# Melting_Ga Documentation

Suemi Rodríguez Romo*, Benjamín Salomón Noyola García

May 18, 2018

# Contents

**Release** 2018.1

**Date** March 08, 2018

Melting_Ga is a free computational melting Gallium embedded in a porous medium simulator based on multi-relaxation time Lattice Boltzmann method (MRLBM). It is implemented in Python-Numba and optimized for modern multi-core systems, especially GPUs (Graphics Processing Units).

Want to see Melting_Ga in action? Check our videos on You Tube , or if you prefer, get the code and see by yourself the examples provided here.

# 1 Contents

## 1.1 Motivation and Design Principles

Melting_Ga is a software to calculate the solid-liquid phase change of Gallium embedded in a porous media, designed for modern multi-core processors, especially Graphics Processing Units (GPUs) and implemented in Python-Numba. The solver is based in the MRLBM, which is quite simple to implement and understand. Furthermore, this software scales very well with increasing computational processing units.

The goals of the project are as follow:

- **Performance:** the most complex (time consuming) steeps are massively parallelized (collision and calculation of macroscopic variables) and performed in local memory, which is the fastest memory in GPUs. The remaining steeps are performed in shared memory.
- **Scalability:** it is possible to scale with increasing number of processing units.
- **Maintainability:** The code is clean and easy to understand. There are parameters that can be modified to address other cases; for instance; the card architecture.
- **Ease of use:** The code is easy to use, many steps are automatized only by installing Anaconda Cloud which is fundamental to use this codes.

## 1.2 Supported features and models

Table 1: Supported features and models

| **Feature type** | **Supported variants** |
|---|---|
| Stencils | D2Q5, D2Q9 |
| Lattice Boltzmann external forces | Bouyancy, porous media 1. |
| Relaxation dynamics | MRT |
| Porosity | Locally homogeneous and globally heterogeneous |
| Fields | Momentum flow and temperature |
| Image | Binary image |
| Computational backends | CUDA-Python-Numba |
| Output formats | Numpy, Matplotlib, plain text |

The total Lattice Boltzmann external force F, stands for the presence of a porous medium and other forces like gravitation in the fluid dynamics. In our instance this is given by:

$$\mathbf{F} = -\frac{\varphi \nu_l}{K}\mathbf{u} - \frac{\varphi C_F}{\sqrt{K}}|\mathbf{u}|\mathbf{u} + \varphi \mathbf{G}, \tag{1}$$

where $\mathbf{u}$ is the fluid velocity, $\nu_l$ is the viscosity of the fluid, $K = \frac{\varphi^3 d_m^2}{175(1-\varphi)^2}$ (here $d_m$ is the mean diameter of the solid particle) is the permeability (Kozeny law is used locally), $\varphi$ is the porosity of the medium and $C_F$ is the inertial coefficient calculated as $C_F = \frac{1.75}{\sqrt{175\varphi^3}}$. Based on the Boussinesq approximation, the buoyancy force $\mathbf{G}$ is given by [[1], [2]];

$$\mathbf{G} = g\beta\left(T - T_0\right)\mathbf{j}, \tag{2}$$

where $g$ is the gravitational acceleration and $\beta$ is the thermal expansion coefficient.

### 1.2.1 Boundary conditions.

We are dealing with two cases to study different physical situations. In Case 1 the simulation from a solid-liquid phase change to a solid material (which can be Ga) as a beam, caused by heat transfer, without porous media, is performed. In Case 2 the simulation of the solid-liquid phase change of a material (here Ga is used) immersed in a 2D porous media is executed.

*We implemented different boundary conditions in the following situations;*

- **Periodic boundary condition** (on the north and south of the model geometry) for the simulation of temperature field in case 1).
- **Dirichlet boundary condition** (on the east and west of the model geometry) for the simulation of temperature field in cases 1) and 2).
- **Adiabatic boundary condition** (on the north and south of the model geometry) for the simulation of temperature field in case 2).
- **Bounce-back boundary condition** (on north, south, east and west of the model geometry) for the simulation of velocity field in case 2)

These boundary conditions can be better explained in [3, 4].

## 1.3 Installation

In order to execute the codes, it is necessary to have a Nvidia video card, Melting_Ga requires no installation and all sample simulation provided can be executed via spyder IDE, provided the required packages are installed in the host system:

General requirements:

- Numba
- Numba-CUDA
- Numpy
- Sympy
- Matplotlib
- Scipy
- Spyder IDE

All these libraries are obtained when installing Anaconda https://www.anaconda.com/download/#linux (Python 2.7 version).

### 1.3.1 Download Melting_Ga

You could Download the codes from git repository: https://github.com/BenjaminNoyola/Numerical-methods

### 1.3.2 Windows installation instructions

In order to execute melting_Ga codes, the following requirements must be satisfied:

1. download Anaconda 5.1: https://www.anaconda.com/download/#windows: (Python 2.7 version)
2. install Anaconda through Anaconda prompt
3. Update Anaconda: *conda update conda*
4. Update Numba: *conda update numba*
5. Install CUDA: *conda install cudatoolkit*

After installing Anaconda (Version 5.1|release Date: February 15, 2018), you will be able to execute the samples and all melting_Ga codes. But if you find the most recent checkout to be somehow broken, you might want to rewind to one of the tagged releases.

### 1.3.3 Ubuntu and Mac OS X installation instructions

Ubuntu and Mac OS X installation are similar to windows case,

1. download Anaconda: https://www.anaconda.com/download/#linux: (Python 2.7 version)
2. install Anaconda through command prompt
3. Update Anaconda: $ *conda update conda*
4. Update Numba: $ *conda update numba*
5. Install CUDA: $ *conda install cudatoolkit*

## 1.4 Tutorial

### 1.4.1 Running simulations

In this section, we show how to create a simple (MRLBM) simulation using Melting_Ga. You can run the main program (main.py) and choose one simulation to execute. The program shows graphics. We focus on two types of simulations:

1. **Case 1. Phase change simulation of a simple bar of a solid material (caused by thermal phenomena). No porous medium is involved. The main features of this simulation are:**
   - **Serial implementation performed only by Numpy.**
   - **Serial implementation performed by Numba with Numpy input.**
   - **Parallel implementation performed by Numba-CUDA, streaming step, boundary conditions and calculation of macroscopic variables are performed in shared memory.**
   - **Parallel implementation performed by Numba-CUDA of the collision step and the calculation of macroscopic variables are performed in local memory.**
   - **Implementation of the analytical solution which is compared with the numerical solution.**
2. **Case 2. Solid-liquid phase change simulation of a material (Ga) immersed in a porous media, performed by Numba-CUDA. This simulation uses MRLBM, with a D2Q5 stencil, for the heat transfer in Ga and a D2Q9 stencil for the momentum transfers in the liquid Ga. The last one takes into account natural convection. The main features of the simulation are the following;**
   - **simulation with global homogeneous porosity ($\phi = 0.368$)**
   - **simulation with global heterogeneous porous media by using the image of an actual packed media, see Fig. 6). This image is treated such that we can work at different scales. In each scale we split the total image in a set of images with a predetermined resolution and calculate the porosity as locally homogeneous. This is repeated the necessary number of times to reproduce the whole image. Therefore the porosity is globally heterogeneous and locally homogeneous. The definition of locality is given by the resolution and can be set by hand.**

It is worth mentioning that we have split the code to explain the way it was designed and some features that we consider important. In this partition we do not keep well known coding structures as a whole, for instance cycles can be initiated in one partition and finalized in the next one. In spite of this fact everything in the code is clear and functional without the partition.

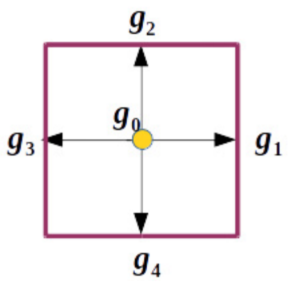


Figure 1: D2Q5 unit stencil

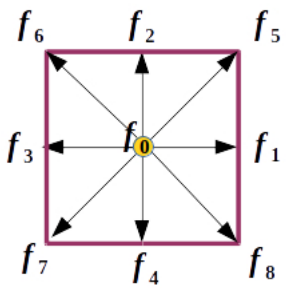


Figure 2: D2Q9 unit stencil

### 1.4.2 Case 1

Unidimensional (the phase change of a bar is modeled in 1D) numerical solutions are performed by MRLB. We use a lattice of $1024 \times 8$, and double time relaxation LBM for the solid-liquid phase change with natural convection in porous media [5].

**Code 1: Serial implementation.**

The following code corresponds to the serial implementation performed by Numba with Numpy inputs. In section A we import the libraries, then we introduce parameters for the simulation. All these parameters could change if desired, for example, the size of this simulation is $1024 \times 8$ (lattice_x=1024, lattice_y=8) but could be different.

```
                        #_________ Section A _________#
from numpy import *
import matplotlib.pyplot as plt
from numba import jit

#...Parameters...#
rho = 1.0               # density
Delta_alfa=10.0         # thermal diffusivity ratio
T_i =-1.0               # low temperature
T_b = 1.0               # high temperature
T_m = 0.0               # melting temperature
alpha_s = 0.002         # thermal diffusivity of solid
alpha_l = Delta_alfa*alpha_s  # thermal diffusivity of liquid
poros = 1.0             # porosity
sigma = 1.0             # thermal capacity ratio
Cpl=Cps = 1.0           # specific heat
w_test = -2.0           # constant of D2Q5 MRT-LB model
k_s = alpha_s*rho*Cps # solid thermal conductivity
k_l = alpha_l*rho*Cpl # liquid thermal conductivity
St = 1.0                # Stefan number
F_0 = 0.01              # Fourier number
H = 200.0               # characteristic length
La = Cpl*(T_b-T_m)/St # latent heat
H_l = Cpl*0.02 + 1.0*La      # Enthalpy of liquid
H_s = Cps*(-0.02) + 0.0*La   # Enthalpy of solid  Ts=-0.02
t = (F_0*H**2)/alpha_s       # time
```

```
delta_x = delta_t=1.0          # time and space step
c = delta_x/delta_t            # lattice speed
c_st = sqrt(1.0/5.0)           # sound speed of the D2Q9 model
lattice_x = 1024     # lattices in x direction; edit this line to change the size
lattice_y = 8        # lattices in y direction; edit this line to change the size
pasos = 300000       # EDIT THIS LINE to change the number of time steps
```

In section B, D2Q5 is used, as shown in Figure ??. We initialize arrays that are going to be filled and used for calculating many variables; for example the temperature $T[lattice_y, lattice_x]$ is initialized in the first line of this section as a matrix with size $lattice_y, lattice_x$ and will change as the code runs in the computer.

```
                    #__________ Section B __________#
T    = ones([lattice_y,lattice_x]) # temperature is saved as a matrix
g = zeros([5,lattice_y,lattice_x]) # distribution function is saved as a tensor
    order 3
g_eq = zeros([5,lattice_y,lattice_x])  # equilibrium distribution function is saved
    .
n = zeros([5,lattice_y,lattice_x])     # distribution function in moment space
n_eq = zeros([5,lattice_y,lattice_x])  # equilibrium distribution function in
    moment space
n_res1 = zeros([5])                    # distribution function after collision
n_res2 = zeros([5])                    # distribution function after collision
H_k  = zeros([lattice_y,lattice_x])    # enthalpy is saved in a matrix
f_l  = zeros([lattice_y,lattice_x])    # liquid fraction is saved in a matrix
t_relax_T_ad = zeros([lattice_y,lattice_x]) # dimensionless relaxing time of
    temperature field
relax_o = zeros([lattice_y,lattice_x])
alpha_e = zeros([lattice_y,lattice_x]) # thermal diffusivity
tau_t = zeros([lattice_y,lattice_x])   # relaxation parameters
k_e = zeros([lattice_y,lattice_x])     # thermal conductivity
w_s = zeros([5])                       # weight coefficients
s_source = zeros([5])                  # source vector
N=array([[1.0, 1.0, 1.0, 1.0,1.0],[0.0, 1.0, 0.0, -1.0, 0.0],[0.0, 0.0, 1.0, 0.0,
    -1.0],\
[-4.0, 1.0, 1.0, 1.0,1.0],[0.0, 1.0, -1.0, 1.0, -1.0]])   # matrix transformation
N_inv= linalg.inv(N)                   # matrix inverse

for i in range(5):                     # weight coefficients of the D2Q9 stencil
    used to calculate the temperature field:
  if i == 0:
    w_s[i] = (1-w_test)/5.
  else:
    w_s[i] = (4+w_test)/20.
```

