

Jonas Lätt

# La méthode de Boltzmann sur réseau pour la simulation des fluides

<https://www.coursera.org/learn/modeling-simulation-natural-processes/>

# Qu'est-ce qu'un fluide?

Un fluide...

- ... se déforme de manière continue sous une contrainte.
- ... résiste légèrement à la déformation, à cause de la viscosité.
- ... adopte la forme de n'importe quel récipient dans lequel il coule.

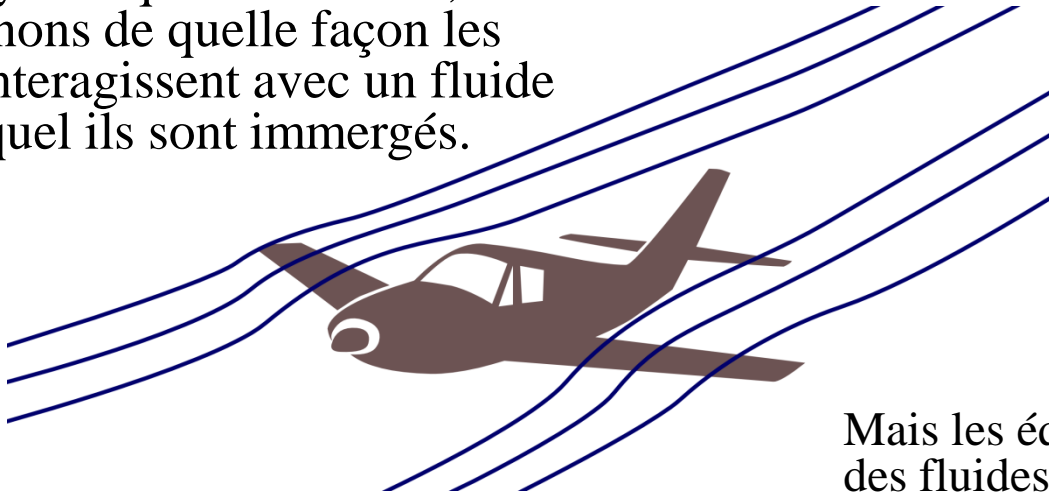
Catégories de fluides

- Liquides
- Gaz
- Plasmas

Fluide = Modèle du continu

# But de la dynamique des fluides

Par la dynamique des fluides, nous comprenons de quelle façon les objets interagissent avec un fluide dans lequel ils sont immergés.



Mais les équations de la dynamique des fluides sont difficiles à résoudre: une résolution numérique est nécessaire.

# On retrouve la dyn. des fluides partout

Domaines d'application importants:

- La physique.
- La biologie et physique médicale.
- La chimie.
- La géologie.
- ... et beaucoup d'autres.

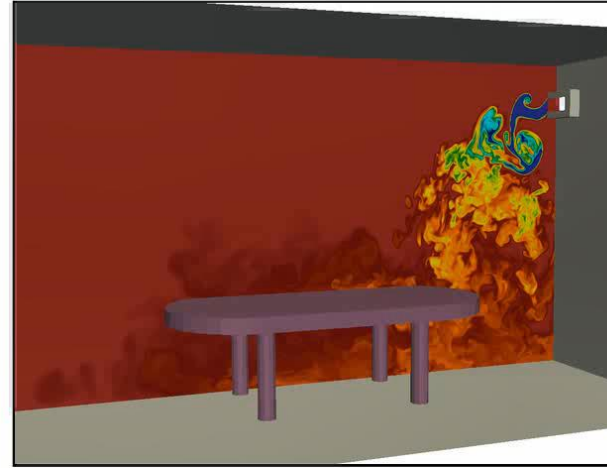
# Exemple 1: Convection thermique

Air-conditioner in a meeting room

 [palabos.org](https://palabos.org)



Fixed air-conditioner



Sweeping air-conditioner

# Exemple 2: Ecoulement sanguin



 palabos.org

Ségment d'une artère humaine avec anévrisme (déformation en bulle)

Simulation:

- Dynamique d'un écoulement sanguin.
- Globules rouges immergés.
- Coagulation du sang dans l'anévrisme.

# Exemple 2: Ecoulement sanguin



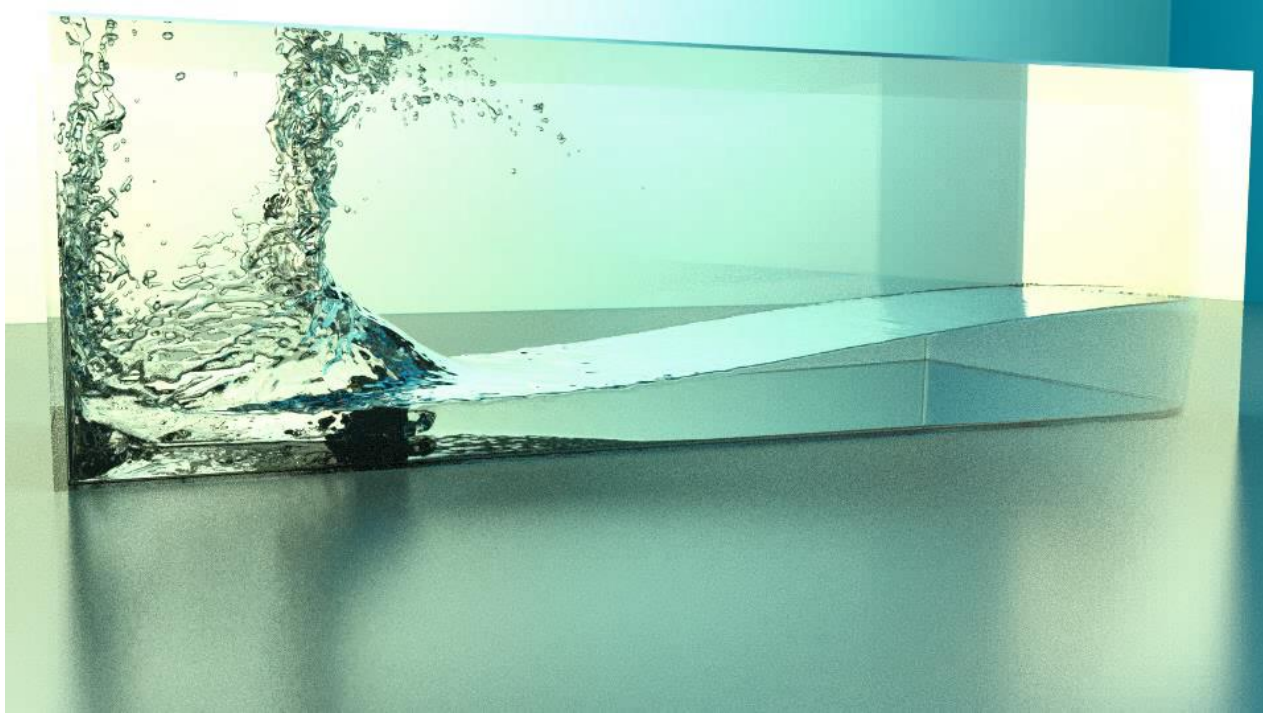
 palabos.org

Ségment d'une artère humaine avec anévrisme (déformation en bulle)

Simulation:

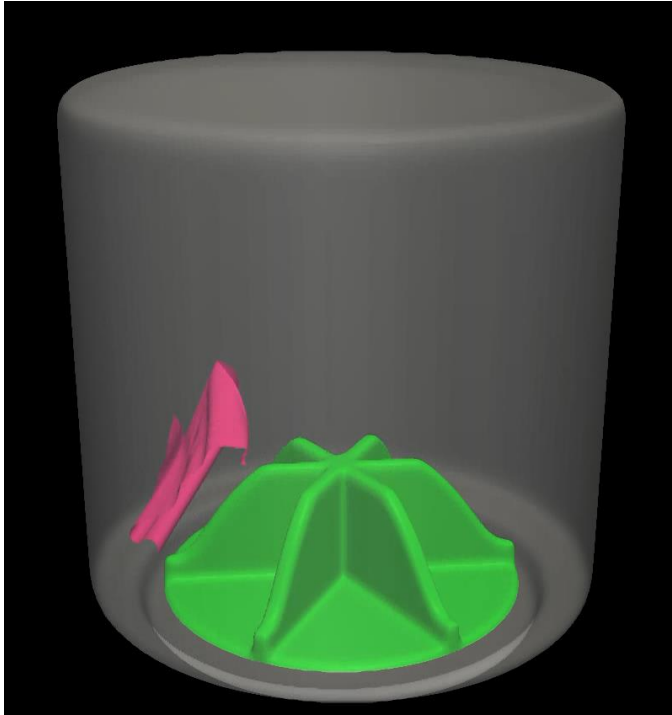
- Dynamique d'un écoulement sanguin.
- Globules rouges immergés.
- Coagulation du sang dans l'anévrisme.

# Exemple 3: Colonne d'eau





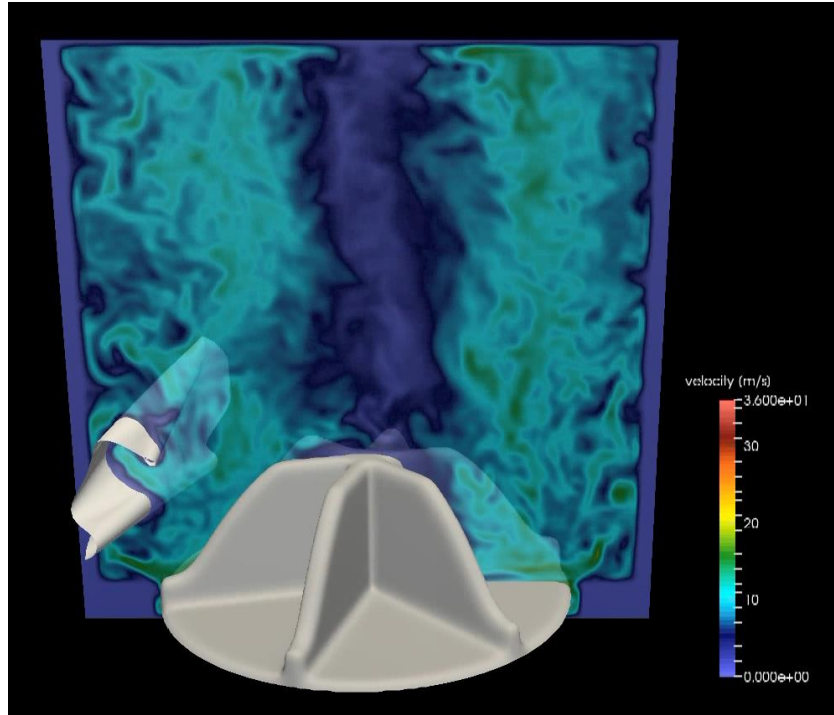
# Exemple 4: Machine à laver



## Simulation:

- Ecoulement d'eau.
- Agitateur en rotation.
- Tissu flexible, couplage bi-directionnel avec l'eau.

