\boldsymbol{u}(\boldsymbol{x},t) = \frac{1}{\rho(\boldsymbol{x},t)} \frac{\delta x}{\delta t} \sum_{i=0}^{8} \boldsymbol{v}_i f_i^{\text{in}}(\boldsymbol{x},t)$$

Velocity

```
v = array([ [ 1,  1], [ 1,  0], [ 1, -1], [ 0,  1], [ 0,  0], [ 0, -1], [-1,  1], [-1,  0], [-1, -1] ])
```

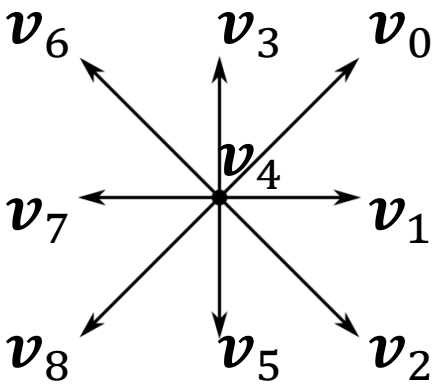
Traditional Python code:

```
u = zeros((2, nx, ny))
for ix in range(nx):
    for iy in range(ny):
        u[0, ix, iy] = 0
        u[1, ix, iy] = 0
        for i in range(9):
            u[0, ix,iy] += v[i,0] * fin[i, ix, iy]
            u[1, ix,iy] += v[i,1] * fin[i, ix, iy]
        u[0, ix, iy] /= rho[ix, iy]
        u[1, ix, iy] /= rho[ix, iy]
```

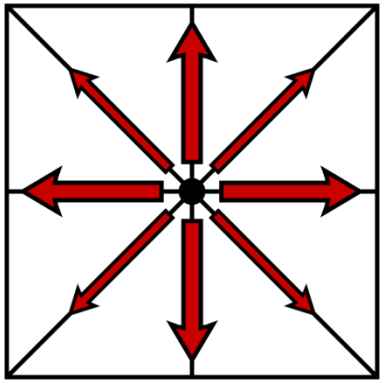
Simpler and faster, with NumPy array-based syntax:

```
u = zeros((2, nx, ny))
for i in range(9):
    u[0,:,:] += v[i,0] * fin[i,:,:]
    u[1,:,:] += v[i,1] * fin[i,:,:]
u /= rho
```

Don’t be confused



- These are 9 2D vectors.
- They have constant values.

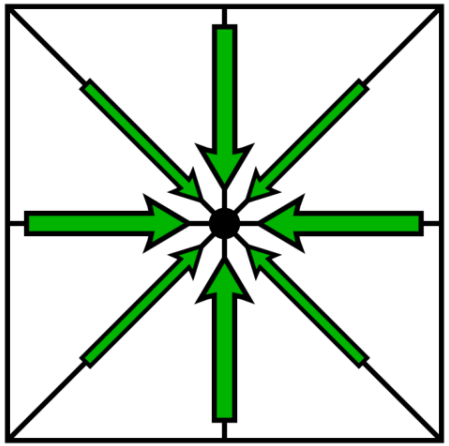


- These are 9 scalar values.
- Their value is time-dependent.

Collision step: the BGK model

What's the value of the populations?

- So far, we have interpreted populations as particle averages of a lattice gas automaton.
- Alternative interpretation: populations are probability densities of particles in a real gas.
- In *kinetic theory*, gas densities are described by a single *probability density function* $f(\boldsymbol{x}, \boldsymbol{v}, t)$.
- The motion of f is described by the *Boltzmann equation*.

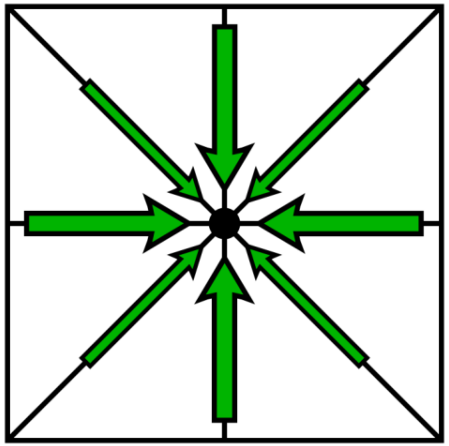


What's the value of the populations?

- When a gas is macroscopically at rest, or has a constant velocity $\boldsymbol{u}$, its molecules are still moving.
- The distribution of their velocities v at every point x in space is given by a *Maxwell-Boltzmann distribution*:

$$f(\boldsymbol{x}, \boldsymbol{v}, t) \sim e^{|\boldsymbol{v}-\boldsymbol{u}|^2}.$$

- Even when it moves, a gas remains close to its equilibrium state. We consider its motion as a *perturbation around the equilibrium*.



Collision: BGK model

Inspired by kinetic theory, we formulate the collision term in lattice Boltzmann as a relaxation to local equilibrium:

$$f_i^{\text{out}} - f_i^{\text{in}} = -\omega \left(f_i^{\text{in}} - E(i, \rho, \vec{u}) \right)$$

- E is the local equilibrium. It depends on the macroscopic variables and has a different value for every direction i.
- The parameter ω is the frequency of relaxation.
- This collision model is called the *BGK model*.