At the beginning of section C, temperature $T_{i,j}$ and liquid fraction $f_l$ in the boundaries are initialized, then the enthalpy $H\_k_{i,j}$ and the liquid fraction $fl_{i,j}$ are initialized. At the end of this section, the distribution function $(g_{k,i,j})$ is obtained by using $(T_{i,j})$.

```
                    #__________ Section C __________#
T=T_i*T                       # initial conditions
for i in range (lattice_y):
    T[i][0]=T_b               # high temperature in boundary
    T[i][lattice_x-1] = T_i # low temperature in boundary
    f_l[i][0]=1.0             # liquid fraction
    f_l[i][lattice_x-1] =0.0
f_2l=copy(f_l)

for i in range(lattice_y):  # calculation of enthalpy
    for j in range(lattice_x):
      H_k[i][j] = Cps*T[i][j] + f_l[i][j]*La

for i in range(lattice_y):  # initial distribution function
  for j in range(lattice_x):
    for k in range(5):
      g[k][i][j] = w_s[k]*T[i][j]
```

Section D is performed by numba, compiling just in time (@jit). This is a function that receives the number of time steps and the distribution function $(g_{k,i,j})$. $g$ is a third order tensor, the indexes $i,j$ represent the size of the domain in 2D and the index $k$ represents the discrete speeds per lattice.

This is used in order to calculate the temperature field, $T_{i,j} = \sum_{k=0}^{4} g_{k,i,j}$ that depends of space and time.

To update the liquid fraction, first the enthalpy is calculated as $H\_k_{i,j} = Cps * T_{i,j} + fl_{i,j} * La$ (part of the code above), then the liquid fraction (part of the code below) can be obtained by the following equation

$$fl = \begin{cases} 0, & \text{if } H_k \leq H_s, \\ \frac{H_k - H_s}{H_l - H_s}, & \text{if } H_s < H_k < H_l, \\ 1, & \text{if } H_k \geq H_l, \end{cases} \quad (3)$$

where $fl$ is the liquid fraction, $H_s$ is the solid phase enthalpy and $H_l$ is the liquid phase enthalpy. Recall that Ga is undergoing a phase change in two phases embedded in a porous medium.

Finally in section D, a linear transformation to the moment space is performed as $\mathbf{n} = \mathbf{N}\mathbf{g}$ [6]. Here

$$\mathbf{N} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \\ -4 & 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & 1 & -1 \end{pmatrix} \quad (4)$$

```
                    #__________ Section D __________#
@jit  # compile just in time
def MRT_LB(pasos,g):
  for kk in range(pasos):

    T = g.sum(axis=0) # temperature calculation

    f_2l=copy(f_l) # copy the liquid fraction
    for i in xrange(lattice_y): # enthalpy calculation
      for j in xrange(lattice_x):
        H_k[i][j] = Cps*T[i][j] + f_l[i][j]*La
        if (H_k[i][j] <= H_s):
          f_l[i][j]=0.0
        elif (H_k[i][j] > H_s and H_k[i][j] < H_l):
          f_l[i][j] = (H_k[i][j] - H_s)/(H_l - H_s)
        else:
          f_l[i][j]=1.0
    n = tensordot(N, g, axes=([1],[0])) # transformation from velocity space to
    moment space
```

In section E, the collision step is performed in moment space.

$$\mathbf{n}^{+}(\mathbf{x},t) = \mathbf{n}(\mathbf{x},t) - \mathbf{\Theta}\left[\mathbf{n}(\mathbf{x},t) - \mathbf{n}^{(eq)}(\mathbf{x},t)\right] + \delta_t \tilde{\mathbf{S}} \quad (5)$$

Previously, it's necessary to calculate, $\mathbf{\Theta}$, $\mathbf{n}^{(eq)}$, $\tilde{\mathbf{S}}$:

$$\mathbf{\Theta} = diag(1, 1/\tau_T, 1/\tau_T, 1.5, 1.5)$$

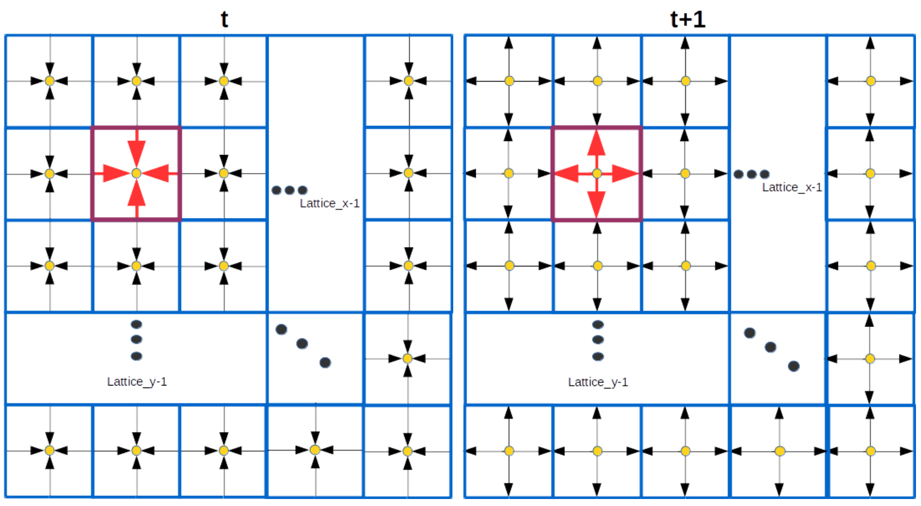
$$g_i^{(eq)} = \tilde{\omega}_i T \left(1 + \frac{\mathbf{e_i} \cdot \mathbf{u}}{\sigma c_{sT}^2}\right),$$

The corresponding equilibrium moments $\{n_i^{(eq)} | i = 0, 1, ..., 5|\}$ in momentary space obtained by transforming the temperature equilibrium distribution function $g_i$ are as follows

$$\mathbf{n}^{(eq)} = \begin{pmatrix} T \\ u_x T/\sigma \\ u_y T/\sigma \\ \tilde{\omega} T \\ 0 \end{pmatrix} \quad (6)$$

Recall that the unitary stencil for D2Q5 is:

$$e_i = \begin{cases} (0,0) & i = 0 \\ (cos[(i-1)\pi/2], sin[(i-1)\pi/2])\, c & i = 1,2,3,4 \end{cases} \quad (7)$$



(a) Collision step in $t$ (b) Collision step in $t+1$

Figure 3: Collision step of temperature field

with the following weight coefficients; $\tilde{\omega}_0 = 4/6$, $\tilde{\omega}_{1,2,3,4} = 1/12$.

At the end of this section, distribution function in moment space is transformed to velocity space:

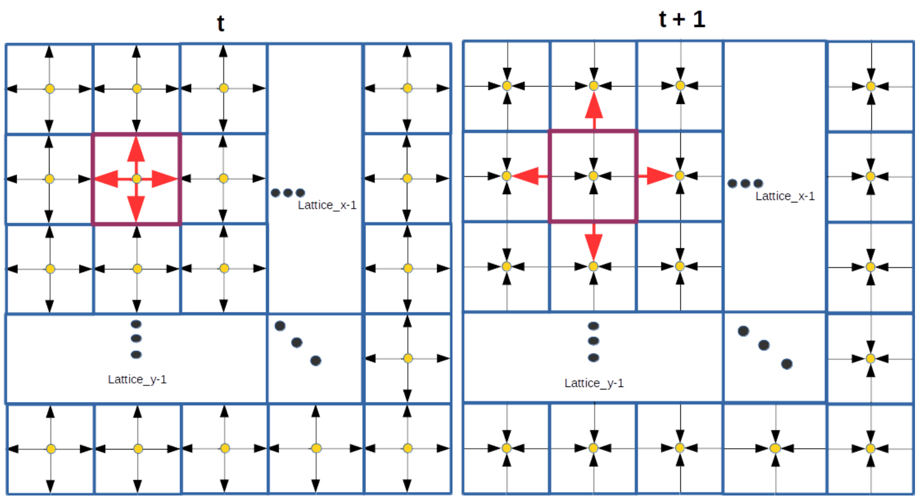
$$\mathbf{g} = \mathbf{N}^{-1}\mathbf{n}$$

In collision step, the operations are carried out as shown in Figure 3

```
                    #__________ Section E __________#
    for i in xrange(lattice_y):
      for j in xrange(lattice_x):
        alpha_e[i][j] = alpha_l*f_l[i][j] + alpha_s*(1.0-f_l[i][j])
        tau_t[i][j] = 0.5 + alpha_e[i][j]/(sigma*c_st**2.0*delta_t)

    for i in xrange(lattice_y):
      for j in xrange(lattice_x):
        relax_o = array([[1.0, 0.0, 0.0, 0.0, 0.0],[0.0, 1.0/(tau_t[i][j]),\
                0.0, 0.0, 0.0],[0.0, 0.0, 1.0/(tau_t[i][j]), 0.0, 0.0],[0.0, 0.0,\
                0.0, 1.5, 0.0],[0.0, 0.0, 0.0, 0.0, 1.5]])
        s_surce = array([-((poros*La)/(sigma*Cpl))*(f_l[i][j]-f_2l[i][j])/1.0,\
                0.0 , 0.0 , -w_test*((poros*La)/(sigma*Cpl))*(f_l[i][j]-\
                f_2l[i][j])/1.0, 0.0])
        n_eq = array([T[i][j] , 0.0 , 0.0 , w_test*T[i][j] , 0.0]) # equilibrium
    distribution function

                for k in xrange(5):
          n_res1[k] = n[k][i][j] - n_eq[k]
        n_res2=dot(relax_o,n_res1)
        for k in xrange(5):
          n[k][i][j] = n[k][i][j] - n_res2[k] + s_surce[k]  # collision step.

    g = tensordot(N_inv, n, axes=([1],[0])) #transformation from moment space to
    velocity space
```

In section F streaming step is performed in velocity space, the operations are carried out as shown in Figure 4

```
                    #__________ Section F __________#
    for i in xrange (lattice_y):
      for j in xrange (lattice_x-1,0,-1): # horizontal
        g[1][i][j] = g[1][i][j-1]         # vector 1
      for j in xrange (lattice_x-1):
        g[3][i][j] = g[3][i][j+1]         # vector 3

    for i in xrange (lattice_y-1):         # vertical
      for j in xrange (lattice_x):
```



(a) Streaming step in $t$ (b) Steaming step in $t + 1$

Figure 4: Streaming step of temperature field

```
        g[2][i][j] = g[2][i+1][j]          # vector 2

    for i in xrange (lattice_y-1,0,-1):
      for j in xrange (lattice_x):
        g[4][i][j] = g[4][i-1][j]          # vector 4
```

In section G, boundary conditions step are performed in velocity space. Dirichlet boundary conditions are implemented for vertical walls and periodic boundary conditions are implemented for horizontal walls. Boundary conditions are represented by the figure 5, it is important to mention that our domain is $1024 \times 8$.