# Exemple 4: machine à laver



Machine à laver:  
champ de vitesse.

# Remerciements

- **Orestis Malaspinas** (Université de Genève) a créé l'exemple de simulation de flux sanguin.
- **Dimitrios Kontaxakis** (FlowKit Ltd, Lausanne) a créé l'application de la machine à laver.
- **Andrea Parmigiani** (Université de Genève) et **Andrea Di Blasio** (FlowKit Ltd, Lausanne) ont travaillé sur l'exemple de la colonne d'eau.

# Dynamique des fluides computationnelle: Equations et enjeux

# Equations de Navier-Stokes

Equations de base: Equations de Navier-Stokes pour fluides incompressibles.

Elles ne tiennent pas compte de

- La compressibilité dans les gaz.
- Les effets thermiques.
- Les réactions chimiques.
- ... et beaucoup d'autres.

Néanmoins, on les utilise très fréquemment, pour les liquides et les gaz.

# Equations de Navier-Stokes

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

Accélération du  
fluide

Accélération convective

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

Gradient de pression

Dissipation visqueuse

Equation de continuité:  
incompressibilité d'un  
fluide

$\mathbf{u}(\mathbf{x}, t)$	Vitesse
$p(\mathbf{x}, t)$	Pression
$\nu$	Viscosité

# 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:  $L$

$$\begin{aligned} \vec{u}^* &= \frac{\vec{u}}{U} & \partial_t^* &= \frac{L}{U} \partial_t \\ p^* &= \frac{p}{\rho_0 U^2} & \vec{\nabla}^* &= L \vec{\nabla} \end{aligned}$$

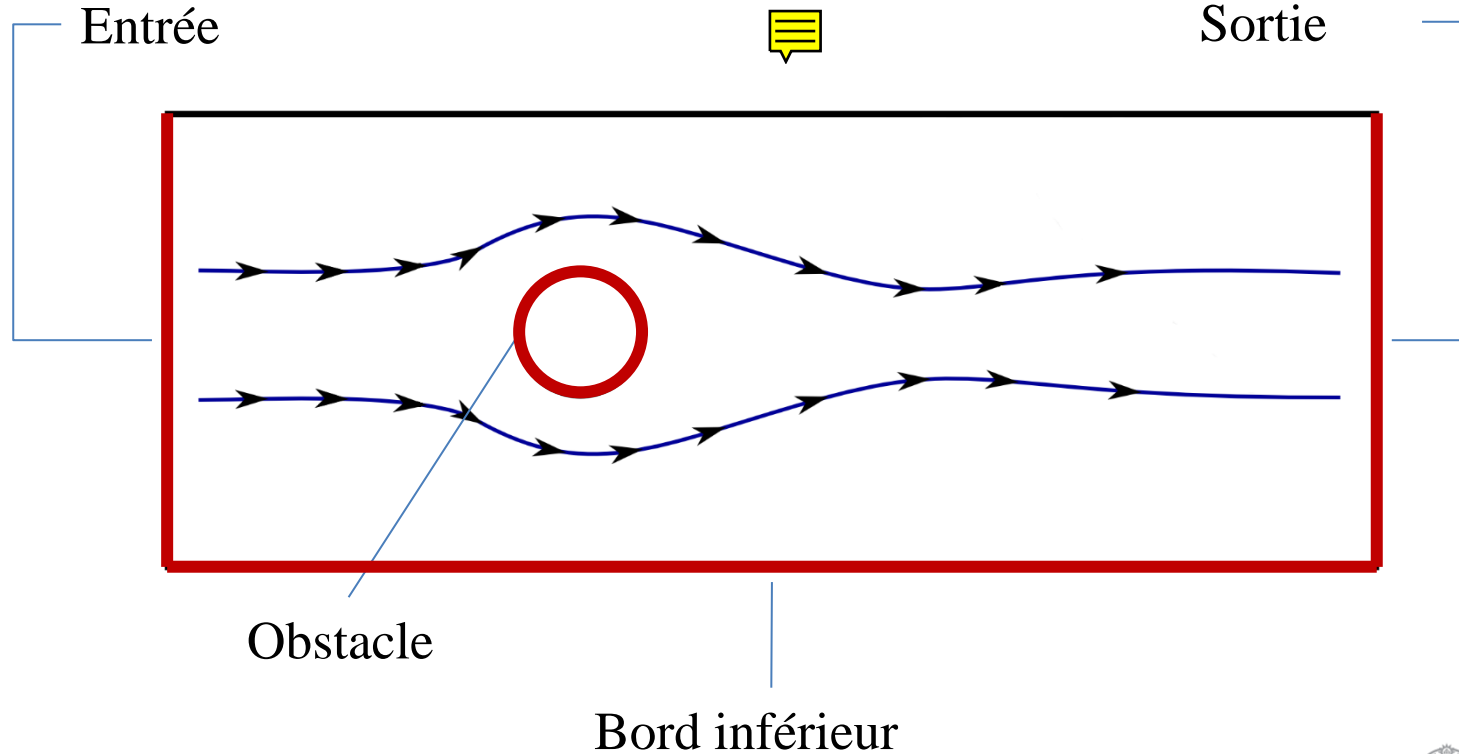
Reynolds number:

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



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

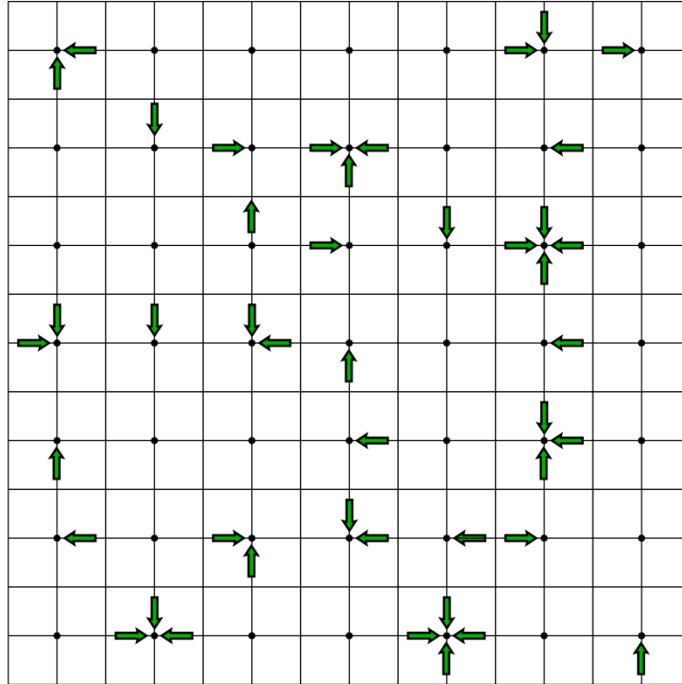
# Conditions aux bords





# Prédécesseur du Lattice Boltzmann: Automate cellulaire à gaz sur réseau

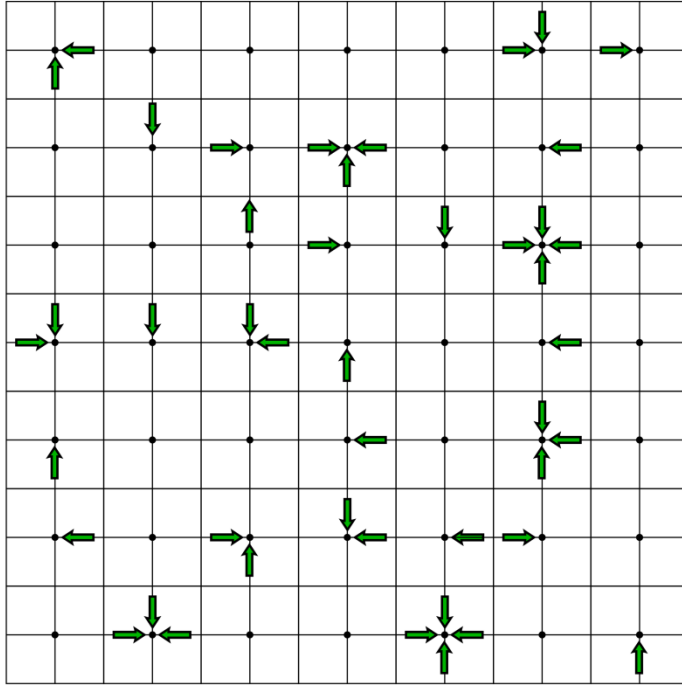
# Gaz sur réseau



**Un automate cellulaire est un réseau de cellules dans lequel:**

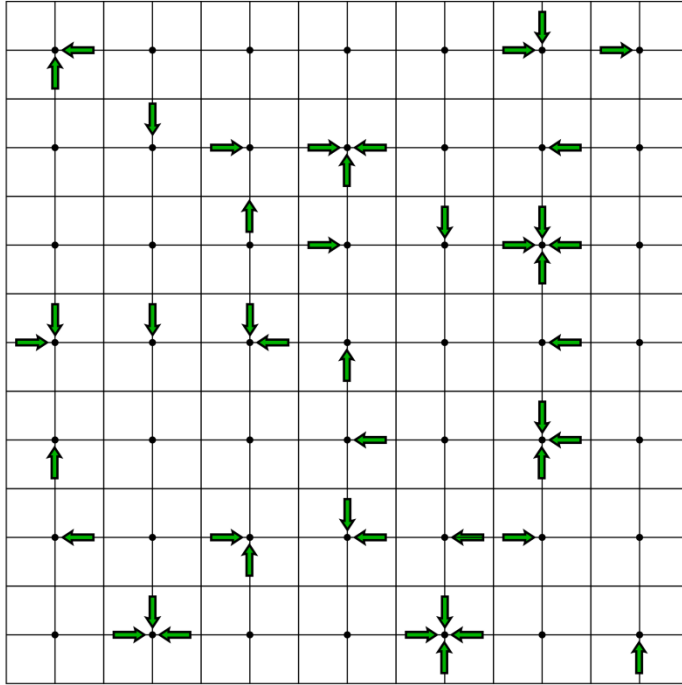
- L'état du système est décrit par des quantités discrètes, booléennes.
- Sur l'image, une cellule est représentée par un point.
- Les degrés de liberté (les particules) sont désignés par  $n_i \in \{0,1\}$ . Chaque cellule possède  $z$  quantités  $n_i(\mathbf{x}, t), i = 1 \cdots z$ . Dans l'exemple à gauche,  $z=4$ .