Collision frequency and viscosity

- If ω is small, the fluid converges only slowly to its equilibrium: it is highly viscous.
- More generally, on can show that the fluid viscosity depends inversely on the relaxation parameter ω:

$$\nu = \delta t \, c_s^2 \left(\frac{1}{\omega} - \frac{1}{2} \right)$$

Collision: BGK model

The equilibrium is obtained as a truncated series of the Maxwell-Boltzmann distribution:

$$E(i, \rho, \boldsymbol{u}) = \rho\, t_i \left(1 + \frac{\boldsymbol{v_i} \cdot \boldsymbol{u}}{c_s^2} + \frac{1}{2\, c_s^4} (\boldsymbol{v_i} \cdot \boldsymbol{u})^2 - \frac{1}{2\, c_s^2} |\boldsymbol{u}|^2 \right)$$

- The constants t_i compensate for the different lengths of velocities v_i.
- t_i is 1/9 for orthogonal directions,1/36 for diagonal directions, and 4/9 for the rest velocity:

```
t = array([ 1/36, 1/9, 1/36, 1/9, 4/9, 1/9, 1/36, 1/9, 1/36])
```

Collision: Python codes

Equilibrium:

$$E(i,\rho,\boldsymbol{u}) = \rho\, t_i \left(1 + \frac{\boldsymbol{v_i}\cdot\boldsymbol{u}}{c_s^2} + \frac{1}{2\,c_s^4}\left(\boldsymbol{v_i}\cdot\boldsymbol{u}\right)^2 - \frac{1}{2\,c_s^2}\left|\boldsymbol{u}\right|^2\right)$$

We write the code in «lattice units», i.e. with $\delta x = \delta t = 1$.

```
def equilibrium(rho, u):
    usqr = 3/2 * (u[0]**2 + u[1]**2)
    eq = zeros((9, nx, ny))
    for i in range(9):
        vu = 3 * (v[i,0]*u[0,:,:] + v[i,1]*u[1,:,:])
        eq[i,:,:] = rho*t[i] * (1 + vu + 0.5*vu**2 - usqr)
    return eq
```

Collision: Python codes

Collision:

$$f_i^{\text{out}} - f_i^{\text{in}} = -\omega \left(f_i^{\text{in}} - E(i, \rho, \vec{u}) \right)$$

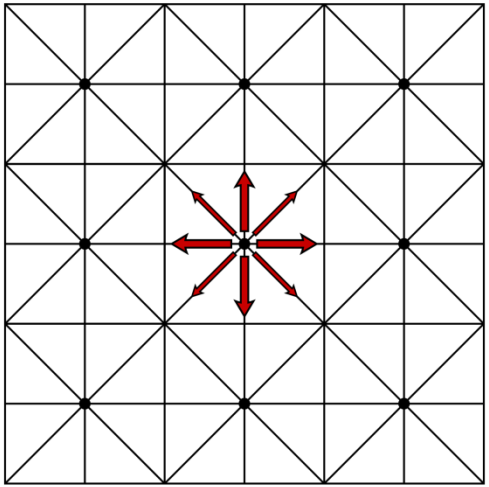
```
fout = fin - omega * (fin - eq)
```

The streaming step

Streaming step

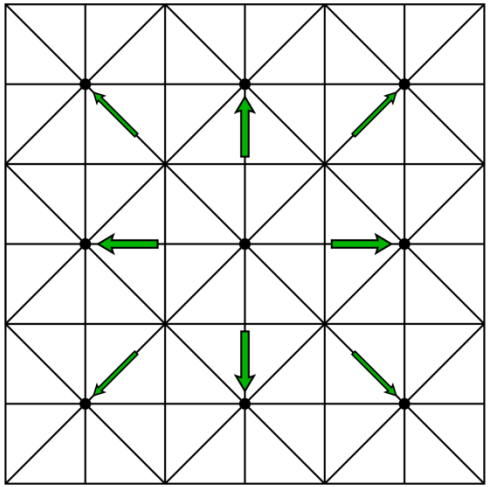
Streaming takes the system to the next time iteration, $t + \delta t$

Time t



Post-collision f^{out}

Time $t + \delta t$



Pre-collision f^{in}

Streaming step: formula

$$f_i^{\text{in}}(\boldsymbol{x}, t) = f_i^{\text{out}}(\boldsymbol{x} - \boldsymbol{v}_i \delta t, t - \delta t)$$

The streaming step

- Maps outgoing populations (red) onto incoming populations (green).
- Copies populations to neighboring cells, according to the direction implied by their index *i*.
- Takes the system from time step t to time step $t + \delta t$.

Streaming step: code

Conventional Python code:

```
for ix in range(nx):
    for iy in range(ny):
        for i in range(9):
            next_x = ix + v[i,0]
            if next_x < 0: next_x = nx-1
            if next_x >= nx: next_x = 0

            next_y = iy + v[i,1]
            if next_x < 0: next_x = nx-1
            if next_x >= nx: next_x = 0
            fin[i, next_x,next_y] = fout[i, ix, iy]
```

- On boundaries, we apply periodicity.
- If a population leaves the domain, it enters the domain again on the other side.

Streaming step: code

Conventional Python code:

```
for ix in range(nx):
    for iy in range(ny):
        for i in range(9):
            next_x = ix + v[i,0]
            if next_x < 0: next_x = nx-1
            if next_x >= nx: next_x = 0

            next_y = iy + v[i,1]
            if next_x < 0: next_x = nx-1
            if next_x >= nx: next_x = 0
            fin[i, next_x,next_y] = fout[i, ix, iy]
```

NumPy Array-based code:

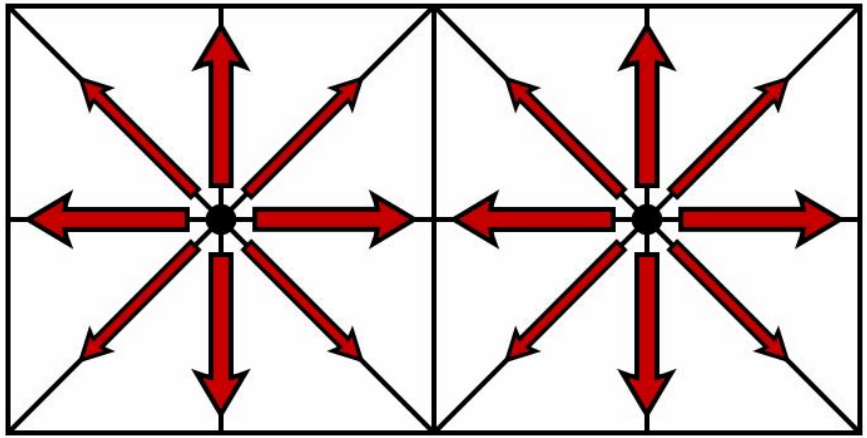
```
for i in range(9):
    fin[i,:,:] = roll(
                    roll(fout[i,:,:], v[i,0], axis=0),
                    v[i,1], axis=1 )
```

Do we really need double the memory?