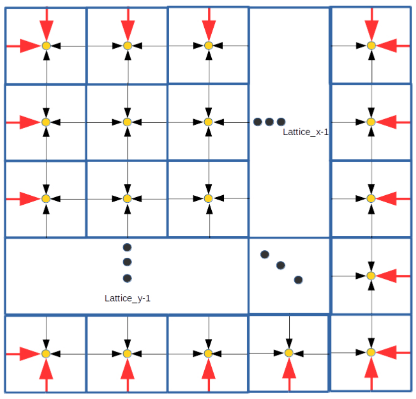


Figure 5: Boundary condition step

```
                    #__________ Section G __________#
    for i in xrange(lattice_y): # Dirichlet boundary condition (vertical)
      g[1][i][0] = w_s[1]*T_b + w_s[3]*T_b - g[3][i][0]
      g[3][i][lattice_x-1] = w_s[1]*T_i + w_s[3]*T_i - g[1][i][lattice_x-1]

    for j in xrange(lattice_x): # periodic boundary conditions (horizontal)
      g[2][lattice_y-1][j] = g[2][0][j]
      g[4][0][j] = g[4][lattice_y-1][j]

  return g
```

In section H, we measure the time and MLUPS (Million Lattice updates Per Second), then save solutions in plain text format.

```
                    #__________ Section H __________#
import time
tiempo_numba = time.time()      # measure computing time at the beginning
gg=MRT_LB(pasos,g)
T = gg.sum(axis=0)
t = time.time() - tiempo_numba  # measure computing time
print"\n", t, lattice_x*lattice_y*pasos/t/1e6 # report MLUPS
print "T= ",T
Perfo=[t, lattice_x*lattice_y*pasos/t/1e6]
import numpy as np
np.savetxt('Perfo_1024X8_300mil.txt',Perfo)        # save performance
np.savetxt('T_1024X8_300mil.txt', T,fmt='%.6f')    # save temperature
np.savetxt('f_l_1024X8_300mil.txt', f_l,fmt='%.3f')# save liquid fraction
```

**Code 2: Parallel implementation**

The following code delivers the same solution that the last code, but this corresponds to parallel implementation performed by Numba-CUDA, collision step and calculation of macroscopic variables are performed in local memory, the rest of variables are performed in shared memory. This program is distributed in **3 different codes**. The first is the main program, the second code coordinates operations between different types of graphic card memories, controls the operations in shared memory and finally call the third one that controls the local memory operations.

In section A, we import Numpy, Numba, Matplotlib and MRT_LB_shared libraries. A Python function is implemented to calculate The distribution function $(g_{i,j,k})$ which is initially obtained using the prescribed temperature $(T_{i,j})$.

**Code 2.1. Main program.**

```
                    #__________ Section A __________#
import numpy as np
from numba import jit, cuda
import matplotlib.pyplot as plt
import MRT_LB_shared as d2q5

@jit
def init_sol(g, w_s, T):    # initial solution of g
    for i in xrange(lattice_x):
          for j in xrange(lattice_y):
            for k in xrange(5):
              g[i][j][k] = w_s[k]*T[i][j]
```

The main program begins in section B. First we define the parameters of the simulation. At the end, you can edit the time steps simulation.

```
                    #__________ Section B __________#
if __name__ == '__main__':
    rho = 1.0                 # density
    Delta_alfa=10.0           # thermal diffusivity ratio
    T_i = -1.0                # low temperature
    T_b = 1.0                 # high temperature
    T_m = 0.0                 # melting temperature
    alpha_s = 0.002                   # thermal diffusivity of solid
    alpha_l = Delta_alfa*alpha_s      # thermal diffusivity of liquid
    poros = 1.0               # porosity
    sigma = 1.0               # thermal capacity ratio
    Cpl=Cps = 1.0             # specific heat
    w_test = -2.0             # constant of D2Q5 MRT-LB model
    k_s = alpha_s*rho*Cps     # solid thermal conductivity
    k_l = alpha_l*rho*Cpl     # liquid thermal conductivity
    St = 1.0                  # Stefan number
    F_0 = 0.01                # Fourier number
    H = 200.0                 # characteristic leght, this is redefined later.
    La = Cpl*(T_b-T_m)/St     # latent heat
    H_l = Cpl*0.02 + 1.0*La # enthalpy of liquid
    H_s = Cps*(-0.02) + 0.0*La    # Enthalpy of solid
    t = (F_0*H**2)/alpha_s  # time
    delta_x = delta_t=1.0   # time and space step
    c = delta_x/delta_t     # lattice speed
    c_st = np.sqrt(1.0/5.0) # sound speed of the D2Q5 model
    lattice_x = 256    # lattices in x direction; edit this line to change the size
    lattice_y = 8      # lattices in y direction; edit this line to change the size
    pasos=300000       # EDIT THIS LINE to change the number of time steps
```

The initialization list can be found in section C

```
                #__________ Section C __________#
    T = np.ones([lattice_x,lattice_y])        # temperature is saved in a matrix
    g = np.zeros([lattice_x,lattice_y,5])     # distribution function as a order 3
    tensor
    h_copia_g = np.zeros([lattice_x,lattice_y,5]) # take a copy of g to be sent to
    device
    f_l  = np.zeros([lattice_x,lattice_y])    # liquid fraction is saved in a matrix
    w_s = np.zeros([5])                       # weight coefficients
    w_s[0] = (1.0-w_test)/5.0
    for i in xrange(1,5,1):
        w_s[i] = (4.0+w_test)/20.
```

The parameters of CUDA; threads and blocks per grid are defined in Section D. Note that in the second row of this section you should edit to be able to introduce the architecture of the video card used by you.

```
                #__________ Section D __________#
    nx, ny, ns  = lattice_x, lattice_y, 5
    threads = 256,4   # EDIT this line, to change the number of threads
    blocks = (nx/threads[0]+(0!=nx%threads[0]),
              ny/threads[1]+(0!=ny%threads[1]))
```

In section E we send g, T, fl, Delta_alfa, w_copia_g and w_s lists from the host to the device.

```
                #__________ Section E __________#
    T=T_i*T                                  # initial temperature
    T[0,:], T[lattice_x-1,:] = T_b, T_i      # initial temperature in boundaries
    f_l[0,:], f_l[lattice_x-1,:] = 1.0, 0.0  # initial liquid fraction
    init_sol(g, w_s, T)                      # Initial solution (g)

    T_b, T_i, D_al = np.array([float(T_b)]), np.array([float(T_i)]),\
    np.array([float(Delta_alfa)])
    d_g = cuda.to_device(g)       # send g from host to device
    copia_g = cuda.to_device(h_copia_g) # send a copy of g from host to device
    d_ws = cuda.to_device(w_s)  # send w_s from host to device
    D_al = cuda.to_device(D_al) # send thermal diffusivity ratio from host to
    device
    d_fl = cuda.to_device(f_l)  # send liquid fraction from host to device
```

The main CUDA loop is introduced in Section F which connects with MRT_LB_shared, identified by D2Q5. Lists sent to device are operated in the kernel. After completed the time steps, lists are sent to device again.

```
                #__________ Section F __________#
  import time
    tiempo_cuda = time.time()   # Measure initial time, before the main CUDA loop
    begins
    for ii in xrange(pasos):
        d2q5.collision_local[blocks, threads](d_g, d_fl, D_al)  # collision step
        d2q5.propagacion[blocks, threads](d_g,copia_g)       # streaming step
        d2q5.condiciones_frontera[blocks, threads](d_g, d_ws)    # boundary
    conditions
    d_g.to_host()        # send g to host
    d_fl.to_host()       # send liquid fraction (f_l) to host
    T = g.sum(axis=2)    # calculate temperature from the distribution function
```

In section G the solution is exported to plain text format, and computing time is measured.

```
                    #__________ Section G __________#
    t = time.time() - tiempo_cuda   # measure final time, after the main CUDA loop
    ends
    print"\n", t, nx*ny*pasos/t/1e6 # print time and MLUPS
    Perfo=[t, nx*ny*pasos/t/1e6]  # simulation time, MLUPS
    T=np.transpose(T)
    f_l=np.transpose(f_l)
    print "\n Temperature: \n",T    # print temperature as a matrix, every cell
    corresponds to a lattice
    print "\n Liquid fraction \n",f_l  # print Liquid fraction as a matrix, every
    cell corresponds to a lattice
    np.savetxt('Perfo_256X8_300mil.txt',Perfo) # save performance in txt
    np.savetxt('T_256X8_300mil.txt', T,fmt='%.6f') # save temperature in txt
    np.savetxt('f_l_256X8_300mil.txt', f_l,fmt='%.3f') # save liquid fraction
```

**Code 2.2. Coordination of operations between different types of memories. Control the operations in shared memory. Call the third code that controls the local memory operations.**

Now this code (MRT_LB_shared) called by the last code, coordinates the operation between local and shared memory. This code calls to MRT_LB_local as D2Q5, in charge to perform local operations. In section A, the libraries are called, then we define the CUDA device functions. All this functions will be executed in local memory and they involve double-precision floating-point number (f8) [7].

```
                    #__________ Section A __________#
import MRT_LB_local as d2q5
from numba import cuda, float64, float32
import numpy as np
import matplotlib.pyplot as plt
# CUDA device functions
getg = cuda.jit('void (f8[:,:,:], f8[::1], i8, i8)', device=True)(d2q5.getg)
getfl = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(d2q5.getfl)
calc_T = cuda.jit('void (f8[::1], f8[:1])', device=True)(d2q5.calc_T)
calc_copiafl = cuda.jit('void (f8[:1],f8[:1])', device=True)(d2q5.calc_copiafl)
calc_Hk = cuda.jit('void (f8[:1], f8[:1], f8[:1])', device=True)(d2q5.calc_Hk)
calc_fl = cuda.jit('void (f8[:1], f8[:1])', device=True)(d2q5.calc_fl)
calc_g2n = cuda.jit('void (f8[::1], f8[::1])', device=True)(d2q5.calc_g2n)

calc_alfe = cuda.jit('void (f8[:1], f8[:1], f8[1])', device=True)(d2q5.calc_alfe)
calc_taut = cuda.jit('void (f8[:1], f8[:1])', device=True)(d2q5.calc_taut)
calc_relax = cuda.jit('void (f8[::1], f8[:1])', device=True)(d2q5.calc_relax)
calc_Ssurce = cuda.jit('void (f8[::1], f8[:1], f8[:1])', device=True)(d2q5.
    calc_Ssurce)
n_eq_loc = cuda.jit('void (f8[::1], f8[:1])', device=True)(d2q5.n_eq_loc)
calc_colision = cuda.jit('void (f8[::1], f8[::1], f8[::1], f8[::1])', device=True)(
    d2q5.calc_colision)
n2g_loc = cuda.jit('void (f8[::1], f8[::1])', device=True)(d2q5.n2g_loc)

setfl = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(d2q5.setfl)
setg = cuda.jit('void (f8[:,:,:], f8[::1], i8, i8)', device=True)(d2q5.setg)
set_prueba = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(d2q5.
    set_prueba)
```

Section B of this code is the streaming step, we create a copy of the distribution function, then we propagate the distribution function.

```
                    #__________ Section B __________#
@cuda.jit('void(f8[:,:,:],f8[:,:,:])')
def propagacion(d_g, copia_g):
    nx, ny, ns = d_g.shape
    i, j = cuda.grid(2)

    copia_g[i,j,1] = d_g[i,j,1]
    copia_g[i,j,2] = d_g[i,j,2]
    copia_g[i,j,3] = d_g[i,j,3]
    copia_g[i,j,4] = d_g[i,j,4]

# streaming step inside the domain without the perimeter #
    if i>0 and i<nx-1 and j>0 and j<ny-1:
        d_g[i,j,1] = copia_g[i-1,j,1]
        d_g[i,j,2] = copia_g[i,j+1,2]
        d_g[i,j,3] = copia_g[i+1,j,3]
        d_g[i,j,4] = copia_g[i,j-1,4]

# streaming step in the perimeter without the corners #
    if j>0 and j<ny-1:
        d_g[nx-1,j,1] = copia_g[nx-2,j,1]
        d_g[nx-1,j,2] = copia_g[nx-1,j+1,2]
        d_g[nx-1,j,4] = copia_g[nx-1,j-1,4]

    if i>0 and i<nx-1:
        d_g[i,0,2] = copia_g[i,1,2]
        d_g[i,0,1] = copia_g[i-1,0,1]
        d_g[i,0,3] = copia_g[i+1,0,3]

    if j>0 and j<ny-1:
        d_g[0,j,3] = copia_g[1,j,3]
        d_g[0,j,2] = copia_g[0,j+1,2]
        d_g[0,j,4] = copia_g[0,j-1,4]

    if i>0 and i<nx-1:
        d_g[i,ny-1,4] = copia_g[i,ny-2,4]
        d_g[i,ny-1,1] = copia_g[i-1,ny-1,1]
        d_g[i,ny-1,3] = copia_g[i+1,ny-1,3]

# streaming step in the corners #
    d_g[nx-1,0,1] = copia_g[nx-2,0,1]
    d_g[nx-1,0,2] = copia_g[nx-1,1,2]

    d_g[0,0,2] = copia_g[0,1,2]
    d_g[0,0,3] = copia_g[1,0,3]

    d_g[0,ny-1,3] = copia_g[1,ny-1,3]
    d_g[0,ny-1,4] = copia_g[0,ny-2,4]

    d_g[nx-1,ny-1,1] = copia_g[nx-2,ny-1,1]
    d_g[nx-1,ny-1,4] = copia_g[nx-1,ny-2,4]
```