# Gaz sur réseau



- Chaque cellule dans cet exemple est peuplée par quatre particules au plus, représentées par des flèches vertes.
- La flèche indique la direction dans laquelle se propage la cellule.
- La constante  $z$  est appelé nombre de coordination.
- Les voisins de la cellule  $x$  sont obtenus par  $x + v_i \delta x$ . Les directions du réseau sont  $v_1 = (1,0)$ ,  $v_2 = (0,1)$ ,  $v_3 = (-1,0)$ , and  $v_4 = (0,-1)$ .

# Gaz sur réseau



**La règle d'évolution consiste de deux pas:**

- **Pas d'interaction:** Les quantités  $n_i$  «entre en collision» localement et les nouvelles valeurs  $n_i'$  sont calculées, avec un opérateur de collision  $\Omega_i(n)$ .
- **Pas de propagation:** La quantité  $n_i'(x)$  est envoyée aux sites voisins le long de la direction du réseau  $v_i$ .

## Commentaire

- Ce modèle mésoscopique décrit la dynamique d'un **gaz**, pas d'un liquide.

# Gaz sur réseau

## Microdynamique

- Collision:  $n_i^{\text{out}}(\mathbf{x}, t) = n_i^{\text{in}}(\mathbf{x}, t) + \Omega_i(\mathbf{x}, t)$
- Propagation:  $n_i^{\text{in}}(\mathbf{x}, t) = n_i^{\text{out}}(\mathbf{x} - \mathbf{v}_i \delta x, t - \delta t)$
- La constante  $\delta t$  possède les unités du temps.
- Durant la propagation, la particule  $n_i$  se propage en direction de  $\mathbf{v}_i$ , atteint la cellule à la position  $\mathbf{x} + \mathbf{v}_i \delta x$ , et maintient sa direction  $\mathbf{v}_i$ .

# Gaz sur réseau

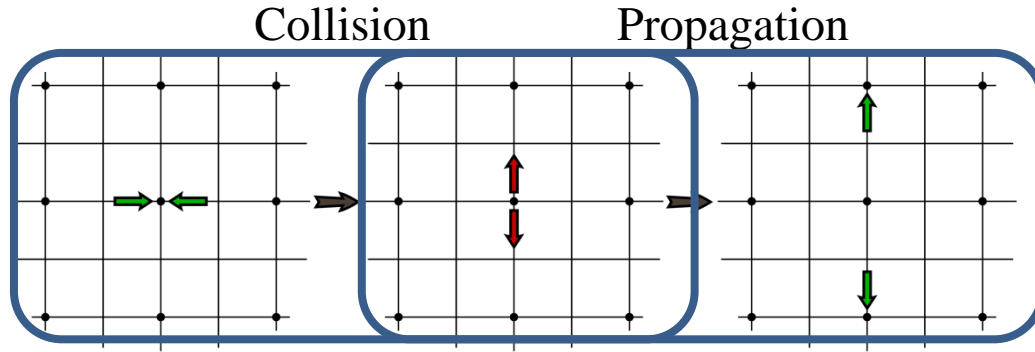
## Quelque modèles de fluide simples

- **HPP**: Hardy, Pomeau, de Pazzis, 1971: Théorie cinétique de particules sur un réseau rectangulaire.
- **FHP**: Frisch, Hasslacher and Pomeau, 1986: premier gaz sur réseau reproduisant (presque) le comportement hydrodynamique correct (c'est-à-dire les équations de Navier-Stokes).

# Gaz sur réseau

## Exemple: collision et propagation dans le modèle HPP

- Durant la collision, les nombres d'occupation dans les états de post-collision (rouge) se construisent à partir de l'état pré-collision. Nous ne fournissons pas les détails de la collision ici, mais juste une exemple.
- La propagation applique l'état de post-collision (rouge) sur l'état de pré-collision (vert) sur les cellules voisines.



Notre schéma de couleurs:

- ➡ Vert: pré-collision
- ➡ Rouge: post-collision

# Du gaz sur réseau vers le Boltzmann sur réseau

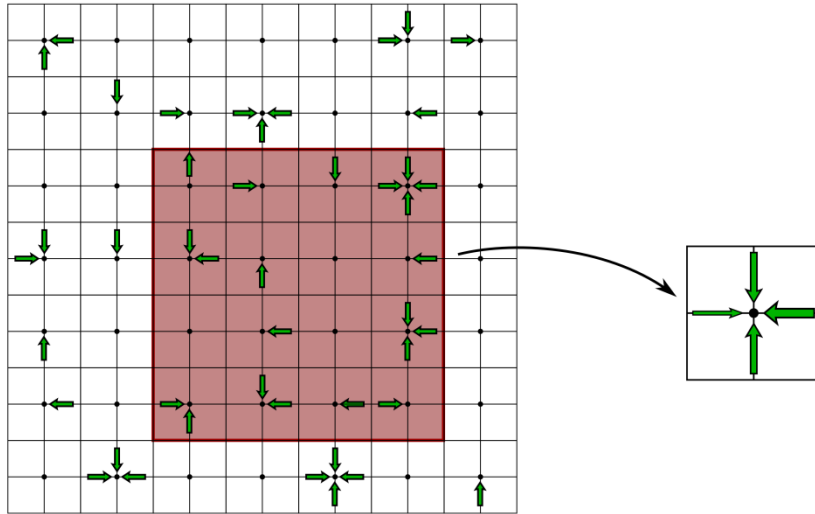


# Pourquoi ne pas rester avec les gaz sur réseau?

## Quelques problèmes des gaz sur réseau:

- Le gaz sur réseau est de plus bas niveau («mésoscopique») que les modèles de Navier-Stokes («macroscopique»), et représentent plus de détails de la physique du gaz. La plupart du temps, nous n'avons pas besoin de ces détails
- Bruit moléculaire: on a besoin de beaucoup de particules pour atteindre une moyenne statistique raisonnable.
- Matériel des ordinateurs: Les ordinateurs modernes sont très efficaces pour des calculs en virgule flottante. Les gaz sur réseau discrets perdent leurs avantages.

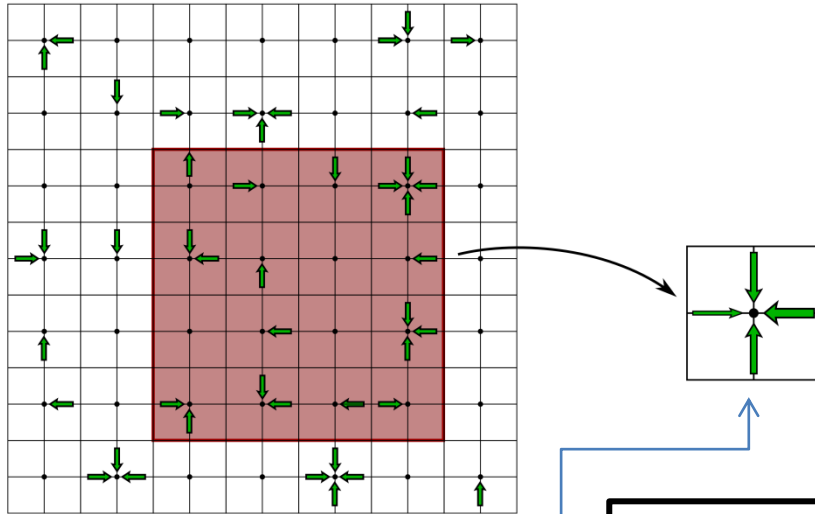
# Des variables discrètes aux variables continues



## Dans le gaz sur réseau

- Exécution du programme sur des variables discrètes.
- A la fin, extraction des variables macroscopiques par le biais de moyennes statistiques.

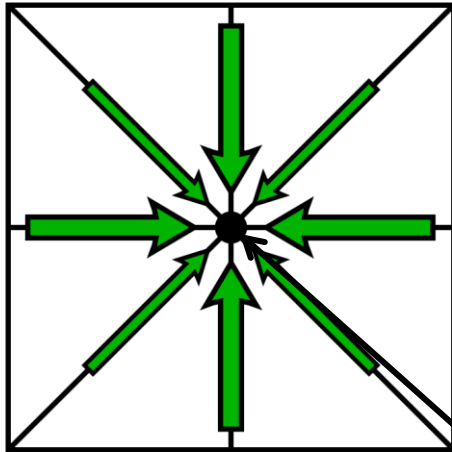
# Boltzmann sur réseau: idée



- On exécute le programme directement avec des variables continues, pour gagner de l'espace.
- Le modèle continu est obtenu à partir du modèle discret par des méthodes de mécanique statistique.

- Densité de particules: valeur réelle.
- L'épaisseur de la flèche indique la densité de particule.

# Boltzmann sur réseau: détails

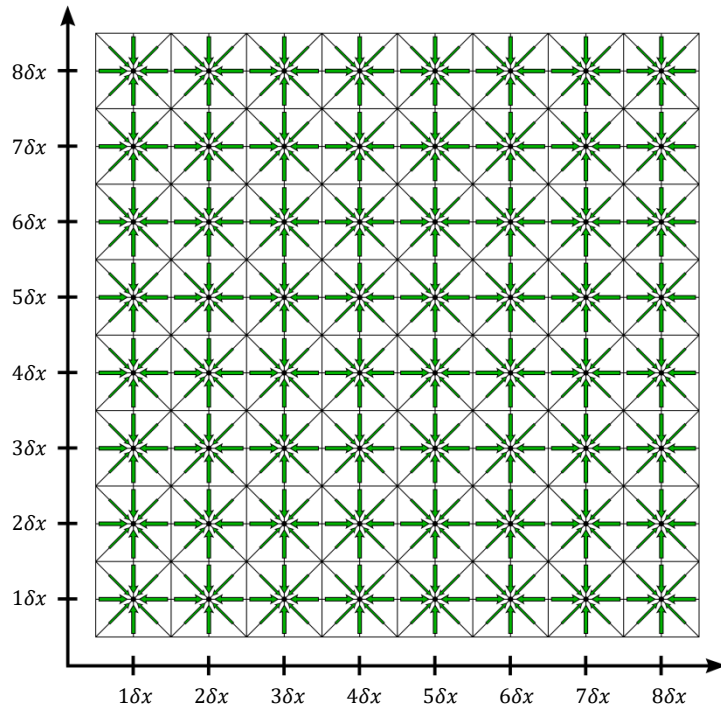


Modèle bi-dimensionnel D2Q9:

- Quatre directions ne suffisent pas.
- Neuf directions: huit connections aux plus proches voisins + «population au repos».
- Les variables représentant les densités de particules s'appellent «populations».