- So far we allocated two sets of populations f^{in} and f^{out}.
- Could we spare one, and store both pre-collision and post-collision in the same variable f?
- Answer: yes, but it needs some care.
- I will show how to do it. But for simplicity, we keep our code with two sets of populations.

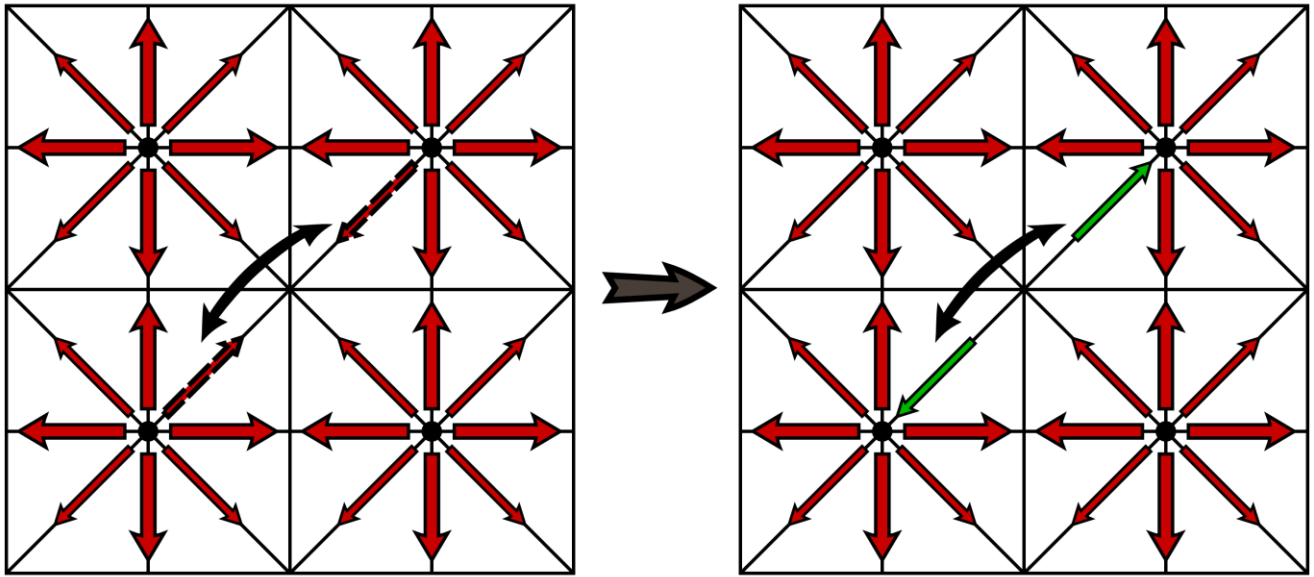
Single-population implementation

- Collision step, no problem: it is local. Simply overwrite f^{in} with f^{out}.
- Streaming step, more tricky. As you stream populations to a neighbor, you risk overwriting one that is still needed.
- Solution: *stream to the neighbor* and *from the neighbor* at the same time, exchanging populations instead of overwriting them.