Recall that we use the following boundary condition; Dirichlet boundary condition for east and west walls and periodic boundary condition for north and south walls. See Section C below.

```
                    #_________ Section C _________#
@cuda.jit('void(f8[:,:,:], f8[:])')
def Boundary_conditions(d_g, d_ws):
    nx, ny, ns = d_g.shape
    d_Tb = 1.0
    d_Ti = -1.0

    for i in xrange(ny): # Dirichlet boundary conditions
        d_g[0,i,1] = d_ws[1]*d_Tb + d_ws[3]*d_Tb - d_g[0,i,3]
        d_g[nx-1,i,3] = d_ws[1]*d_Ti + d_ws[3]*d_Ti - d_g[nx-1,i,1]

    for j in xrange(nx): # periodic boundary conditions
        d_g[j,ny-1,2] = d_g[j,0,2]
        d_g[j,0,4] = d_g[j,ny-1,4]
```

Collision step and determination of macroscopic variables. These operations are performed in local memory. First we define CUDA device functions, then we call (MRT_LB_local) as D2Q5 to perform operations.

```
                    #_________ Section D _________#
@cuda.jit('void(f8[:,:,:], f8[:,:], f8[:])')
def collision_local(d_g, d_fl, D_al):
    nx, ny, ns = d_g.shape # obtain sizes from the shape of d_g
    gloc = cuda.local.array(5, dtype=float64)
    nloc = cuda.local.array(5, dtype=float64)
    neqloc = cuda.local.array(5, dtype=float64)
    Tloc = cuda.local.array(1, dtype=float64)
    Hkloc = cuda.local.array(1, dtype=float64)
    flloc = cuda.local.array(1, dtype=float64)
    copiaflloc = cuda.local.array(1, dtype=float64)
    alfeloc= cuda.local.array(1, dtype=float64)
    tautloc = cuda.local.array(1, dtype=float64)
    relaxloc= cuda.local.array(5, dtype=float64)
    Ssurceloc=cuda.local.array(5, dtype=float64)

    i, j = cuda.grid(2)             # sizes of grid

    getg(d_g, gloc, i, j)           # obtain g in local memory from shared memory
    getfl(d_fl, flloc, i ,j)        # obtain liquid fraction
    calc_T(gloc, Tloc)              # temperature is calculated
    calc_copiafl(copiaflloc, flloc) # take a copy of liquid fraction
    calc_Hk(Tloc, Hkloc, flloc)     # calculation of enthalpy
    calc_fl(flloc, Hkloc) # calculation of liquid fraction
    calc_g2n(nloc, gloc)  # linear transformation from velocity space to moment
space
    calc_alfe(alfeloc, flloc, D_al) # calculation of thermal diffusivity
    calc_taut(tautloc, alfeloc)     # relaxation parameter
    calc_relax(relaxloc, tautloc)   # multiple relaxation vector
    calc_Ssurce(Ssurceloc, flloc, copiaflloc)
    n_eq_loc(neqloc, Tloc)          # calculation of equilibrium distribution
function
    calc_colision(nloc, relaxloc, neqloc, Ssurceloc) # collision
    n2g_loc(gloc, nloc) # transform distribution function from moment space to
velocity space

    setfl(d_fl, flloc, i, j) # export new liquid fraction
    setg(d_g, gloc, i, j)    # export distribution function in velocity space.
```

**Code 2.3. Control of the local memory operations.**

This code (only section A) performs the most important and demanding calculations in the program, it is called from MRT_LB_shared. All these operations are performed by all processing units in parallel. Temperature is calculated as:

$$T = \sum_{i=0}^{4} g_i.$$

Enthalpy calculation:

$$H = C_p * T + f_l * La.$$

Where $C_p$ is the capacity heat, T is the temperature, $f_l$ es the liquid fraction and $La$ is the latent heat The enthalpy is calculated in all cells, even in liquid and solid regions. Calculation of liquid fraction:

$$f_l = \begin{cases} 0, & \text{if } H_k \le H_s, \\ \frac{H_k - H_s}{H_l - H_s}, & \text{if } H_s < H_k < H_l, \\ 1, & \text{if } H_k \ge H_l, \end{cases} .$$

Transformation from velocity space to moment space:

$$n = Ng$$

where:

$$\mathbf{N} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \\ -4 & 1 & 1 & 1 & 1 \\ 0 & 1 & -1 & 1 & -1 \end{pmatrix}$$

Recall $g_{k,i,j}$ (g) is a tensor with 3 indexes, Matrix N operates in parallel the tensor g with k-index in every i,j position of domain.

```
                    #_________ Section A _________#
import numpy as np
from numba import cuda

def getg(d_g, gloc, i, j):    # every processing unit obtain a cell of global
    gloc[0] = d_g[i, j, 0]    # distribution function tensor
    gloc[1] = d_g[i, j, 1]
    gloc[2] = d_g[i, j, 2]
    gloc[3] = d_g[i, j, 3]
    gloc[4] = d_g[i, j, 4]

def getfl(d_fl, flloc, i, j): # obtain liquid fraction from matrix (shared memory)
    flloc[0] = d_fl[i,j]

def calc_T(gloc, Tloc):       # calculation of temperature as the sum
    Tloc[0] = gloc[0] + gloc[1] + gloc[2] + gloc[3] + gloc[4]

def calc_copiafl(copiaflloc, flloc): # take a copy of liquid fraction
    copiaflloc[0] = flloc[0]

def calc_Hk(Tloc, Hkloc, flloc):# enthalpy calculation
    Cps = 1.0                   # specific heat of solid
    La = 1.0                    # latent heat
    Hkloc[0] = Cps*Tloc[0] + flloc[0]*La #Tloc[0]#d_fl[i,j]

def calc_fl(flloc, Hkloc):      # new liquid fraction
    Hs = -0.02     # H_s = Cps*(-0.02) + 0.0*La , H_l = Cpl*0.02 + 1.0*La
    Hl = 1.02
    if (Hkloc[0] <= Hs):
        flloc[0] = 0.0      # liquid fraction in solid region
    elif ((Hkloc[0] > Hs) and (Hkloc[0] < Hl)):  # liquid fraction between
        flloc[0] = (Hkloc[0] - Hs)/(Hl - Hs)      # solid and liquid regions
    else:
        flloc[0] = 1.0      # liquid fraction in liquid region

def calc_g2n(nloc, gloc):  # transformation from velocity space to moment space
    nloc[0] = gloc[0] + gloc[1] + gloc[2] + gloc[3] + gloc[4]
    nloc[1] = gloc[1] - gloc[3]
    nloc[2] = gloc[2] - gloc[4]
    nloc[3] = -4.0*gloc[0] + gloc[1] + gloc[2] + gloc[3] + gloc[4]
    nloc[4] = gloc[1] - gloc[2] + gloc[3] - gloc[4]

def calc_alfe(alfeloc, flloc, D_al):  # obtain thermal diffusivity
    alfa_s = 0.002
    alfa_l = D_al[0]*alfa_s
    alfeloc[0] = alfa_l*flloc[0] + alfa_s*(1.0-flloc[0])

def calc_taut(tautloc, alfeloc): # relaxation parameter
    sigma=1.0
    c_st =0.4472135955
    delta_t=1.0
    tautloc[0] = 0.5 + alfeloc[0]/(sigma*c_st**2*delta_t)

def calc_relax(relaxloc, tautloc): # obtain multiple relaxation vector
    relaxloc[0] = 1.0
    relaxloc[1] = 1.0/tautloc[0]
    relaxloc[2] = 1.0/tautloc[0]
    relaxloc[3] = 1.5
    relaxloc[4] = 1.5

def calc_Ssurce(Ssurceloc, flloc, copiaflloc): # calculate S in collision step
    poros, La, sigma, Cpl, w_test = 1.00, 1.00, 1.00, 1.00, -2.00
    Ssurceloc[0] = -((poros*La)/(sigma*Cpl))*(flloc[0] - copiaflloc[0])/1.0
    Ssurceloc[1] = 0.0
    Ssurceloc[2] = 0.0
    Ssurceloc[3] = -w_test*((poros*La)/(sigma*Cpl))*(flloc[0] - copiaflloc[0])/1.0
    Ssurceloc[4] = 0.0

def n_eq_loc(neqloc, Tloc):  # calculate equilibrium distribution function
    w_test = -2.0
    neqloc[0] = Tloc[0]
    neqloc[1] = 0.0
    neqloc[2] = 0.0
    neqloc[3] = w_test*Tloc[0]
    neqloc[4] = 0.0

def calc_colision(nloc, relaxloc, neqloc, Ssurceloc):   # collision step
    nloc[0] = nloc[0] - relaxloc[0] * (nloc[0]-neqloc[0]) + Ssurceloc[0]
    nloc[1] = nloc[1] - relaxloc[1] * (nloc[1]-neqloc[1]) + Ssurceloc[1]
    nloc[2] = nloc[2] - relaxloc[2] * (nloc[2]-neqloc[2]) + Ssurceloc[2]
    nloc[3] = nloc[3] - relaxloc[3] * (nloc[3]-neqloc[3]) + Ssurceloc[3]
    nloc[4] = nloc[4] - relaxloc[4] * (nloc[4]-neqloc[4]) + Ssurceloc[4]

# transform distribution function from moment space to velocity space
def n2g_loc(gloc, nloc):
    gloc[0] = 0.2*nloc[0] - 0.2*nloc[3]
    gloc[1] = 0.2*nloc[0] + 0.5*nloc[1] + 0.05*nloc[3] + 0.25*nloc[4]
    gloc[2] = 0.2*nloc[0] + 0.5*nloc[2] + 0.05*nloc[3] - 0.25*nloc[4]
    gloc[3] = 0.2*nloc[0] - 0.5*nloc[1] + 0.05*nloc[3] + 0.25*nloc[4]
    gloc[4] = 0.2*nloc[0] - 0.5*nloc[2] + 0.05*nloc[3] - 0.25*nloc[4]

def setfl(d_fl, flloc, i, j): # export liquid fraction as a matrix
    d_fl[i,j] = flloc[0]

def setg(d_g, gloc, i, j): # export distribution function as a tensor
    d_g[i, j, 0] = gloc[0]
    d_g[i, j, 1] = gloc[1]
    d_g[i, j, 2] = gloc[2]
    d_g[i, j, 3] = gloc[3]
    d_g[i, j, 4] = gloc[4]
```

### 1.4.3 Case 2.

**Code 1. Main program with CUDA parallelization.**

Numerical solution through MRLB and parallel computing in CUDA. This code models the phase change of Gallium immersed in a 2D porous media. Energy and moment equations are implemented. This code is inspired by [5]. We divide the code into sections as an aid to the reader.

First we import the libraries.