Les particules de la «population au repos» restent sur la cellule, et ne se propagent pas vers un voisins durant l'étape de propagation.

# Variables du Boltzmann sur réseau



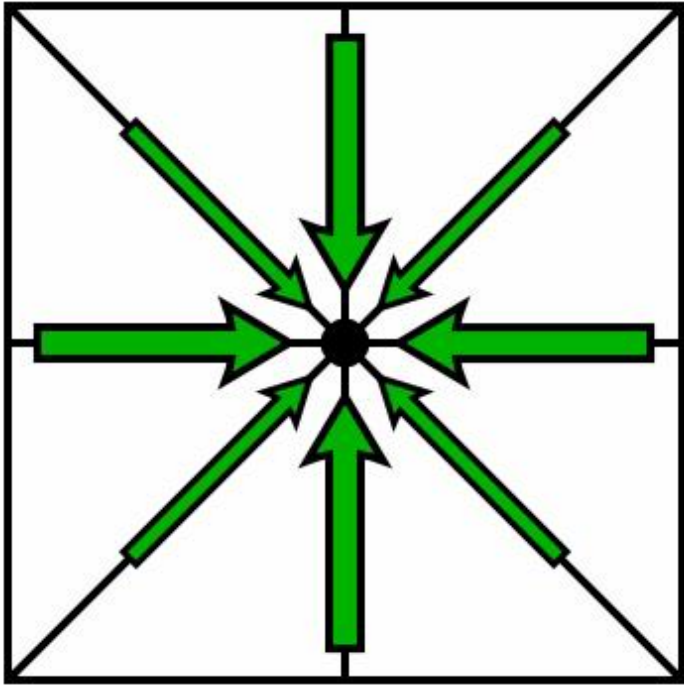
## Discrétisation de l'espace

- Chaque cellule contient les variables représentant le fluide en un point de l'espace.
- On utilise une distance égale  $\delta x$  entre cellules en direction x et y.
- Si on représente l'espace par  $8 \times 8$  points, on a besoin de  $8 \times 8 \times 9 = 576$  variables en virgule flottante.

## Allocation de la mémoire en Python

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

# Etape de collision



- Comme dans les gaz sur réseau: la collision applique l'état pré-collision sur l'état post-collision.
- Toutes les populations sont modifiées durant la collision. On ne peut pas voir les changements sur l'animation, car ils sont trop faibles

Notre schéma de couleur:

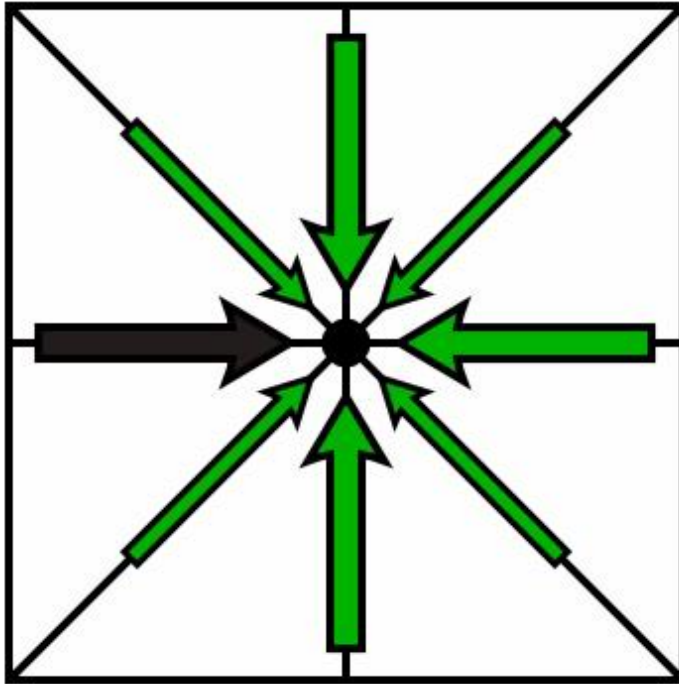


Vert: pré-collision



Rouge: post-collision



# Etape de collision



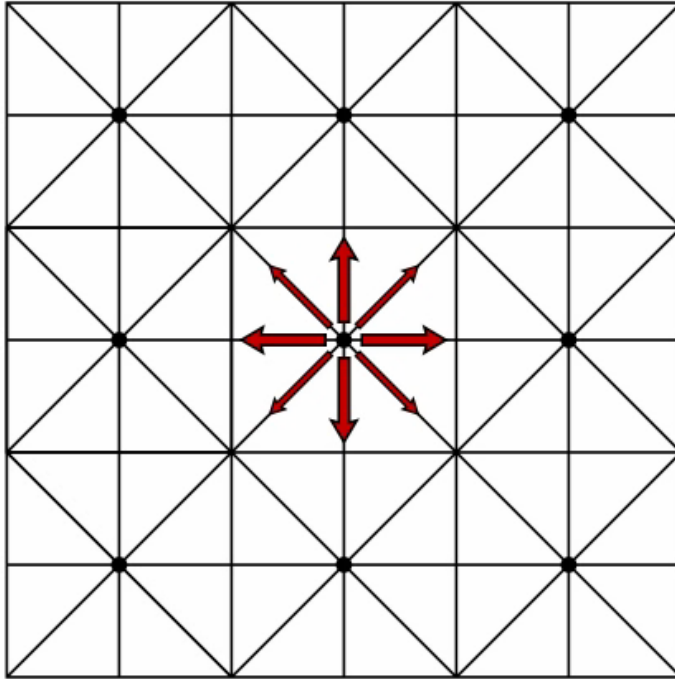
Modèle:

- Collision instantanée.
- Collision localisée sur une cellule.

Notre schéma de couleur:

 Vert: pré-collision  
 Rouge: post-collision

# Etape de propagation



Modèle:

- La propagation transporte le système du temps  $t$  au temps  $t + \delta t$ .
- Accès aux plus proches voisins.

Notre schéma de couleur:



Vert: pré-collision

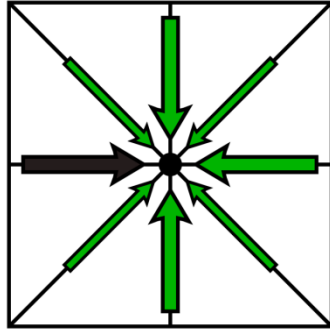


Rouge: post-collision

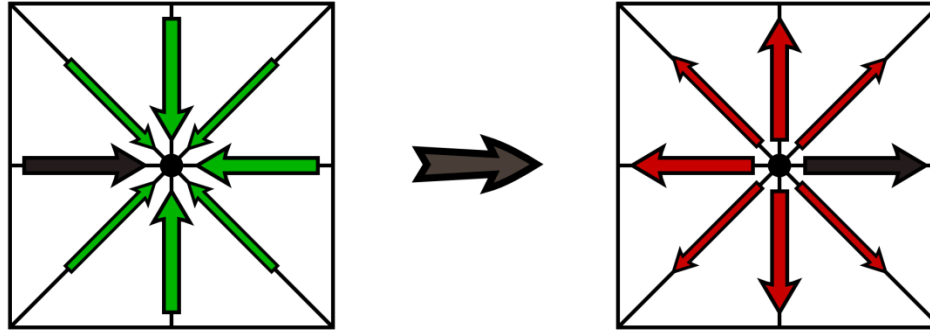


# Variables macroscopiques

# Retour à l'étape de collision



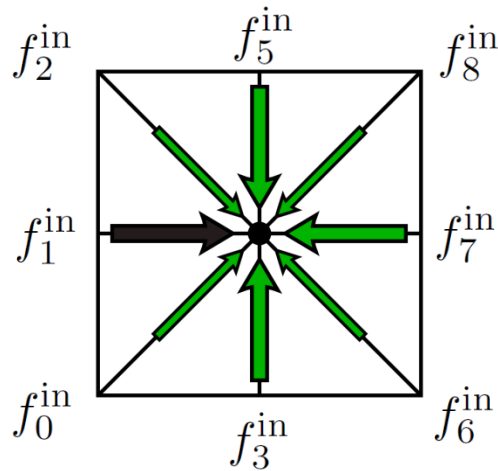
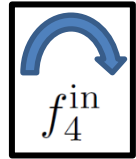
# Retour à l'étape de collision



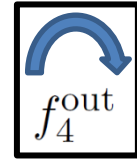
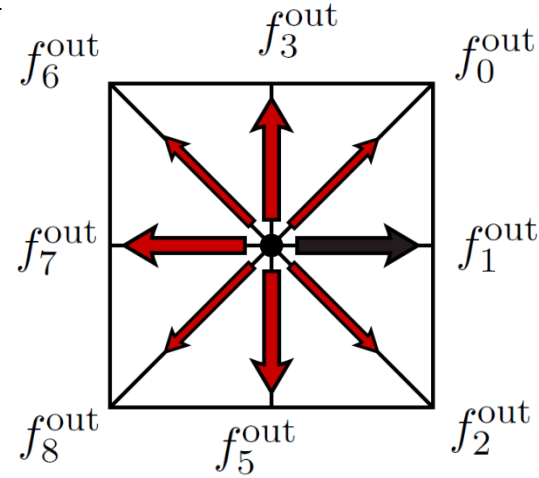
Pré-collision:  $f_i^{\text{in}}(\mathbf{x}, t)$   
«Populations entrantes»

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

# Retour à l'étape de collision



Pré-collision:  $f_i^{\text{in}}(\mathbf{x}, t)$   
«Populations entrantes»



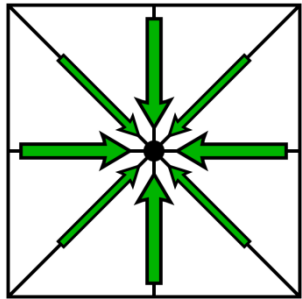
Post-collision:  $f_i^{\text{out}}(\mathbf{x}, t)$   
«Populations sortantes»

L'indice  $i$  va de 0 à 8.

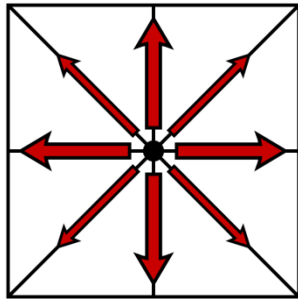
# Pré- et post-collision: variables

Dans notre code Python, nous sauvegardons les variables entrantes et sortantes dans des matrices différentes.