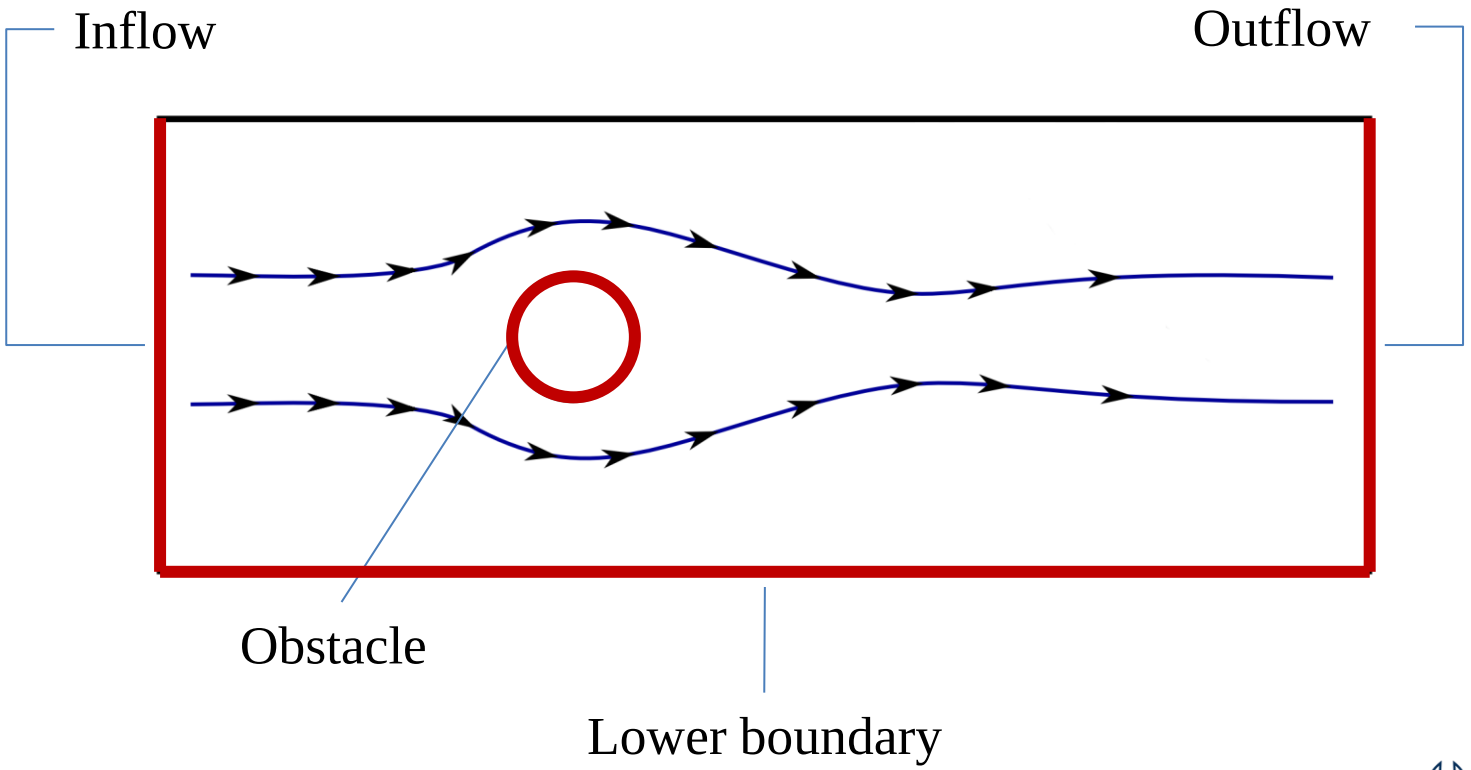
Single-population implementation

This strategy works along all directions:



Boundary conditions

Boundary conditions



Boundary conditions

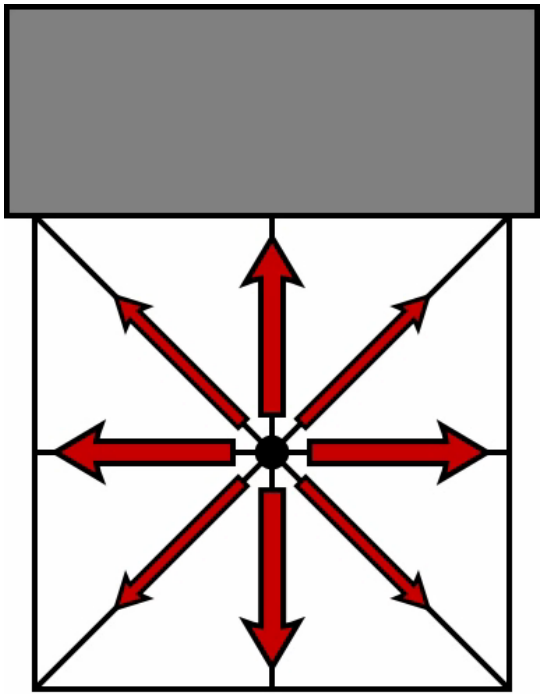
- We need different types of boundary conditions:
  - Inflow: imposes the velocity profile.
  - Outflow, and lower and upper boundary: pretend that the domain is larger.
  - Physical obstacle.
- There exists already a boundary condition in our code: periodicity.
  - Behaves as if our simulation was repeated an infinitely many times.
  - We will use periodicity for lower/upper boundary.

Obstacle: no-slip wall

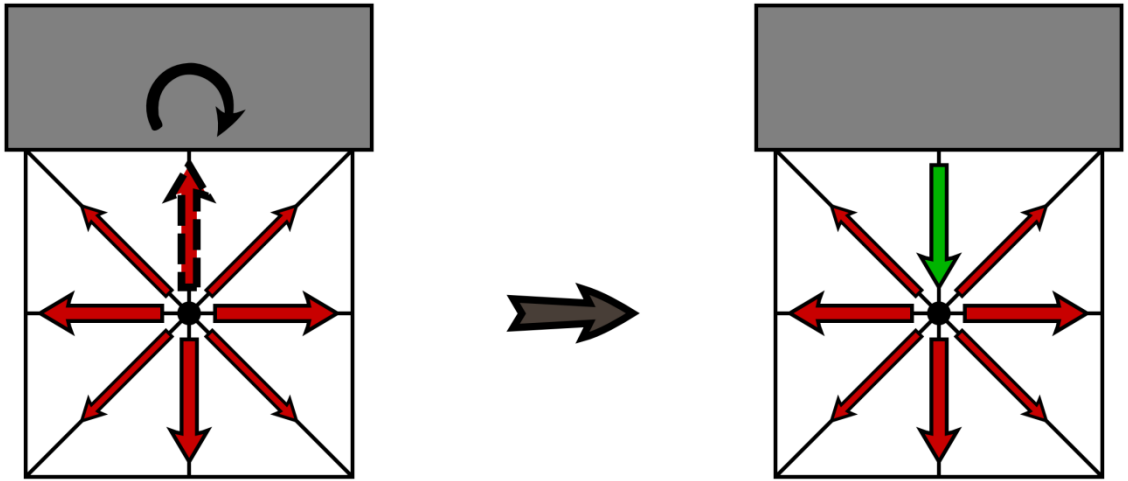
- Most physical obstacles act like no-slip walls.
- Macroscopically: the velocity is zero along the wall.
- At the molecular level: the obstacle has a surface roughness, and molecules «adhere» to the surface.
- Not all obstacles have a no-slip condition. Example: an air bubble in water is smooth, even at molecular level.

No-slip walls: bounce-back condition

- The bounce-back condition is a way to implement a no-slip wall in lattice Boltzmann.
- During streaming, a population which leaves the fluid and hits an obstacle «bounces back».
- Its value is unchanged, and it is copied to the population with reverse direction.