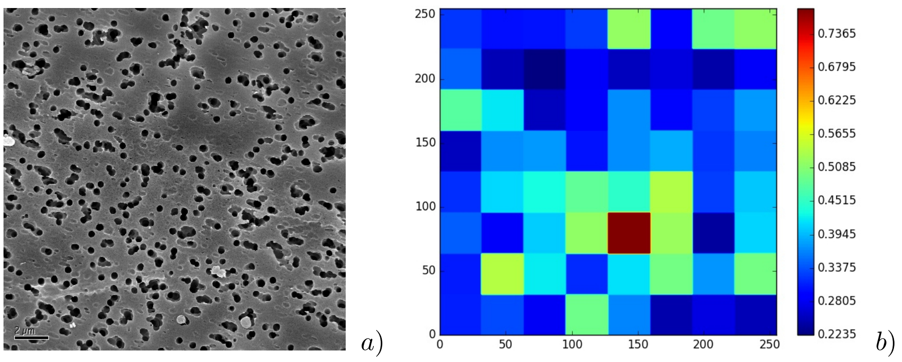


Figure 6: a) Packed porous media image [8], b) porosity map obtained from the packed porous media with 8x8 lattice cells.

```
                         #__________ Section A __________#
import numpy as np
import scipy.misc, sys, random
from numpy import linalg as LA
from numba import jit,cuda
import time
import matplotlib.pyplot as plt
import modulos as mod
```

In section B begins the main program, we define parameters that might change in other simulations and can be edited by the reader. Recall that our study is done using Ga.

```
                        #__________ Section B __________#
if __name__ == '__main__':
    lattice_x , lattice_y = 256, 256 # EDIT lattices in X, Y directions
    Ra = 8.409e5          # Rayleight number
    J = 1.0               # viscosity ratio
    Pr = 0.0208           # Prandtl number
    dm = 25.21118882097 # diameter of particle
    Lambda = 0.2719       # effective thermal diffusivity
    T_h = 45.0            # hot temperature (liquid)
    T_c = 20.0            # cold temperature (solid)
    Cp=Cpl=Cps = 1.0      # specific heat
    T_m = 29.78           # melting temperature
    T_i = T_o = 20.0      # initial temperature
    H = float(lattice_y)  # characteristic length
    Fo = 1.829            # Fourier number
    St = 0.1241           # Stephan number
    Ma = 0.1              # Mach number
    sigma_l = 0.8604      # thermal capacity ratio in liquid
    sigma_s = 0.8352      # thermal capacity ratio in solid
    La = Cpl*(T_h-T_c)/St # latent heat
    rho_o = 1.0           # reference density
    w_test = -2.0         # constant of D2Q5 model
    delta = delta_x = delta_y = delta_t=1.0  # space and time step
    c = delta_x/delta_t # lattice speed
    c_st = np.sqrt(1.0/5.0) # sound speed of the D2Q5 model
    c_s = c/np.sqrt(3.0)    # sound speed of the D2Q9 model
    H_l = Cpl*(T_m+0.5) + 1.0*La # liquid enthalpy
    H_s = Cps*(T_m-0.5) + 0.0*La # solid enthalpy
    t = int((Fo*H)/0.08731) # time
    tau_v = 0.5           # Relaxation time parameter
```

We calculate porosity and permeability from a the porous media image shown in Fig 6 in section C. The flowchart for these calculations is shown in Figure 7 where porous and permeability coarse matrices are obtained.

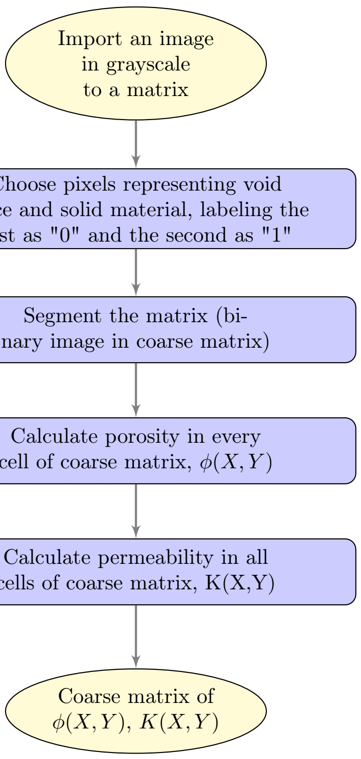

Import an image in grayscale to a matrix
Choose pixels representing void space and solid material, labeling the first as "0" and the second as "1"
Segment the matrix (binary image in coarse matrix)
Calculate porosity in every cell of coarse matrix, $\phi(X,Y)$
Calculate permeability in all cells of coarse matrix, K(X,Y)
Coarse matrix of $\phi(X,Y)$, $K(X,Y)$

Figure 7: Porous image treatment flow diagram.

```
                    #__________ Section C __________#
    medio = scipy.misc.imread('m_poro_.png',flatten=True) # read the image and
    transform to a grayscale matrix
    np.savetxt('medio_no-binario.dat', medio,fmt='%.2f')  # save gray scale matrix
    nx,ny = medio.shape  # obtain nx, ny as the shape of matrix image
    size_im = nx*ny       # size image
    umbral = 90.0         # threshold defines that before 90 it is a pore and equal
    or higher than 90 is solid
    for i in range(nx):
        for j in range(ny):
            if medio[i,j] <= umbral:   # pore is assigned as 1
                medio[i,j]= 1.0
            else:
                medio[i,j]=0.0         # solid is assigned as 0

    porosidad = np.sum(medio)/size_im  # porosity = (pore elements)/total elements
    print "\n Porosidad global: ", porosidad # print global porosity

    # porous media(matrix or image) is divided into subregions (part_x X part_y)
    part_x = 4       # EDIT  this line to change the number of x divisions
    part_y = 4       # EDIT  this line to change the number of x divisions
    div_x = nx//part_x      # find the entire of nx/part_x
    div_y = ny//part_y      # find the entire of ny/part_y

    if nx % part_x != 0 or ny % part_y != 0:    # ensure that the remainder is zero
        print "\n correct (part_x, part_y) in order to be divided by ",nx, ny
        sys.exit(1) # program break if reminder is different of zero.

    porosidades=np.zeros([part_x, part_y]) # coarse porous matrix (empty)

    m,init_1 = 0, div_x
    for i in range (0,nx, div_x):
        n, init_2 = 0, div_y
        for j in range (0,ny, div_y): # fill coarse porous matrix
            porosidades[m,n] = (medio[i:init_1, j:init_2].sum()) / float((nx*ny)/(
    part_x*part_y))
            if porosidades[m,n]==1.0:
                porosidades[m,n]=random.uniform(0.9, 0.99)
            init_2 = init_2+div_y
            n = n+1
        init_1 = init_1+div_x
        m = m+1
    print "\n porosity mean (matrix): ",porosidades.mean() # mean porosity.
    np.savetxt('porosidades.dat', porosidades,fmt='%.5f')  # save coarse porosity
    matrix
```

Then we map out the coarse porosity matrix to a global porosity matrix, same size of domain. In our case we (initially) calculated permeability assuming that the whole porous medium is formed of packed spheres. Thus Kozeny law is applicable; straightforward to the homogeneous porosity medium or locally to an heterogeneous one.

$$K = a\frac{\phi^3 d_m^2}{(1-\phi)}$$

Here we set the average particle diameter $d_m$=25.2111 and $a$=$\frac{1}{175}$. These choices can be edited as well. Then, we assume that permeability is only a function of porosity locally in the medium.

```
                    #__________ Section D __________#
    poros = np.zeros([lattice_x, lattice_y])
    Ks = np.zeros([lattice_x, lattice_y])

    steep_x = lattice_x/part_x
    steep_y = lattice_y/part_y

    k2=0       # Construction of porosity matrix
    for i in range(0,lattice_y,steep_y):
        k1=0
        for j in range(0,lattice_x,steep_x):
            for k in range(steep_y):
                for l in range(steep_x):
                 poros[j+l,  i+k] = porosidades[k1,k2] # global porosity matrix
            k1 = k1+1
        k2 = k2+1

    for i in range(lattice_y):          # construction of permeability matrix, Ks
        for j in range(lattice_x):
            Ks[i,j]=(poros[i,j]**3 * dm**2)/(175*(1.0-poros[i,j])**2)

    np.savetxt('poros.dat', poros,fmt='%.5f') # save porosity matrix
    np.savetxt('K.dat', Ks,fmt='%.5f')        # save permeability matrix
```

In section E empty arrays are defined, these arrays will be filled later by the code. For example, we introduce the temperature as a matrix size ($lattice_x$, $lattice_y$) where $lattice_x, lattice_y$ are defined in section B. The elements of these matrix are calculated and updated by the code as needed, they are placed in the right entry of the temperature matrix ( initially void) as we performed the calculations to the standards we required.

```
                     #__________ Section E __________#
 # arrays for temperature field
   T = np.ones([lattice_x, lattice_y])      # matrix of temperatures
   g = np.zeros([lattice_x, lattice_y, 5]) # distribution function
   g_eq = np.zeros([lattice_x, lattice_y, 5]) # equilibrium distribution function
   H_k  = np.zeros([lattice_x,lattice_y])  # enthalpy matrix
   f_l  = np.zeros([lattice_x,lattice_y])  # liquid fraction matrix
   w_s = np.zeros([5])                     # weights vector

   # arrays for velocity field
   den = np.ones([lattice_x,lattice_y])     # density matrix
   f = np.zeros([lattice_x,lattice_y,9])    # distribution function
   f_eq = np.zeros([lattice_x,lattice_y,9])# equilibrium distribution function
   w = np.zeros([9])                        # weights
   s = np.array([1.0, 1.1, 1.1, 1.0, 1.2, 1.0, 1.2, 1.0/tau_v, 1.0/tau_v])
   S = np.zeros([9])
   relax_f = np.zeros([9,9])   # force term
   np.fill_diagonal(relax_f,s) # relaxation matrix
   vel_vx = np.zeros([lattice_x,lattice_y]) # mesoscopic velocity in x direction
   vel_vy = np.zeros([lattice_x,lattice_y]) # mesoscopic velocity in y direction
   vel_ux = np.zeros([lattice_x,lattice_y]) # macroscopic velocity in x direction
   vel_uy = np.zeros([lattice_x,lattice_y]) # macroscopic velocity in y direction
   Fx = np.zeros([lattice_x,lattice_y])     # force term in x direction
   Fy = np.zeros([lattice_x,lattice_y])     # force term in y direction

   # definition of the D2Q5 weights in each direction (temperature field):
   for k in range(5):
       if k == 0:
           w_s[k] = (1.0-w_test)/5.0
       else:
           w_s[k] = (4.0+w_test)/20.0

   # definition of the D2Q9 weights in each direction (velocity field):
   for k in range(9):
       if k == 0:
           w[k] = 4./9.
       elif (k >= 1 and k <= 4):
           w[k] = 1./9.
       else:
           w[k] = 1./36.
```

The initial conditions are established in section F. Initially all the domain is at equilibrium at the low temperature and we only have (Ga) solid, the liquid fraction is zero. Thus, and according to this, enthalpy, density and distribution functions are initialized.

```
                     #__________ Section F __________#
   # Temperature field:
   T=T_i*T
   for j in range (lattice_y):
       T[0,j]=T_h                   # temperature
       T[lattice_x-1,j] = T_c
       f_l[0,j]=1.0                 # liquid fraction
       f_l[lattice_x-1,j] =0.0
   f_2l = np.copy(f_l)              # copy of liquid fraction
```

```
    for i in xrange(lattice_x):
        for j in xrange(lattice_y):
            H_k[i,j] = Cps*T[i,j] + f_l[i,j]*La # enthalpy

    for i in range(lattice_x):
        for j in range(lattice_y):
            for k in range(5):                    # temperature field:
                g_eq[i,j,k] = w_s[k]*T[i,j]       # equilibrium distribution function
    g=np.copy(g_eq)                    # distribution function
    g2=np.copy(g_eq)                   # copy of distribution function

    for i in range(lattice_x):
        for j in range(lattice_y):
            for k in xrange(9):                   # velocity field:
                f_eq[i,j,k] = w[k]*den[i,j]       # equilibrium distribution function
    f=np.copy(f_eq)                    # distribution function
    f2=np.copy(f_eq)                   # copy of distribution function
```