$f^{\text{in}}$



$f^{\text{out}}$



## Code Python:

```
# à une taille donnée...  
 $nx, ny = \dots$   
 $fin = \text{zeros}(9, nx, ny)$   
 $fout = \text{zeros}(9, nx, ny)$ 
```

# Densité

La densité de particules est définie comme dans un gaz sur réseau:

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

Chaque cellule possède sa propre densité, qui est la somme des neufs populations.

# Densité

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

**Code Python:**

```
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]
```

**Plus simple et plus rapide, avec la notation matricielle de NumPy:**

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

# Pression

La pression est proportionnelle à la densité, à température constante. La constante de proportionnalité est la vitesse du son au carré:

$$p = c_s^2 \rho$$

Dans ce modèle D2Q9, la vitesse du son  $c_s$  est une constante:

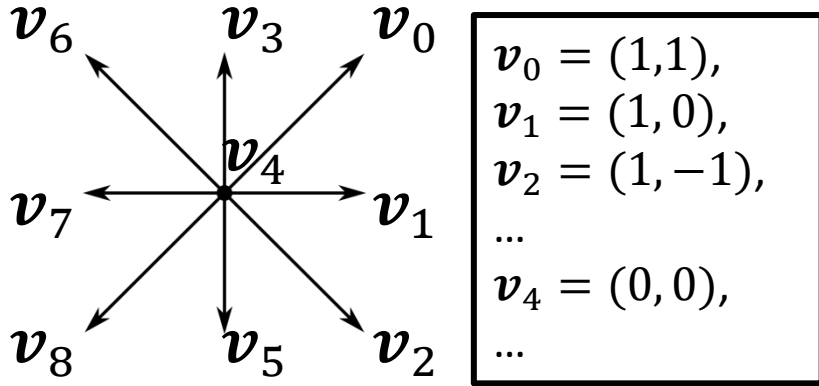
$$c_s^2 = \frac{1}{3} \frac{\delta x^2}{\delta t^2}$$

Unités: Disons que  $\delta x$  possède les unités d'un mètre  $[m]$  et  $\delta t$  les unités de secondes  $[s]$ . Alors la pression possède les unités  $\frac{m^2}{s^2}$ . En multipliant par la densité du fluide (p. ex.  $1 \frac{kg}{m^3}$ ), on obtient l'unité du Pascal  $\left(\frac{kg}{ms^2}\right)$ .



# Vitesse

Introduisons un ensemble de 9 vecteurs:



Une fois de plus, la vitesse est définie comme dans un gaz sur réseau:

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

On les appelle les *constantes du réseau*: S'il y a une cellule en  $\mathbf{x}$ , il y en a aussi une en  $\mathbf{x} + \mathbf{v}_i \delta x$ .

# Vitesse

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

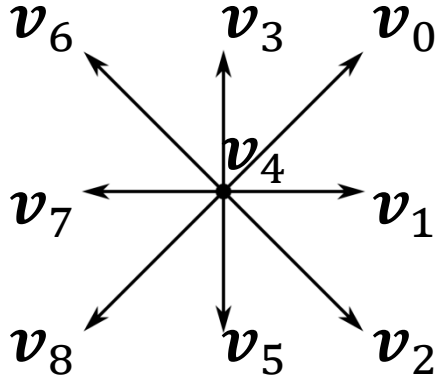
## Code Python traditionnel:

```
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]
```

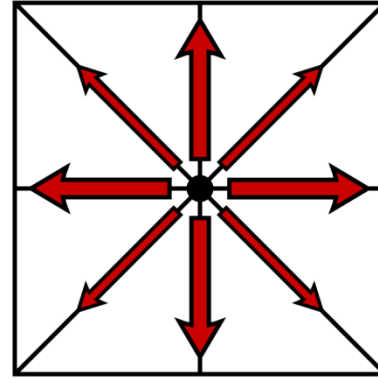
## Plus simple et plus rapide, avec la notation matricielle de NumPy:

```
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
```

# Attention à la confusion!



- Il s'agit de 9 vecteurs 2D.
- Leur valeur est constante.

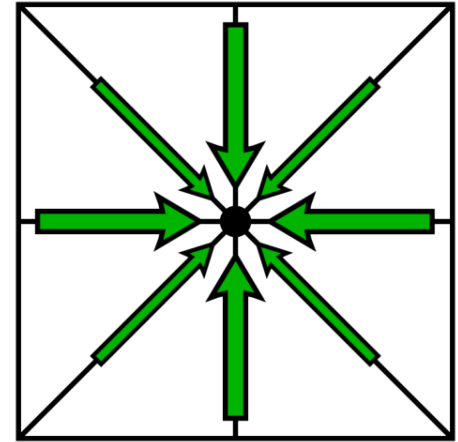


- Il s'agit de 9 valeurs scalaires.
- Leur valeur dépend du temps.

# Etape de collision: le modèle BGK

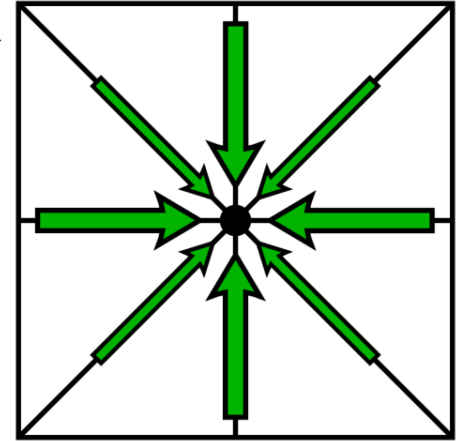
# Que valent les populations?

- Jusqu'à présent, nous avons interprété les populations comme des moyennes des particules d'un gaz sur réseau.
- Interprétation alternative: les populations représentent des densités de probabilité de particules dans un gaz réel.
- En *théorie cinétique*, les densités de gaz sont décrites par une *fonction de densité de probabilité*  $f(\mathbf{x}, \mathbf{v}, t)$ .
- L'évolution de  $f$  est décrite par l'*équation de Boltzmann*.



# Que valent les populations?

- Même lorsqu'un gaz est au repos macroscopiquement, ou bouge à une vitesse constante  $\mathbf{u}$ , ses molécules sont quand même en mouvement.
- La distribution de leurs vitesses  $\mathbf{v}$  en tout point  $\mathbf{x}$  est donnée par une *distribution Maxwell-Boltzmann* :
$$f(\mathbf{x}, \mathbf{v}, t) \sim e^{-|\mathbf{v}-\mathbf{u}|^2}.$$
- Et lorsqu'il est en mouvement, un gaz reste proche de son état d'équilibre. Nous considérons son mouvement comme une *perturbation autour de l'équilibre*.



# Collision: le modèle BGK

Inspiré par la théorie cinétique, le terme de collision dans le modèle BGK est formulé comme une relaxation vers l'équilibre local:

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

- $E$  est l'équilibre local. Il dépend des variables macroscopiques et possède une valeur différente pour chaque direction  $i$ .
- Le paramètre  $\omega$  est la fréquence de relaxation.
- Ce modèle de collision s'appelle le *modèle BGK*.

# Fréquence de collision et viscosité

- Si  $\omega$  est petit, le fluide converge lentement vers son équilibre local: il est très visqueux.
- Plus généralement, on peut montrer que la viscosité dépend de la valeur inverse du paramètre de relaxation  $\omega$ :

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



# Collision: modèle BGK

L'équilibre s'obtient par une série tronquée de la distribution de Maxwell-Boltzmann (une distribution Gaussienne autour de la vitesse moyenne  $\mathbf{u}$ ):

$$E(i, \rho, \mathbf{u}) = \rho t_i \left( 1 + \frac{\frac{\delta x}{\delta t} \mathbf{v}_i \cdot \mathbf{u}}{c_s^2} + \frac{1}{2 c_s^4} \left( \frac{\delta x}{\delta t} \mathbf{v}_i \cdot \mathbf{u} \right)^2 - \frac{1}{2 c_s^2} |\mathbf{u}|^2 \right)$$

- Les constantes  $t_i$  corrigent la différence de longueur de directions  $\mathbf{v}_i$ .
- $t_i$  vaut 1/9 pour les directions orthogonales, 1/36 pour les directions diagonales, et 4/9 pour la vitesse nulle (la population au repos):

$t = \text{array}([1/36, 1/9, 1/36, 1/9, 4/9, 1/9, 1/36, 1/9, 1/36])$

# Collision: Codes Python

Equilibre:

$$E(i, \rho, \mathbf{u}) = \rho t_i \left( 1 + \frac{\frac{\delta x}{\delta t} \mathbf{v}_i \cdot \mathbf{u}}{c_s^2} + \frac{1}{2 c_s^4} \left( \frac{\delta x}{\delta t} \mathbf{v}_i \cdot \mathbf{u} \right)^2 - \frac{1}{2 c_s^2} |\mathbf{u}|^2 \right)$$

Nous formulons ce code en «unités du réseau», c'est-à-dire avec  $\delta x = \delta t = 1$ .

```
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: Codes Python

Collision:

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

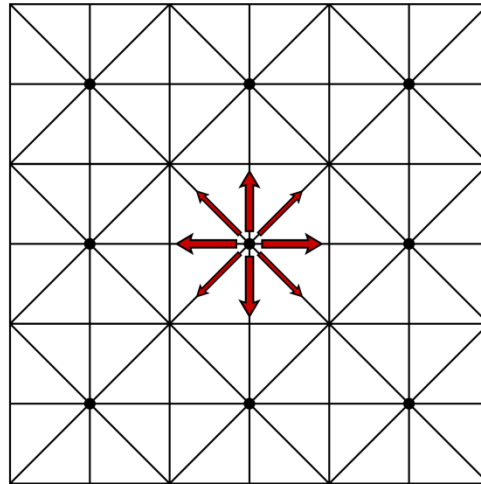
$$f_{\text{out}} = f_{\text{in}} - \text{omega} * (f_{\text{in}} - eq)$$

# L'étape de propagation

# Etape de propagation

La propagation  
transporte le  
système vers  
son prochain  
pas de temps,  
 $t + \delta t$

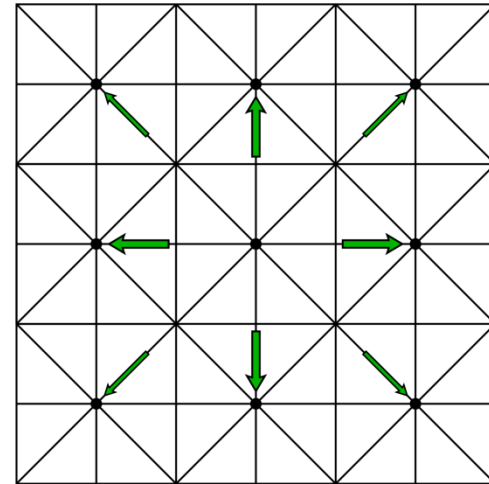
Temps  $t$



Post-collision  $f^{\text{out}}$



Temps  $t + \delta t$



Pré-collision  $f^{\text{in}}$



# Etape de propagation: formule

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

## L'étape de propagation

- Transporte les populations sortantes (rouges) sur les populations entrantes (vertes).
- Copie les populations vers les cellules voisines, le long de la direction impliquée par l'indice  $i$ .
- Transporte le système du temps  $t$  vers le temps  $t + \delta t$ .

# Etape de propagation: code

Code Python conventionnel:

- Sur les bords, une condition de périodicité est appliquée.
- Si une population quitte le domaine, elle rentre dans le domaine du côté opposé.