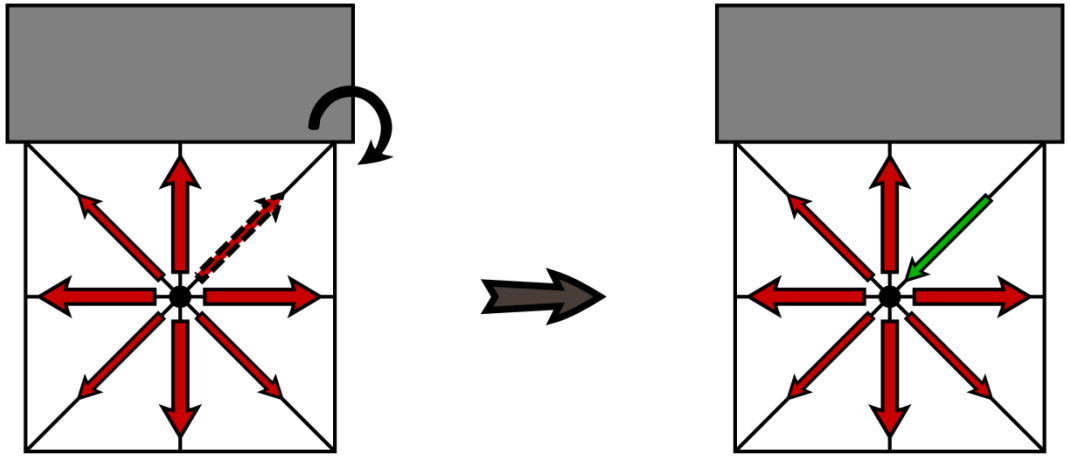


No-slip walls: bounce-back condition



This condition mimicks the physical process that occurs when molecules hit a wall.

No-slip walls: bounce-back condition



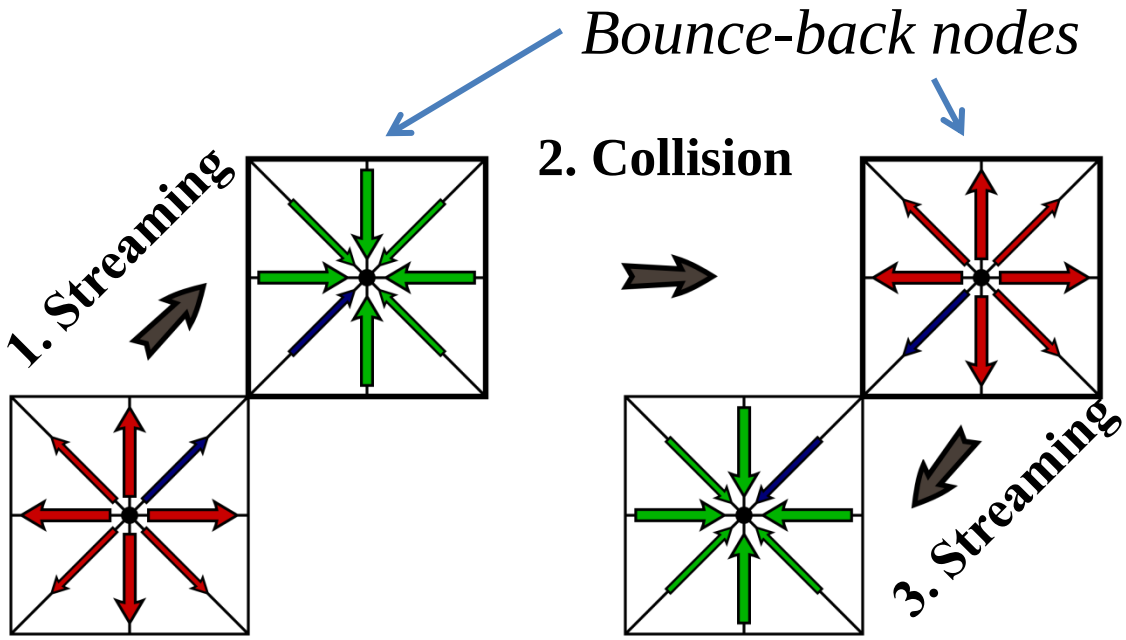
- Attention: populations on diagonal directions are not reflected like on a mirror. They also bounce-back to where they came from.
- This reflects the fact that at a molecular level, the obstacle is rough, and not smooth like a mirror.

No-slip walls: bounce-back condition

- Bounce-back is simple: track all populations that hit an obstacle, and send them back.
- Yet, this involves quite some book-keeping. Which nodes are affected? Which directions?
- Simpler solution: let the populations travel into the obstacle. Then, we revert them inside the obstacle, and send them back.
- No bookkeeping of directions. All cells inside an obstacle are bounce-back cells: instead of colliding, they revert all the directions.

The bounce-back cycle

During collision, a bounce-back node replaces every population by the one in the opposite direction.



The bounce-back cycle

Instead of colliding, bounce-back nodes implement the following relation:

$$f_i^{\text{in}}(\boldsymbol{x}, t+1) = f_j^{\text{out}}(\boldsymbol{x}, t)$$

where direction *i* is opposite to direction *j*:

$$v_i = -v_j$$

Bounce-back: code

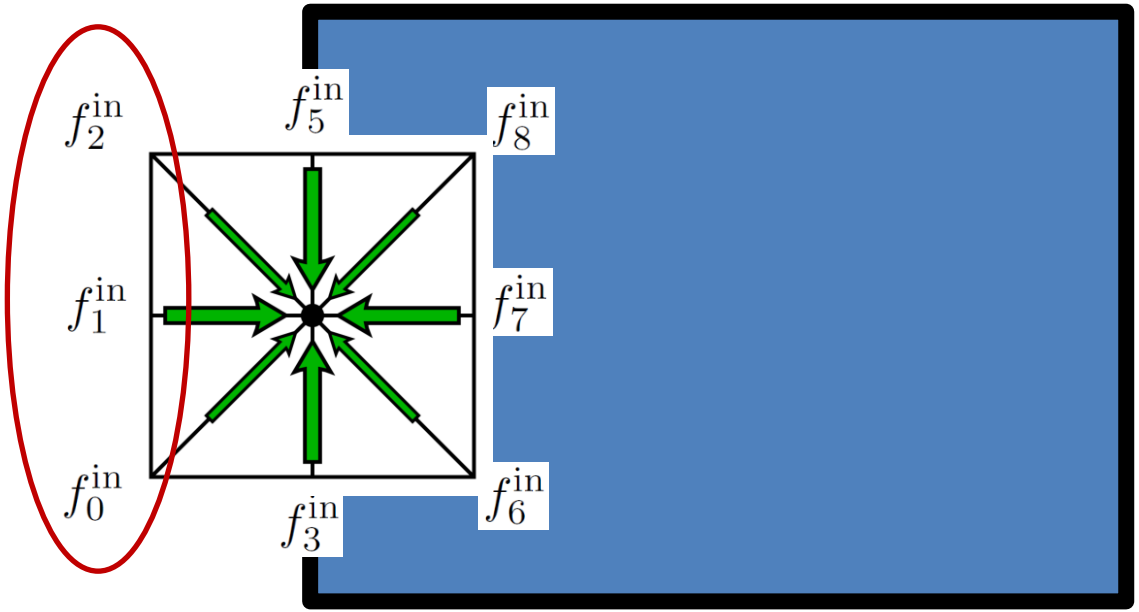
NumPy array-based code:

```
for i in range(9):
    fout[i, obstacle] = fin[8-i, obstacle]
```