The definition of CUDA parameters and arrays that are sent from the host to the device are set in section in section G. This allows us to operate in the CUDA kernel.

```
                    #__________ Section G __________#
    threads = 256,1                        # treads per block
    blocks = (lattice_x/threads[0]+(0!=lattice_x%threads[0]), # blocks per grid
              lattice_y/threads[1]+(0!=lattice_y%threads[1]) )

    # send arrays from host to device:
    d_f = cuda.to_device(f)
    d_f2 = cuda.to_device(f2)
    d_g = cuda.to_device(g)
    d_g2 = cuda.to_device(g2)
    d_vel_ux = cuda.to_device(vel_ux)
    d_vel_uy = cuda.to_device(vel_uy)
    d_Fx = cuda.to_device(Fx)
    d_Fy = cuda.to_device(Fy)
    d_den = cuda.to_device(den)
    d_T = cuda.to_device(T)
    d_fl = cuda.to_device(f_l)
    d_f2l = cuda.to_device(f_2l)
    d_poros = cuda.to_device(poros)
    d_Ks = cuda.to_device(Ks)
```

In section H we introduced the main loop, call the "modulos.py" as "mod" and operates in the device memory.

```
                    #__________ Section H __________#
    pasos = 2000000                        # EDIT THIS LINE to change the time steps
    tiempo_cuda_1 = time.time()            # measure the time at the beginning
    for ii in xrange(pasos):               # main loop

        mod.momento[blocks, threads](d_f, d_vel_ux, d_vel_uy, d_Fx, d_Fy, d_den,
    d_fl, d_poros)      # collision step in velocity field
        mod.energia[blocks, threads](d_g,d_T,d_fl,d_vel_ux,d_vel_uy,d_f2l, d_poros)
      #  collision step in temperature field
        mod.propagacion[blocks, threads](d_f,d_f2)     # propagation in velocity
    field
        mod.propagacion_g[blocks, threads](d_g, d_g2) # propagation in temperature
    field

        mod.c_frontera[blocks, threads](d_f)       # boundary conditions in velocity
     field
        mod.c_frontera_g[blocks, threads](d_g)     # boundary conditions in
    temperature field

        mod.cal_den_u_F[blocks, threads](d_f, d_vel_ux, d_vel_uy, d_Fx, d_Fy, d_den
    , d_fl, d_T, d_poros, d_Ks)        # calculation of macroscopic variables (
    velocity field)
        mod.cal_T_fl_H[blocks, threads](d_g, d_T, d_fl, d_f2l)     # calculation of
    macroscopic variables in temperature field.

    d_f.to_host()        # send f from device to host
    d_g.to_host()        # send g from device to host
    d_vel_ux.to_host()  # send velocity in x direction, from device to host
    d_vel_uy.to_host()  # send velocity in y direction, from device to host
    d_T.to_host()        # send T from device to host
    d_den.to_host()      # send density from device to host
    d_fl.to_host()       # send liquid fraction from device to host
    tiempo_cuda_2 = time.time() # measure the time at the end of CUDA calculations.
```

The calculation of the streamlines is introduced in section I.

```
                        #__________ Section I __________#
    strf=np.zeros([lattice_x,lattice_y])
    strf[0,0]=0.0
    for j in range(lattice_y):
        rhoav=0.5*(den[0,j-1]+den[0,j])
        if j != 0.0: strf[0,j] = strf[0,j-1]-rhoav*0.5*(vel_uy[0,j-1]+vel_uy[0,j])
        for i in range(1,lattice_x):
            rhom=0.5*(den[i,j]+den[i-1,j])
            strf[i,j]=strf[i-1,j]+rhom*0.5*(vel_ux[i-1,j]+vel_ux[i,j]) # this
    matrix contains the stream lines.

    strf2=np.zeros([lattice_x,lattice_y])
    strf2[0,0]=0.0
    for j in range(lattice_x):
        rhoav2=0.5*(den[j-1,0]+den[j,0])
        if j != 0.0: strf2[j,0] = strf2[j-1,0]-rhoav2*0.5*(vel_uy[j-1,0]+vel_uy[j
    ,0])
        for i in range(1,lattice_y):
            rhom=0.5*(den[j,i]+den[j,i-1])
            strf2[j,i]=strf2[j,i-1]+rhom*0.5*(vel_ux[j,i-1]+vel_ux[j,i]) # this
    matrix contains the stream lines.
```

Finally macroscopic quantities should be calculated. Metrics are also needed. We estimate the simulation time and the millions of lattice-node updates per second (MLUPS). Then we save the obtained information as a .txt format file.

```
                        #__________ Section J __________#
    den=f.sum(axis=2)
    t = np.array([tiempo_cuda_2 - tiempo_cuda_1])
    mlups = np.array([lattice_x*lattice_y*pasos/t/1e6])

    np.savetxt('den.txt', den,fmt='%.13f')
    np.savetxt('vel_ux.txt', vel_ux,fmt='%.14f')
    np.savetxt('vel_uy.txt', vel_uy,fmt='%.14f')
    np.savetxt('strf.txt', strf,fmt='%.14f')
    np.savetxt('strf2.txt', strf,fmt='%.14f')
    np.savetxt('T.txt', T,fmt='%.5f')
    np.savetxt('tiempo.txt',t ,fmt='%.4f') # save time.
    np.savetxt('mlups.txt',mlups ,fmt='%.4f') # save mlups

    print "f:", f   # print solutions
    print "g:", g
    print "T:", T
    print "den_1:", den
    print "den_2:", f.sum(axis=2)
    print "vel_ux", vel_ux
    print "vel_uy", vel_uy
    print "T(g.sum):", g.sum(axis=2)
    print "f_l",f_l
    print"\n mlups:", lattice_x*lattice_y*pasos/t/1e6, "\t Tiempo:", t  # print
    MLUPS and time
```

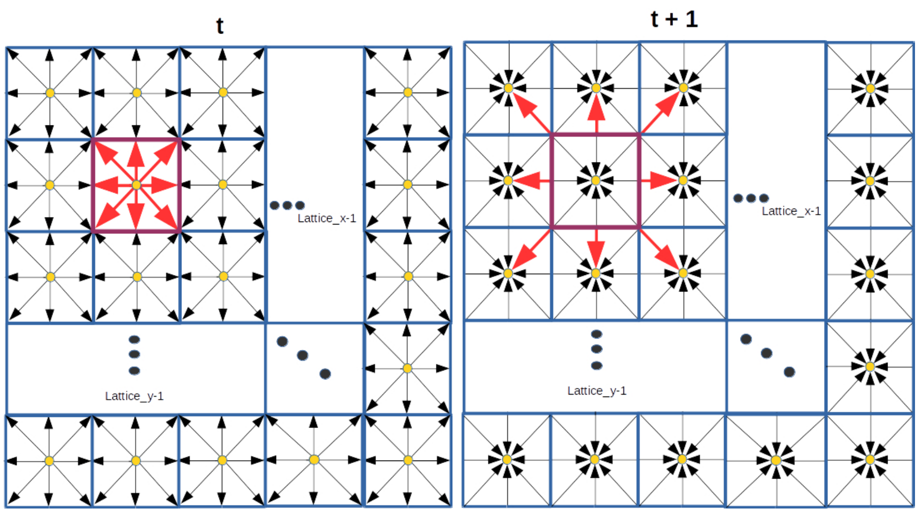
**Code 2. Modulos.py**. Here we deal with modulos.py called by last code as mod. This code is in charge of operating arrays in shared memory and communicate with local memory. In section A the libraries are called, then we define the CUDA device functions where CUDA functions link with local.py code

```
                        #__________ Section A __________#
import local as loc
from numba import cuda, float64, float32
import numpy as np
```

```
################################### velocity field ###################################
getf = cuda.jit('void (f8[:,:,:], f8[::1], i8, i8)', device=True)(loc.getf)
getux = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(loc.getux)
getuy = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(loc.getuy)
getFx = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(loc.getFx)
getFy = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(loc.getFy)
getden = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(loc.getden)
getf_l = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(loc.getf_l)
getf_2l = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(loc.getf_2l)
getT = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(loc.getT)
getporos = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(loc.getporos)
getKs = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(loc.getKs)
f2m = cuda.jit('void (f8[::1], f8[::1])', device=True)(loc.f2m)
calc_fl_PCMloc=cuda.jit('void (f8[:1], f8[:1], f8[:1])', device=True)(loc.
    calc_fl_PCMloc)
calc_noruloc=cuda.jit('void (f8[:1], f8[:1], f8[:1])', device=True)(loc.
    calc_noruloc)
calc_m_eqloc=cuda.jit('void (f8[::1], f8[:1], f8[:1], f8[:1], f8[:1], f8[:1], f8
    [:1])', device=True)(loc.calc_m_eqloc)
calc_Sloc=cuda.jit('void (f8[:1], f8[:1], f8[:1], f8[:1], f8[:1], f8[::1], f8[:1])'
    , device=True)(loc.calc_Sloc)
operloc = cuda.jit('void (f8[::1],f8[::1], f8[::1], f8[::1], f8[::1], f8[:1])',
    device=True)(loc.operloc)
colision =  cuda.jit('void (f8[::1], f8[::1], f8[::1])', device=True)(loc.colision)
m2f = cuda.jit('void (f8[::1], f8[::1])', device=True)(loc.m2f)
setf = cuda.jit('void (f8[:,:,:], f8[::1], i8, i8)', device=True)(loc.setf)
calc_denloc = cuda.jit('void (f8[:1], f8[::1])', device=True)(loc.calc_denloc)
calc_Hlsloc = cuda.jit('void (f8[:1], f8[:1])', device=True)(loc.calc_Hlsloc)
calc_cfloc = cuda.jit('void (f8[:1], f8[:1], f8[:1])', device=True)(loc.calc_cfloc)
calc_sigmaloc = cuda.jit('void (f8[:1], f8[:1])', device=True)(loc.calc_sigmaloc)
calc_tau_alpha_vl = cuda.jit('void (f8[:1], f8[:1], f8[:1], f8[:1], f8[:1], f8[:1])
    ', device=True)(loc.calc_tau_alpha_vl)
calc_lloc = cuda.jit('void (f8[:1], f8[:1], f8[:1], f8[:1], f8[:1], f8[:1])',
    device=True)(loc.calc_lloc)
calc_Gloc = cuda.jit('void (f8[:1], f8[:1], f8[:1], f8[::1],f8[:1])', device=True)(
    loc.calc_Gloc)
calc_Vloc = cuda.jit('void (f8[:1], f8[:1], f8[:1], f8[:1], f8[::1], f8[::1])',
    device=True)(loc.calc_Vloc)
calc_Uloc = cuda.jit('void (f8[:1], f8[:1], f8[:1], f8[:1], f8[:1], f8[:1], f8[:1],
     f8[:1], f8[:1], i8,i8)', device=True)(loc.calc_Uloc)
calc_Floc = cuda.jit('void (f8[:1], f8[:1], f8[:1], f8[:1], f8[:1], f8[::1], f8
    [::1], f8[:1], f8[:1], f8[:1], i8, i8)', device=True)(loc.calc_Floc)
setvar2D = cuda.jit('void (f8[:,:], f8[:1], i8, i8)', device=True)(loc.setvar2D)
################################### temperature field ###################################
getg = cuda.jit('void (f8[:,:,:], f8[::1], i8, i8)', device=True)(loc.getg)
g2n = cuda.jit('void (f8[::1], f8[::1])', device=True)(loc.g2n)
calc_neqloc = cuda.jit('void (f8[:1], f8[:1], f8[:1], f8[::1])', device=True)(loc.
    calc_neqloc)
calc_tautloc = cuda.jit('void (f8[:1], f8[:1], f8[:1])', device=True)(loc.
    calc_tautloc)
calc_relaxloc = cuda.jit('void (f8[::1], f8[:1])', device=True)(loc.calc_relaxloc)
calc_Ssurceloc = cuda.jit('void (f8[::1], f8[:1], f8[:1], f8[:1], f8[:1])', device=
    True)(loc.calc_Ssurceloc)
colis_g = cuda.jit('void (f8[::1], f8[::1], f8[::1], f8[::1])', device=True)(loc.
    colis_g)
n2g = cuda.jit('void (f8[::1], f8[::1])', device=True)(loc.n2g)
setg = cuda.jit('void (f8[:,:,:], f8[::1], i8, i8)', device=True)(loc.setg)
calc_Tloc = cuda.jit('void (f8[::1], f8[:1])', device=True)(loc.calc_Tloc)
calc_cpfl = cuda.jit('void (f8[:1], f8[:1])', device=True)(loc.calc_cpfl)
calc_Hk = cuda.jit('void (f8[:1], f8[:1], f8[:1])', device=True)(loc.calc_Hk)
calc_fl = cuda.jit('void (f8[:1], f8[:1])', device=True)(loc.calc_fl)
```