```
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_y < 0: next_y = ny-1  
            if next_y >= ny: next_y = 0  
            fin[i, next_x,next_y] = fout[i, ix, iy]
```

# Etape de propagation: code

Code Python conventionnel:

```
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]
```

Code NumPy matriciel:

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

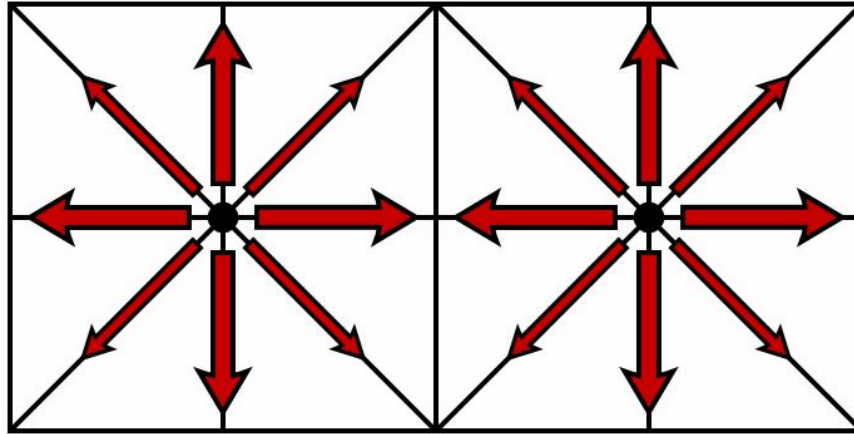


# Est-il vraiment nécessaire de doubler la mémoire?

- Dans ce qui précède, deux matrices de populations  $f^{in}$  et  $f^{out}$  ont été allouées.
- Est-il possible de s'en passer d'une, et de sauvegarder l'état pré- et post-collision dans la même variable  $f$ ?
- Réponse: oui, en s'y prenant soigneusement.
- Je vais vous montrer comment vous y prendre. Néanmoins, par souci de simplicité, le code développé durant cette séance utilisera deux matrices pour sauvegarder les populations.

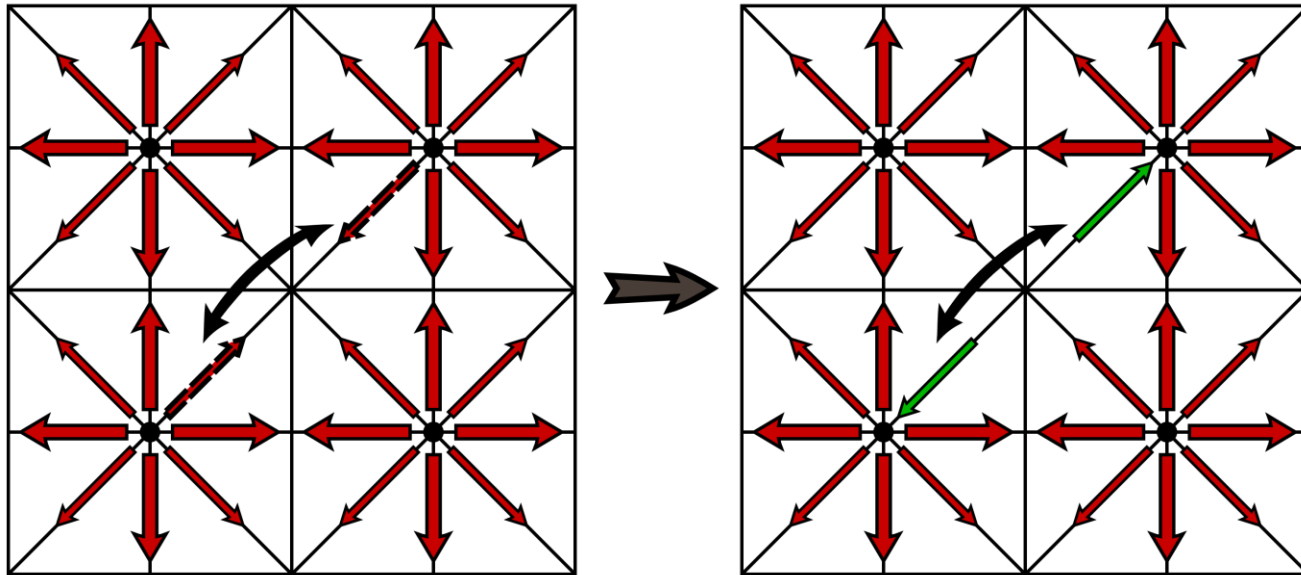
# Implémentation à population unique

- La collision est locale, pas de problème. On efface les  $f^{in}$  par les  $f^{out}$ .
- La propagation est plus subtile. En propageant les populations vers un voisin, on risque d'effacer des données dont on a encore besoin.
- Solution: *on propage vers le voisin, et du voisin, en même temps en échangeant des populations au lieu de les effacer.*



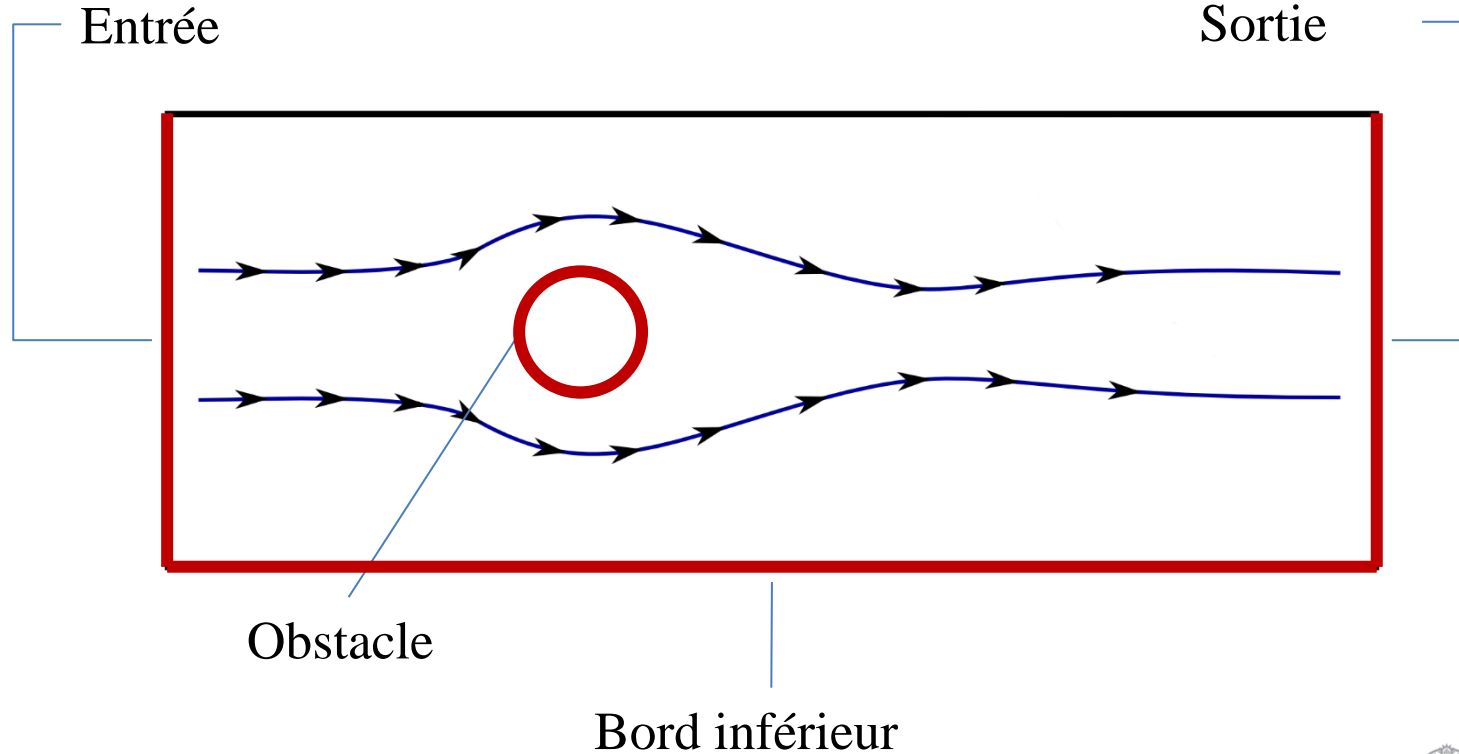
# Implémentation à population unique

Cette stratégie s'applique le long de toutes les directions:



# Conditions aux bords

# Conditions aux bords



# Conditions aux bords

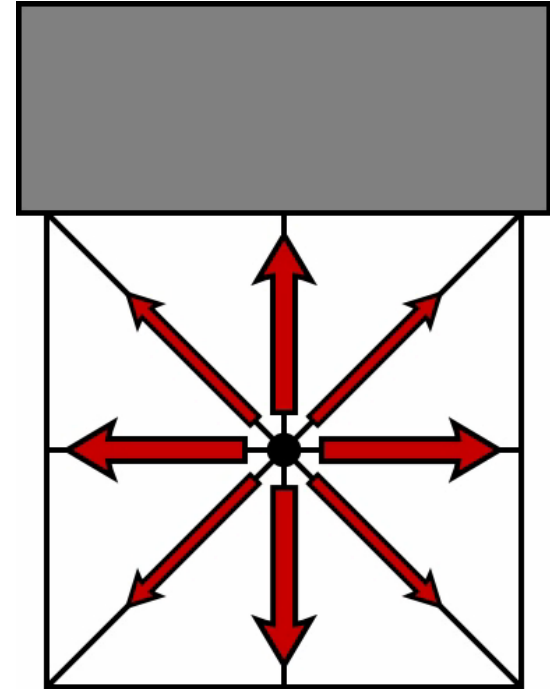
- Différents types de conditions aux bords sont nécessaires:
  - Entrée: profil de vitesse imposé.
  - Sortie et bords inférieur/supérieur: on fait semblant que le domaine est effectivement plus grand.
  - Obstacle physique.
- Notre code possède déjà une condition aux bords: la périodicité.
  - Signification: un domaine répété identiquement et infiniment.
  - Nous exploiterons la périodicité sur le bord inférieur/supérieur.

# Obstacle: condition d'adhérence

- La plupart des obstacles exhibent une condition d'adhérence.
- Signification macroscopique: vitesse nulle sur l'interface.
- Signification moléculaire: l'obstacle possède une surface accidentée, et les molécules du fluide y «adhèrent».
- La condition d'adhérence ne s'applique pas à tous les obstacles. Exemple: une bulle d'air est lisse, même au niveau moléculaire.

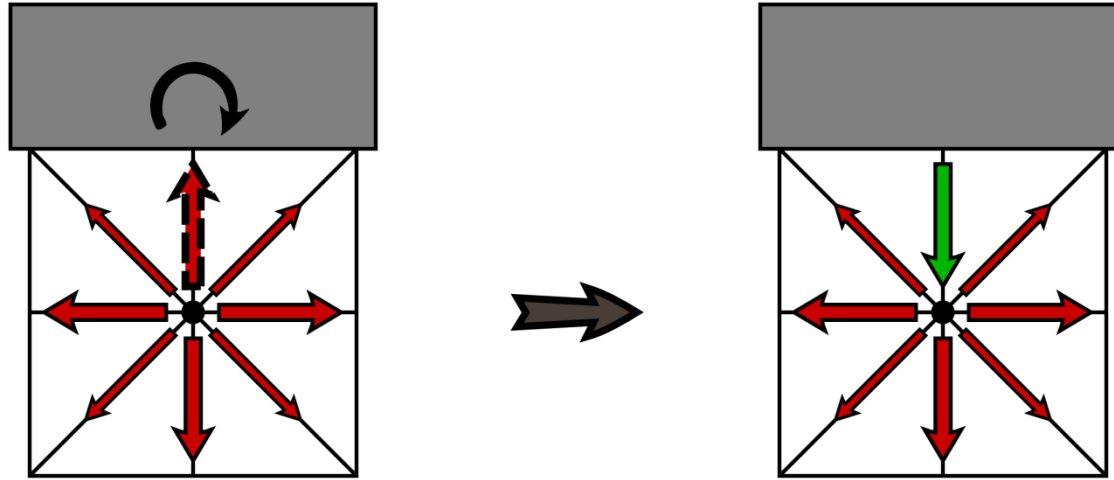
# Adhérence: condition bounce-back

- La condition «bounce-back» permet de mettre en œuvre une condition d'adhérence en Boltzmann sur réseau.
- Durant la propagation, une population qui heurte un obstacle est réfléchie, et revient sur le site qu'elle a quitté.
- Sa valeur reste inchangée: elle est simplement copiée dans la population de vitesse inverse.



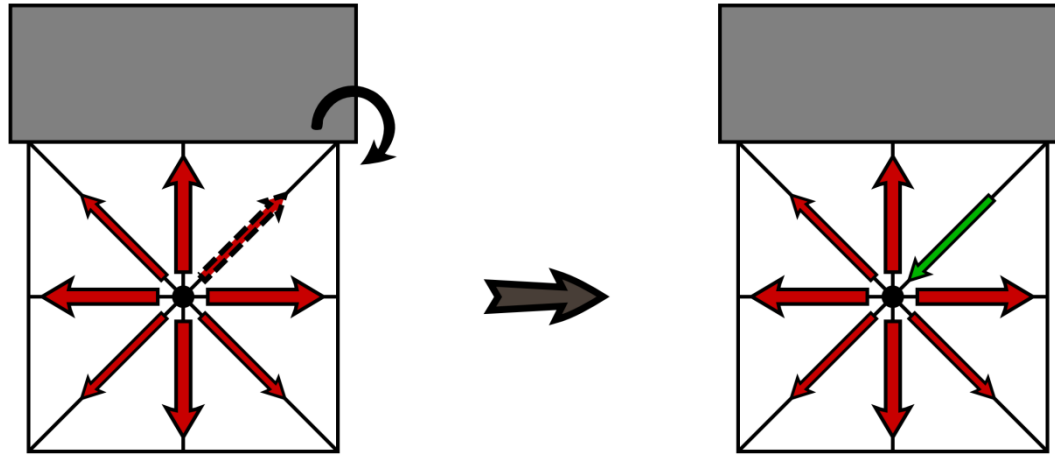


# Adhérence: condition bounce-back



Cette condition imite le processus qui a lieu dans un vrai gaz au niveau microscopique, lorsque des molécules de gaz heurtent une paroi.

# Adhérence: condition bounce-back



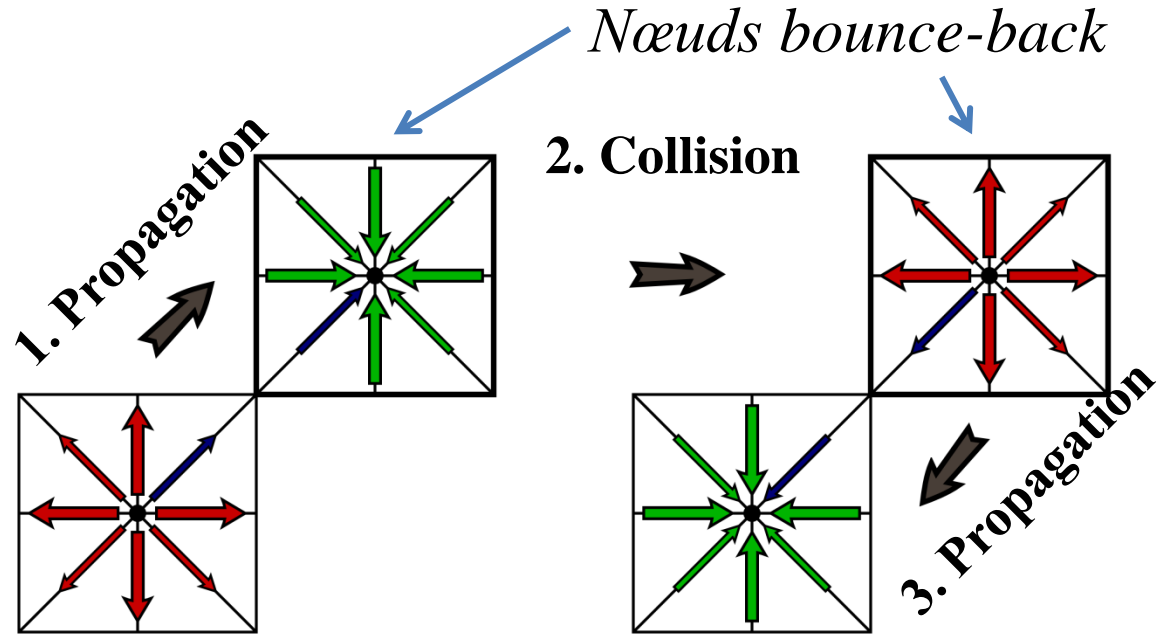
- Attention: Le long des directions diagonales, les populations ne sont pas réfléchies comme sur un miroir. Elles reviennent simplement d'où elles sont venues.
- Cela est lié au fait qu'à un niveau moléculaire, une paroi n'est pas lisse, mais rugueuse.

# Adhérence: condition bounce-back

- La condition de bounce-back est simple: on garde trace des populations qui heurtent un obstacle, et on les renvoie.