- We have on purpose listed the lattice velocities in such a way that the direction opposite to *i* is given by 8-*i*.

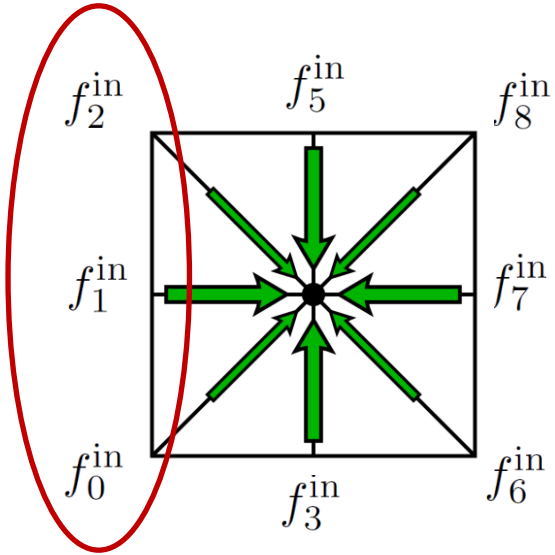
Inflow condition

After streaming, the populations f_0, f_1, and f_2 are unknown on all cells along the left boundary.



Inflow condition

- We want to impose a velocity $\boldsymbol{u}$ on inflow boundary cells. So, $\boldsymbol{u}$ is known. But what about ρ? And what about f_0, f_1, and f_2?
- We must extract this missing information using the known populations, and using $\boldsymbol{u}$.



Inflow condition: density

Remember:

$$\rho(\boldsymbol{x},t) = \sum_{i=0}^{8} f_i^{\text{in}}(\boldsymbol{x},t)$$

$$\boldsymbol{u}(\boldsymbol{x},t) = \frac{1}{\rho(\boldsymbol{x},t)} \frac{\delta x}{\delta t} \sum_{i=0}^{8} \boldsymbol{v}_i f_i^{\text{in}}(\boldsymbol{x},t)$$

Let's define:

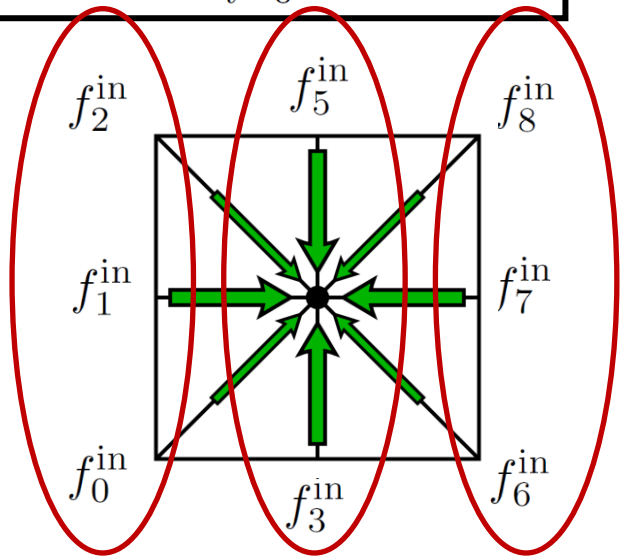
- $\rho_1 = f_0 + f_1 + f_2$ (unknown)
- $\rho_2 = f_3 + f_4 + f_5$ (known)
- $\rho_3 = f_6 + f_7 + f_8$ (known)

Then:

- $\rho = \rho_1 + \rho_2 + \rho_3$
- $\rho u_x = \rho_1 - \rho_3$

And hence:

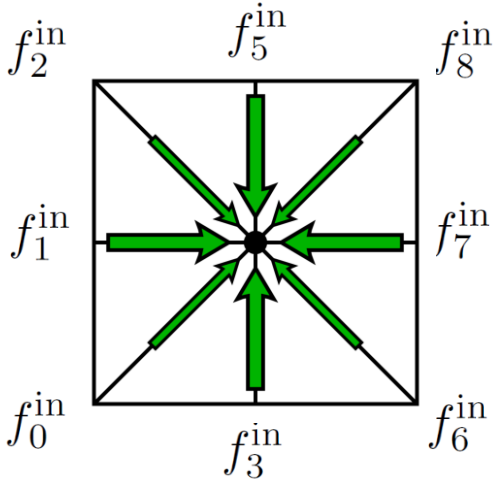
$$\rho = \frac{\rho_2 + 2\rho_3}{1 - u_x}$$



Inflow condition: unknown populations

- Remember: populations are always close to their equilibrium.
- We first initialize the unknown populations to their equilibrium value.
- Then, we check how much the opposite population deviates from equilibrium, at add this value as a correction.