We executed the streaming step for the velocity field in section B, these operations are performed in shared memory, as shown in figure 8.

```
                    #__________ Section B __________#
@cuda.jit('void(f8[:,:,:], f8[:,:,:])')
def propagacion(f, f2):
    nx, ny, ns = f.shape
    i, j = cuda.grid(2)

    for k in xrange(ns):
```



(a) Streaming step in $t$ (b) Steaming step in $t+1$

Figure 8: Streaming step of velocity field

```
        f2[i,j,k] = f[i,j,k]

    if i > 0:
        f[i,j,1] = f2[i-1,j,1]
    if j < ny-1:
        f[i,j,2] = f2[i,j+1,2]
    if i < nx-1:
        f[i,j,3] = f2[i+1,j,3]
    if j > 0:
        f[i,j,4] = f2[i,j-1,4]
    if i > 0 and j < ny-1:
        f[i,j,5] = f2[i-1,j+1,5]
    if i < nx-1 and j < ny-1:
        f[i,j,6] = f2[i+1,j+1,6]
    if i < nx-1 and j > 0:
        f[i,j,7] = f2[i+1,j-1,7]
    if i > 0 and j > 0:
        f[i,j,8] = f2[i-1,j-1,8]
```

In section C the streaming step is executed for the temperature field, these operations are performed in shared memory, see figure 4

```
                    #__________ Section C __________#
@cuda.jit('void(f8[:,:,:], f8[:,:,:])')
def propagacion_g(g, g2):
    nx, ny, ns = g.shape
    i, j = cuda.grid(2)

    for k in xrange(ns):
        g2[i,j,k] = g[i,j,k] # copy the distribution function

    if i > 0:
        g[i,j,1] = g2[i-1,j,1] # propagate distribution function in direction 1
    if j < ny-1:
        g[i,j,2] = g2[i,j+1,2] # propagate distribution function in direction 2
    if i < nx-1:
        g[i,j,3] = g2[i+1,j,3] # propagate distribution function in direction 3
    if j > 0:
        g[i,j,4] = g2[i,j-1,4] # propagate distribution function in direction 4
```

The boundary conditions for the velocity field are bounce-back, as can be seen in Section D, see figure 9.

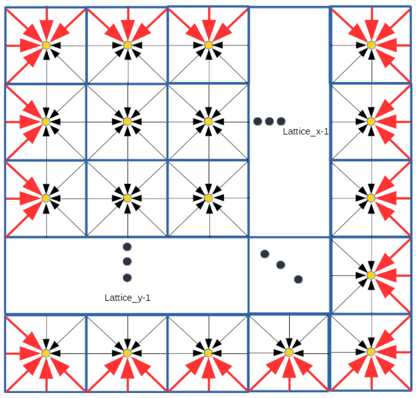
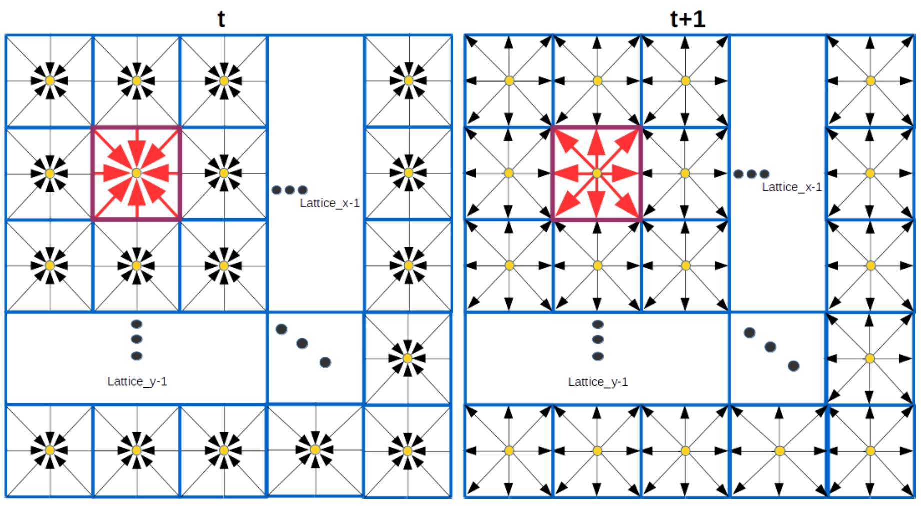


Figure 9: Boundary condition step of velocity field

```
                        #__________ Section D __________#
@cuda.jit('void(f8[:,:,:])')
def c_frontera(f):
    nx, ny, ns = f.shape
    i, j = cuda.grid(2) # CUDA grid dimension

    if i==0:             # west
        f[i,j,1] = f[i,j,3]
        f[i,j,5] = f[i,j,7]
        f[i,j,8] = f[i,j,6]

    if i==nx-1:          # east
        f[i,j,3] = f[i,j,1]
        f[i,j,6] = f[i,j,8]
        f[i,j,7] = f[i,j,5]

    if j==0:             # north
        f[i,j,4] = f[i,j,2]
        f[i,j,7] = f[i,j,5]
        f[i,j,8] = f[i,j,6]

    if j==ny-1:          # south
        f[i,j,2] = f[i,j,4]
        f[i,j,5] = f[i,j,7]
        f[i,j,6] = f[i,j,8]
```

The boundary conditions for the temperature field are set in section E as; west and east are Dirichlet while north and south are adiabatic, see figure 5

```
                        #__________ Section E __________#
@cuda.jit('void(f8[:,:,:])')
def c_frontera_g(g):
    nx, ny, ns = g.shape
    i, j = cuda.grid(2)

    T_h, T_c = 45.0, 20.0    # temperature parameters
    w_s0, w_s1, w_s2, w_s3, w_s4 = 0.6, 0.1, 0.1, 0.1, 0.1 # weight parameters

    if i==0:             # west  (Dirichlet)
        g[i,j,1] = w_s1*T_h + w_s3*T_h - g[i,j,3]

    if i==nx-1:          # east  (Dirichlet)
        g[i,j,3] = w_s1*T_c + w_s3*T_c - g[i,j,1]

    if j==0:             # north (adiabatic)
        g[i,j,0] = g[i,j+1,0]
        g[i,j,1] = g[i,j+1,1]
        g[i,j,2] = g[i,j+1,2]
        g[i,j,3] = g[i,j+1,3]
        g[i,j,4] = g[i,j+1,4]

    if j==ny-1:          # south (adiabatic)
        g[i,j,0] = g[i,j-1,0]
```



(a) Collision step in $t$ (b) Collision step in $t+1$

Figure 10: Collision step of velocity field

```
        g[i,j,1] = g[i,j-1,1]
        g[i,j,2] = g[i,j-1,2]
        g[i,j,3] = g[i,j-1,3]
        g[i,j,4] = g[i,j-1,4]
```

In section F the collision step is described for the velocity field which is performed in local memory, see figure 10. This code calls to local.py code as loc, to operate. Mathematically it is written as:

$$\mathbf{m}^{+}(\mathbf{x},t)=\mathbf{m}(\mathbf{x},t)-\Lambda\left[\mathbf{m}(\mathbf{x},t)-\mathbf{m}^{(\mathbf{eq})}(\mathbf{x},t)\right]+\delta_t\left(\mathbf{I}-\frac{\Lambda}{2}\right)\mathbf{S}$$

In section F is calculated velocity, external force, density, liquid fraction, porosity, collision and linear spaces transformations. The mathematical equations that describe this operations are showed in detail in the **Code 3 local.py**.

```
                    #__________ Section F __________#
@cuda.jit('void(f8[:,:,:],f8[:,:],f8[:,:],f8[:,:],f8[:,:],f8[:,:],f8[:,:],f8[:,:])'
    )
def momento(d_f, d_vel_ux, d_vel_uy, d_Fx, d_Fy, d_den, d_fl, d_poros):
    floc = cuda.local.array(9, dtype = float64)        # distribution function
    uxloc = cuda.local.array(1, dtype = float64)       # velocity in x direction
    uyloc = cuda.local.array(1, dtype = float64)       # velocity in y direction
    Fxloc = cuda.local.array(1, dtype = float64)       # force in x direction
    Fyloc = cuda.local.array(1, dtype = float64)       # force in y direction
    denloc = cuda.local.array(1, dtype = float64)      # density
    f_lloc = cuda.local.array(1, dtype = float64)      # liquid fraction
    nor_uloc=cuda.local.array(1, dtype = float64)      # velocity norm
    mloc = cuda.local.array(9, dtype = float64)        # distribution function in
    moment space
    fl_PCMloc = cuda.local.array(1, dtype = float64)# liquid fraction
    m_eqloc = cuda.local.array(9, dtype = float64) # equilibrium distribution
    function in moment space
    Sloc = cuda.local.array(9, dtype = float64)
    res1loc = cuda.local.array(9, dtype = float64)
    fuenloc = cuda.local.array(9, dtype = float64)
    porosloc = cuda.local.array(1, dtype = float64)# porosity

    i, j = cuda.grid(2)  # grid size in 2-dimensions

    getf(d_f, floc, i, j)          # obtain distribution function in local memory
    getux(d_vel_ux, uxloc, i, j)    # obtain velocity in x direction in local
    memory
    getuy(d_vel_uy, uyloc, i, j)    # obtain velocity in y direction in local
    memory
    getFx(d_Fx, Fxloc, i, j)        # obtain force in x direction in local memory
    getFy(d_Fy, Fyloc, i, j)        # obtain force in y direction in local memory
    getden(d_den, denloc, i, j)      # obtain density in local memory
    getf_l(d_fl, f_lloc, i, j)       # obtain liquid fraction in local memory
    getporos(d_poros, porosloc, i, j) # obtain porosity in local memory
    f2m(mloc, floc)                  # transform distribution function from velocity
     space to moment space
    calc_fl_PCMloc(f_lloc, fl_PCMloc, porosloc)  # calculate liquid fraction
    calc_noruloc(nor_uloc, uxloc, uyloc) # calculate velocity norm
    calc_m_eqloc(m_eqloc, denloc, nor_uloc, fl_PCMloc, uxloc, uyloc, f_lloc)
    calc_Sloc(uxloc, uyloc, Fxloc, Fyloc, fl_PCMloc, Sloc, f_lloc)
    operloc(res1loc, fuenloc, mloc, m_eqloc, Sloc, f_lloc)
    colision(mloc, res1loc, fuenloc)  # collision step
    m2f(floc, mloc)                    # transform distribution function from moment
     space to velocity space
    setf(d_f, floc, i, j)
```