- Elles demandent néanmoins un peu de travail. Quels nœuds sont affectés? Quelles directions?
- Solution plus simple: Laissons les populations entrer dans l'obstacle. On invertit alors leur direction à l'intérieur de l'obstacle, puis on les renvoie d'où elles viennent.
- Leurs directions n'ont pas besoin d'être sauvegardées. Toutes cellules à l'intérieur d'un obstacle sont de type bounce-back: à la place d'une étape de collision, elles invertissent la direction de chaque population.

# Le cycle bounce-back

Durant la collision, un nœud bounce-back remplace chaque population par celle attribuée à la direction opposée.



# Le cycle bounce-back

A la place d'une collision, les nœuds bounce-back mettent en œuvre la relation suivante:

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

où la direction  $i$  est opposée à la direction  $j$ :

$$v_i = -v_j$$

# Bounce-back: le code

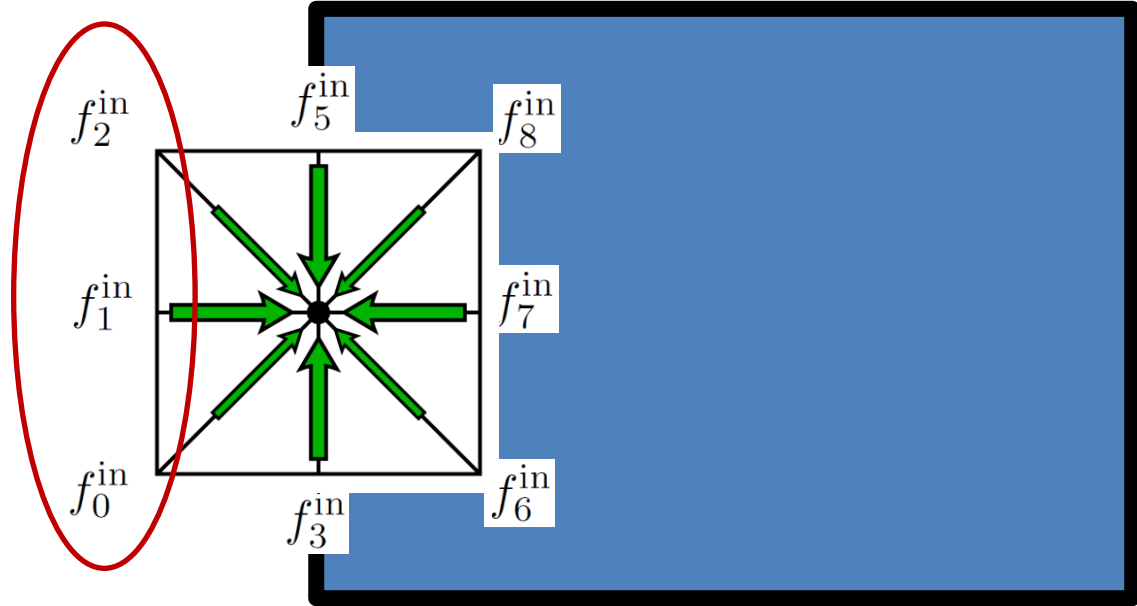
En syntaxe matricielle NumPy:

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

- Nous avons fait exprès de définir les constantes du réseau de manière à ce que la direction opposée à la direction  $i$  soit  $8-i$ .

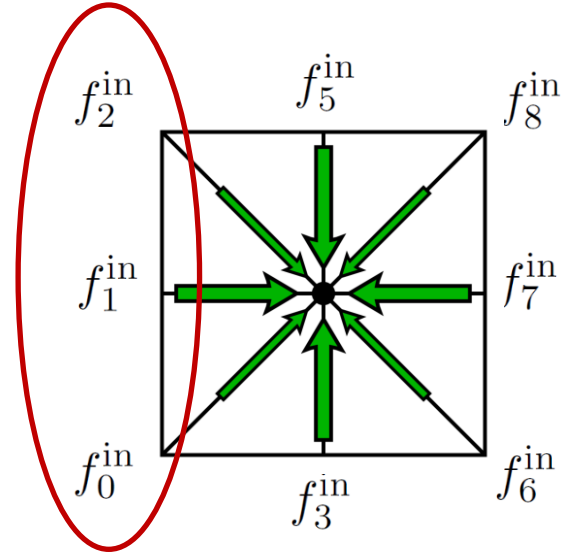
# Condition d'entrée

Après propagation,  
les populations  
 $f_0, f_1$ , et  $f_2$  sont  
inconnues sur  
toutes les cellules  
du bord gauche.



# Condition d'entrée

- Nous voulons imposer une vitesse  $\mathbf{u}$  sur les cellules de l'entrée. Donc,  $\mathbf{u}$  est connu. Mais qu'en est-il de  $\rho$ ? Et de  $f_0, f_1$ , and  $f_2$ ?
- Nous devons déduire ces informations inconnues des populations connues et de la vitesse  $\mathbf{u}$ .





# Condition d'entrée: densité

Rappel:

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

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

Définissons:

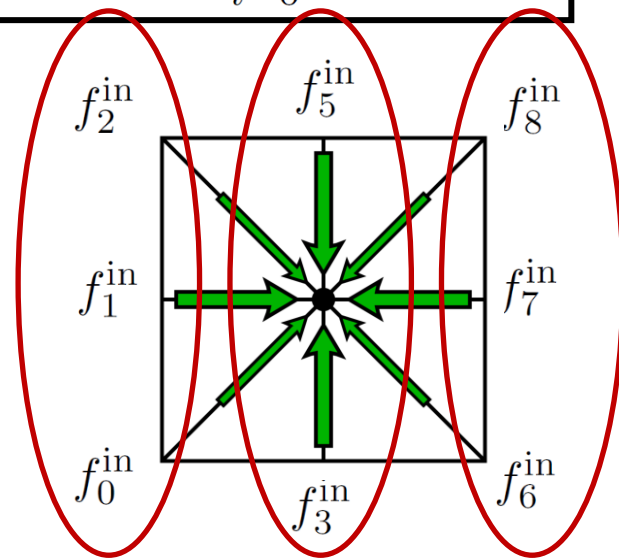
- $\rho_1 = f_0 + f_1 + f_2$  (inconnu)
- $\rho_2 = f_3 + f_4 + f_5$  (connu)
- $\rho_3 = f_6 + f_7 + f_8$  (connu)

Alors:

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

Et donc:

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



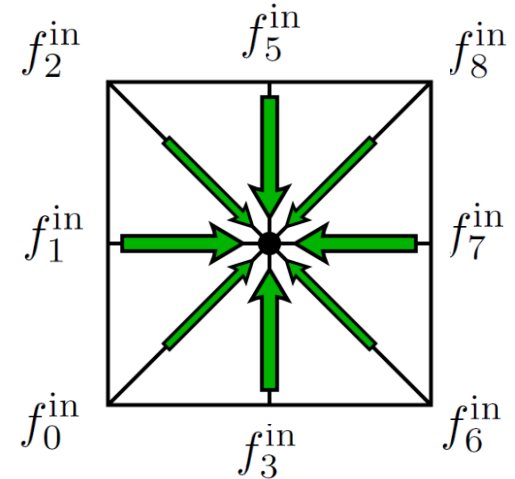
# Condition d'entrée: populations inconnues

- Rappel: les populations sont toujours proches de leur équilibre local.
- Pour commencer, les populations inconnues sont initialisées à l'équilibre.
- Ensuite, nous leur rajoutons une correction de la valeur d'équilibre, qui est copiée de la population opposée.

$$f_0^{\text{in}} = E(0, \rho, \mathbf{u}) + (f_8^{\text{in}} - E(8, \rho, \mathbf{u}))$$

$$f_1^{\text{in}} = E(1, \rho, \mathbf{u}) + (f_7^{\text{in}} - E(7, \rho, \mathbf{u}))$$

$$f_2^{\text{in}} = E(2, \rho, \mathbf{u}) + (f_6^{\text{in}} - E(6, \rho, \mathbf{u}))$$



# Condition d'entrée: code

- Définition des indices pour les trois colonnes:

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

- Calcul de la densité:

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

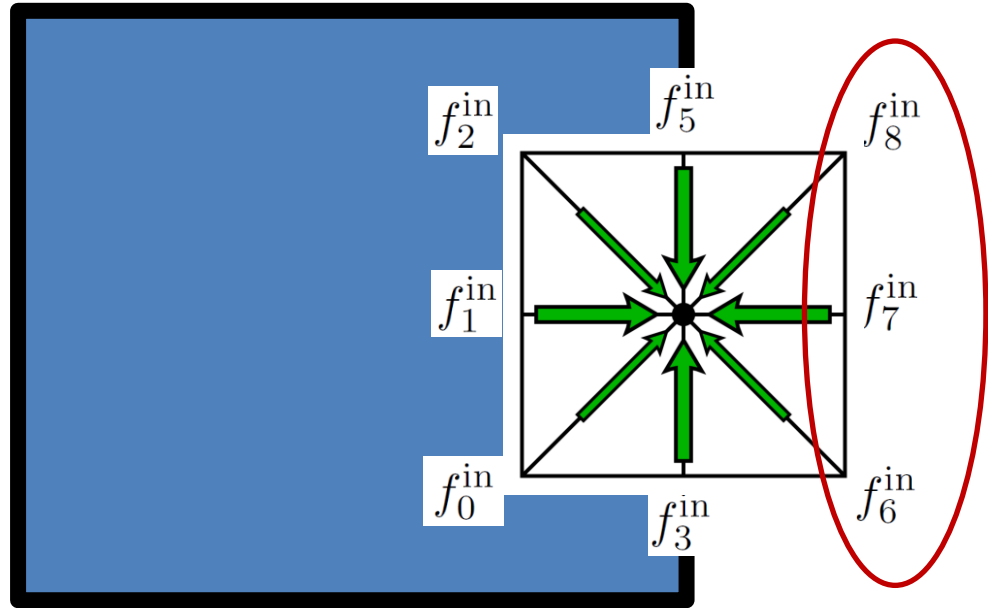
- Calcul des populations:

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

# Condition de sortie

- La condition de sortie doit se comporter comme si le domaine ne s'arrêtait pas.
- Pour les populations inconnues,  $f_6$ ,  $f_7$ , et  $f_8$ , nous copions simplement la valeur de la cellule voisine qui se trouve droit derrière.

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



# Application: écoulement autour d'un obstacle

# Notre code: synthèse des morceaux