$$f_0^{\text{in}} = E(0, \rho, \boldsymbol{u}) + (f_8^{\text{in}} - E(8, \rho, \boldsymbol{u}))$$
$$f_1^{\text{in}} = E(1, \rho, \boldsymbol{u}) + (f_7^{\text{in}} - E(7, \rho, \boldsymbol{u}))$$
$$f_2^{\text{in}} = E(2, \rho, \boldsymbol{u}) + (f_6^{\text{in}} - E(6, \rho, \boldsymbol{u}))$$



Inflow condition: code

- Define the indices of the three columns:

```
col1 = array([0, 1, 2])
col2 = array([3, 4, 5])
col3 = array([6, 7, 8])
```

- Calculate the density:

```
rho[0,:] = 1/(1-u[0,0,:]) * (     sum(fin[col2,0,:], axis=0) +
                                2*sum(fin[col3,0,:], axis=0) )
```

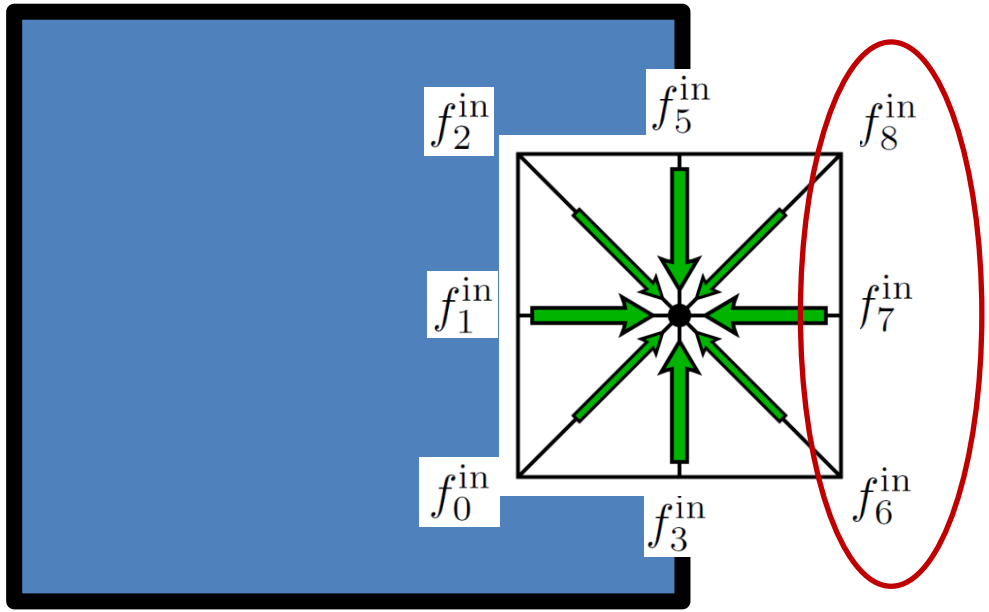
- Calculate the populations:

```
fin[[0,1,2],0,:] = feq[[0,1,2],0,:] + fin[[8,7,6],0,:] - feq[[8,7,6],0,:]
```

Outflow condition

- The outflow boundary must behave as if the domain didn't end.
- For the unknown populations, f_6, f_7, and f_8, we simply copy the value from the neighboring cell, right behind.

```
fin[col3,-1,:] = fin[col3,-2,:]
```