```
#!/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.
nub = uLB*r/Re; # Viscosity in lattice units.
omega = 1 / (3*nub+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,:,i] += v[i,0] * fin[i,:,i]
        u[1,:,i] += v[i,1] * fin[i,:,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,:,i] + v[i,1]*u[1,:,i])
        feq[i,:,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,:] * ( 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,:,i] = roll(
            roll(fout[i,:,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{:03d}.png".format(time//100))
```



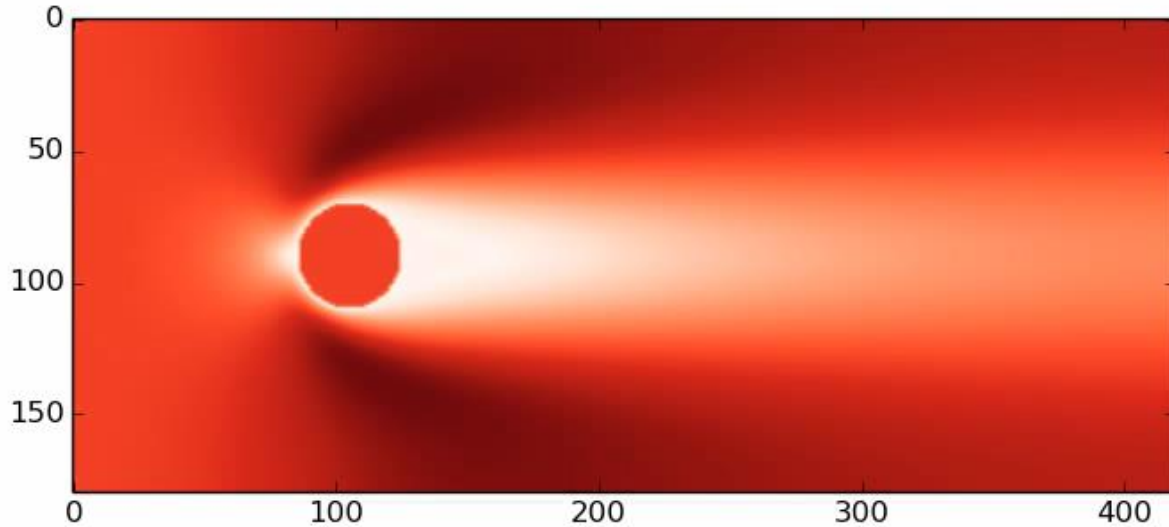
UNIVERSITÉ  
DE GENÈVE

CENTRE UNIVERSITAIRE  
D'INFORMATIQUE

# Le début: paramètres de la simulation

```
nx, ny = 420, 180 # Nombre de cellules  
r = ny//9        # Rayon de l'obstacle  
uLB = 0.04        # Vitesse à l'entrée  
Re = 220.0        # Nombre de Reynolds  
nulb = uLB*r/Re  # Viscosité  
omega = 1 / (3*nulb+0.5) # Param. de relaxation
```

# Résultat: $Re=10$





# Résultat: $Re=220$