Application: flow around an obstacle

Our code: all pieces put together

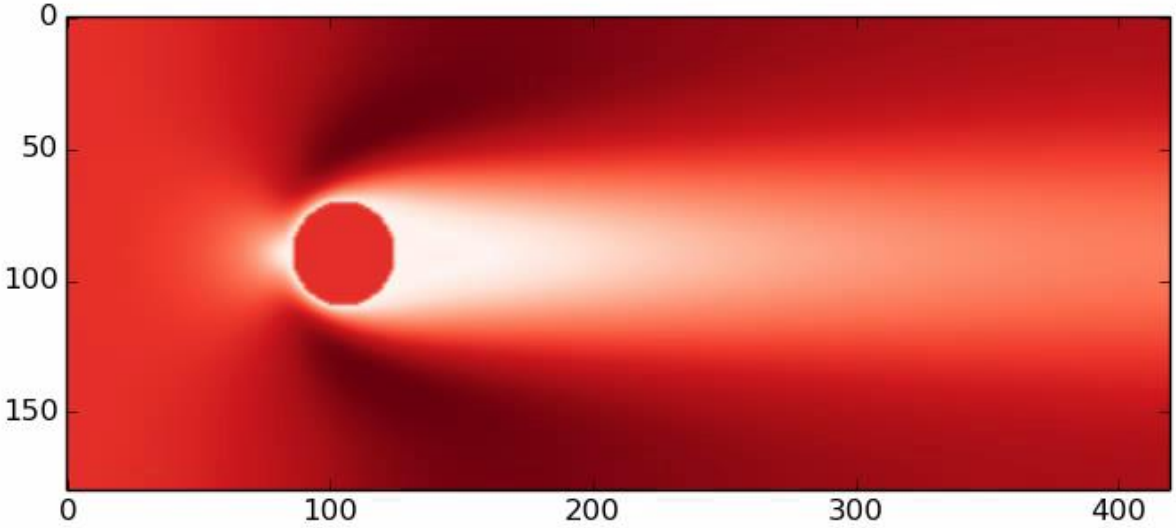
```
#!/usr/bin/python3
# Copyright (C) 2015 Universite de Geneve, Switzerland
# E-mail contact: jonas.latt@unige.ch
#
# 2D flow around a cylinder
#

from numpy import *
import matplotlib.pyplot as plt
from matplotlib import cm

###### Flow definition #################################################################
maxIter = 200000  # Total number of time iterations.
Re = 10.0         # Reynolds number.
nx, ny = 420, 180 # Numer of lattice nodes.
ly = ny-1         # Height of the domain in lattice units.
cx, cy, r = nx//4, ny//2, ny//9 # Coordinates of the cylinder.
uLB     = 0.04                  # Velocity in lattice units.
nulb    = uLB*r/Re;             # Viscoscity in lattice units.
omega = 1 / (3*nulb+0.5);    # Relaxation parameter.

###### Lattice Constants ###############################################################
v = array([ [ 1,  1], [ 1,  0], [ 1, -1], [ 0,  1], [ 0,  0],
            [ 0, -1], [-1,  1], [-1,  0], [-1, -1] ])
t = array([ 1/36, 1/9, 1/36, 1/9, 4/9, 1/9, 1/36, 1/9, 1/36])

col1 = array([0, 1, 2])
col2 = array([3, 4, 5])
col3 = array([6, 7, 8])

###### Function Definitions ############################################################
def macroscopic(fin):
    rho = sum(fin, axis=0)
    u = zeros((2, nx, ny))
    for i in range(9):
        u[0,:,:] += v[i,0] * fin[i,:,:]
        u[1,:,:] += v[i,1] * fin[i,:,:]
    u /= rho
    return rho, u

def equilibrium(rho, u):              # Equilibrium distribution function.
    usqr = 3/2 * (u[0]**2 + u[1]**2)
    feq = zeros((9,nx,ny))
    for i in range(9):
        cu = 3 * (v[i,0]*u[0,:,:] + v[i,1]*u[1,:,:])
        feq[i,:,:] = rho*t[i] * (1 + cu + 0.5*cu**2 - usqr)
    return feq
```

```
###### Setup: cylindrical obstacle and velocity inlet with perturbation ########
# Creation of a mask with 1/0 values, defining the shape of the obstacle.
def obstacle_fun(x, y):
    return (x-cx)**2+(y-cy)**2<r**2

obstacle = fromfunction(obstacle_fun, (nx,ny))

# Initial velocity profile: almost zero, with a slight perturbation to trigger
# the instability.
def inivel(d, x, y):
    return (1-d) * uLB * (1 + 1e-4*sin(y/ly*2*pi))

vel = fromfunction(inivel, (2,nx,ny))

# Initialization of the populations at equilibrium with the given velocity.
fin = equilibrium(1, vel)

###### Main time loop ##################################################################
for time in range(maxIter):
    # Right wall: outflow condition.
    fin[col3,-1,:] = fin[col3,-2,:]

    # Compute macroscopic variables, density and velocity.
    rho, u = macroscopic(fin)

    # Left wall: inflow condition.
    u[:,0,:] = vel[:,0,:]
    rho[0,:] = 1/(1-u[0,0,:]) * ( sum(fin[col2,0,:], axis=0) +
                                  2*sum(fin[col3,0,:], axis=0) )
    # Compute equilibrium.
    feq = equilibrium(rho, u)
    fin[[0,1,2],0,:] = feq[[0,1,2],0,:] + fin[[8,7,6],0,:] - feq[[8,7,6],0,:]

    # Collision step.
    fout = fin - omega * (fin - feq)

    # Bounce-back condition for obstacle.
    for i in range(9):
        fout[i, obstacle] = fin[8-i, obstacle]

    # Streaming step.
    for i in range(9):
        fin[i,:,:] = roll(
                            roll(fout[i,:,:], v[i,0], axis=0),
                            v[i,1], axis=1 )

    # Visualization of the velocity.
    if (time%100==0):
        plt.clf()
        plt.imshow(sqrt(u[0]**2+u[1]**2).transpose(), cmap=cm.Reds)
        plt.savefig("vel.{0:03d}.png".format(time//100))
```

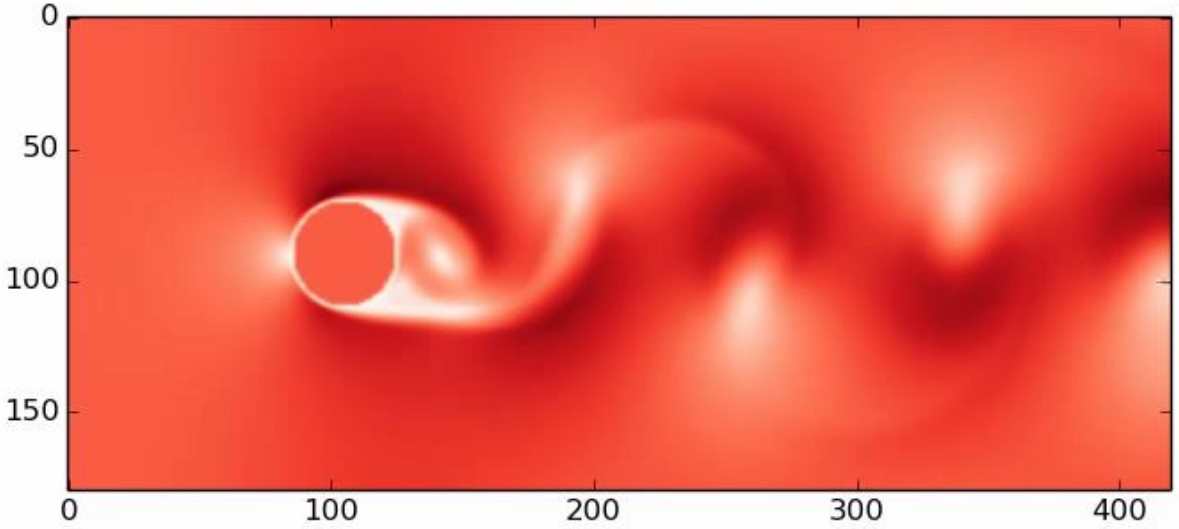
In detail: simulation parameters

```
nx, ny = 420, 180  # Number of cells
r = ny//9          # Radius of obstacle
uLB = 0.04         # Velocity at inlet
Re = 220.0         # Reynolds number
nulb = uLB*r/Re    # Viscosity
omega = 1 / (3*nulb+0.5)   # Relax. parameter
```

Result: Re=10



Result: Re=220



And this ends our introduction to the Lattice Boltzmann Method…