The calculation of the macroscopic variables associated to the velocity field is performed in local memory. See section G.

```
                        #__________ Section G __________#
@cuda.jit('void(f8[:,:,:],f8[:,:],f8[:,:],f8[:,:],f8[:,:],f8[:,:],f8[:,:],f8[:,:],
    f8[:,:],f8[:,:])')
def cal_den_u_F(d_f, d_vel_ux, d_vel_uy, d_Fx, d_Fy, d_den, d_fl, d_T, d_poros,
    d_Ks):
    floc = cuda.local.array(9, dtype = float64)  # distribution function
    vxloc = cuda.local.array(1, dtype = float64) # velocity in x direction
    vyloc = cuda.local.array(1, dtype = float64) # velocity in y direction
    uxloc = cuda.local.array(1, dtype = float64) # macroscopic velocity x
    uyloc = cuda.local.array(1, dtype = float64) # macroscopic velocity y
    nor_vloc=cuda.local.array(1, dtype = float64)# mesoscopic velocity norm
    nor_uloc=cuda.local.array(1, dtype = float64)# macroscopic velocity norm
    Fxloc = cuda.local.array(1, dtype = float64) # force in x direction
    Fyloc = cuda.local.array(1, dtype = float64) # force in y direction
    denloc = cuda.local.array(1, dtype = float64)# density
    f_lloc = cuda.local.array(1, dtype = float64)# liquid fraction
    fl_PCMloc = cuda.local.array(1, dtype = float64)# liquid fraction of PCM
    H_lloc = cuda.local.array(1, dtype = float64)# liquid enthalpy
    H_sloc = cuda.local.array(1, dtype = float64)# solid enthalpy
    cfloc = cuda.local.array(1, dtype = float64) # properties of porous media
    Tloc = cuda.local.array(1, dtype = float64)  # temperature
    sigmaloc = cuda.local.array(1, dtype = float64) # thermal capacity ratio
    tau_tloc = cuda.local.array(1, dtype = float64) # relaxation parameter
    alf_eloc = cuda.local.array(1, dtype = float64) #
    alf_lloc = cuda.local.array(1, dtype = float64) # thermal diffusivity of liquid
    vlloc = cuda.local.array(1, dtype = float64) # mesoscopic velocity
    l_0loc = cuda.local.array(1, dtype = float64)# parameter 0 of velocity equation
    l_1loc = cuda.local.array(1, dtype = float64)# parameter 1 of velocity equation
    Gloc = cuda.local.array(9, dtype = float64)  # buoyancy force
    TFloc = cuda.local.array(9, dtype = float64)
    porosloc = cuda.local.array(1, dtype = float64) #porosity
    Ks = cuda.local.array(1, dtype = float64)    # permeability

    i, j = cuda.grid(2)  # CUDA grid dimension
```

The following functions operate in local memory, calling **local.py** code as loc. In the **code 3 local.py** we describe the code lines and equations solved in each function. Note that PCM is the acronym for phase change material. See section H.

```
                        #__________ Section H __________#
    # obtain arrays from shared memory to local memory
    getf(d_f, floc, i, j)
    getf_l(d_fl, f_lloc, i, j)
    getT(d_T, Tloc, i, j)
    getporos(d_poros, porosloc, i, j)
    getKs(d_Ks, Ks, i, j)

    calc_Hlsloc(H_lloc, H_sloc)              # compute enthalpy
    calc_denloc(denloc, floc)                # compute density
    calc_fl_PCMloc(f_lloc, fl_PCMloc, porosloc) # compute PCM liquid fraction
    calc_cfloc(cfloc, fl_PCMloc, f_lloc)     # compute inertial coefficient
    calc_sigmaloc(Tloc, sigmaloc)            # compute thermal capacity ratio
    calc_tau_alpha_vl(sigmaloc, tau_tloc, alf_eloc, alf_lloc, vlloc, f_lloc)
    calc_lloc(fl_PCMloc, vlloc, cfloc, l_0loc, l_1loc, Ks) # compute parameters l
    calc_Gloc(vlloc, alf_lloc, Tloc, Gloc,f_lloc) # compute buoyancy force
    calc_Vloc(vxloc, vyloc, fl_PCMloc, nor_vloc, Gloc, floc)# compute mesoscopic
    velocity
    calc_Uloc(l_0loc, l_1loc, vxloc, vyloc, nor_vloc, uxloc, uyloc, nor_uloc,
    fl_PCMloc, i, j)    # compute macroscopic velocity
    calc_Floc(fl_PCMloc, uxloc, uyloc, cfloc, vlloc, Gloc, TFloc, Fxloc, Fyloc, Ks,
     i, j)              # compute total body force
    setvar2D(d_den, denloc, i, j)        # set density as an array
    setvar2D(d_vel_ux, uxloc, i, j)      # set velocity in x direction as an array
    setvar2D(d_vel_uy, uyloc, i, j)      # set velocity in y direction as an array
    setvar2D(d_Fx, Fxloc, i, j)          # set force in x direction as an array
    setvar2D(d_Fy, Fyloc, i, j)          # set force in y direction as an array
```

The collision step for the temperature field is calculated in local memory. Section I calls to **local.py** code as loc, to execute. Mathematically we are doing in this section the following;

$$\mathbf{n}^{+}(\mathbf{x},t) = \mathbf{n}(\mathbf{x},t) - \mathbf{\Theta}\left[\mathbf{n}(\mathbf{x},t) - \mathbf{n}^{(eq)}(\mathbf{x},t)\right] + \delta_t \tilde{\mathbf{S}}$$

```
                    #__________ Section I __________#
@cuda.jit('void(f8[:,:,:],f8[:,:],f8[:,:],f8[:,:],f8[:,:], f8[:,:], f8[:,:])')
def energia(d_g, d_T, d_fl, d_vel_ux, d_vel_uy, d_f2l, d_poros):
    gloc = cuda.local.array(5, dtype = float64)     # distribution function g
    nloc = cuda.local.array(5, dtype = float64)     # n=Ng
    neqloc = cuda.local.array(5, dtype = float64)   # equilibrium distribution
    function
    f_lloc = cuda.local.array(1, dtype = float64)   # liquid fraction
    f_2lloc = cuda.local.array(1, dtype = float64)  # copy liquid fraction
    Tloc = cuda.local.array(1, dtype = float64)     # local temperature
    uxloc = cuda.local.array(1, dtype = float64)    # macroscopic velocity in x
    uyloc = cuda.local.array(1, dtype = float64)    # macroscopic velocity in y
    tautloc = cuda.local.array(1, dtype = float64)  # relaxation time
    relaxloc = cuda.local.array(5, dtype = float64) # multiple relaxation vector
    Ssurceloc = cuda.local.array(5, dtype = float64)# source vector
    porosloc = cuda.local.array(1, dtype = float64) # porosity

    i, j = cuda.grid(2)  # CUDA grid dimension

    # obtain arrays for operating in local memory
    getf_l(d_fl, f_lloc, i, j)      # obtain liquid fraction
    getf_2l(d_f2l, f_2lloc, i, j)   # obtain copy of liquid fraction
    getg(d_g, gloc, i, j)           # obtain distribution function
    g2n(nloc, gloc)                 # transform distribution function
    getT(d_T, Tloc, i, j)           # obtain temperature
    getux(d_vel_ux, uxloc, i, j)    # obtain macroscopic velocity in x direction
    getuy(d_vel_uy, uyloc, i, j)    # obtain macroscopic velocity in y direction
    getporos(d_poros, porosloc, i, j)# obtain porosity
    calc_neqloc(Tloc, uxloc, uyloc, neqloc)# compute equilibrium distribution fun
    calc_tautloc(tautloc, Tloc, f_lloc)# relaxation parameter
    calc_relaxloc(relaxloc, tautloc)# compute multiple relaxation vector
    calc_Ssurceloc(Ssurceloc, f_lloc, f_2lloc, Tloc, porosloc)# compute source
    colis_g(nloc, neqloc, Ssurceloc, relaxloc) # collision step
    n2g(gloc, nloc)                 # transform distribution function
    setg(d_g, gloc, i, j)           # export distribution function as an array
```

Finally we must calculate in local memory the macroscopic variables associated to the temperature field. See section J.

```
                    #__________ Section J __________#
@cuda.jit('void(f8[:,:,:],f8[:,:],f8[:,:],f8[:,:])')
def cal_T_fl_H(d_g, d_T, d_fl, d_f2l):
    gloc = cuda.local.array(5, dtype = float64)
    Tloc = cuda.local.array(1, dtype = float64)
    f_lloc = cuda.local.array(1, dtype = float64)
    f_2lloc = cuda.local.array(1, dtype = float64)
    Hkloc = cuda.local.array(1, dtype = float64)

    i, j = cuda.grid(2)          # CUDA grid dimension

    getg(d_g, gloc, i, j)        # obtain distribution function
    getf_l(d_fl, f_lloc, i, j)   # obtain liquid fraction
    calc_Tloc(gloc, Tloc)        # compute temperature
    calc_cpfl(f_lloc, f_2lloc)   # compute liquid fraction of PCM

    calc_Hk(Tloc, Hkloc, f_lloc)  # compute enthalpy
    calc_fl(f_lloc, Hkloc)        # compute liquid fraction
    # export arrays
    setvar2D(d_fl, f_lloc, i, j)  # liquid fraction
    setvar2D(d_T, Tloc, i, j)     # temperature
    setvar2D(d_f2l, f_2lloc, i, j)# liquid fraction
```

**Code 3.local.py**. Here we perform the most important calculations in this case. This code is called from modulos.py and operate in local CUDA memory. All operations are performed simultaneously. First we describe the collision step of the velocity field as shown in the following equation.

$$\mathbf{m}^{+}(\mathbf{x},\, t) = \mathbf{m}(\mathbf{x},\, t) - \Lambda\left[\mathbf{m}(\mathbf{x}, t) - \mathbf{m}^{(\mathbf{eq})}(\mathbf{x}, t)\right] + \delta_t\left(\mathbf{I} - \frac{\Lambda}{2}\right)\mathbf{S}$$

```
                      #_________ Section A _________#
import numpy as np
from numba import cuda
import math

def getf(d_f, floc, i, j):    # obtain distribution function
    floc[0] = d_f[i, j, 0]    # in local memory from shared
    floc[1] = d_f[i, j, 1]    # memory.
    floc[2] = d_f[i, j, 2]
    floc[3] = d_f[i, j, 3]
    floc[4] = d_f[i, j, 4]
    floc[5] = d_f[i, j, 5]
    floc[6] = d_f[i, j, 6]
    floc[7] = d_f[i, j, 7]
    floc[8] = d_f[i, j, 8]

def getux(d_vel_ux, uxloc, i, j): # obtain macroscopic velocity
    uxloc[0] = d_vel_ux[i,j]

def getuy(d_vel_uy, uyloc, i, j): # obtain macroscopic velocity
    uyloc[0] = d_vel_uy[i,j]

def getFx(d_Fx, Fxloc, i, j):     # obtain force x
    Fxloc[0] = d_Fx[i,j]

def getFy(d_Fy, Fyloc, i, j):     # obtain force y
    Fyloc[0] = d_Fy[i,j]

def getden(d_den, denloc, i, j):  # obtain density
    denloc[0] = d_den[i,j]

def getf_l(d_fl, f_lloc, i, j):   # obtain liquid fraction
    f_lloc[0] = d_fl[i,j]

def getporos(d_poros, porosloc, i, j): # obtain porosity
    porosloc[0] = d_poros[i,j]

def getKs(d_Ks, Ks, i, j):        # obtain permeability
    Ks[0] = d_Ks[i,j]

def getT(d_T, Tloc, i, j):        # obtain temperature
    Tloc[0]=d_T[i,j]
```

A linear transformation from velocity space to moment space $m = Mf$ is needed. Here

$$\mathbf{M} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -4 & -1 & -1 & -1 & -1 & 2 & 2 & 2 & 2 \\ 4 & -2 & -2 & -2 & -2 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 & 1 & -1 & -1 & 1 \\ 0 & -2 & 0 & 2 & 0 & 1 & -1 & -1 & 1 \\ 0 & 0 & 1 & 0 & -1 & 1 & 1 & -1 & -1 \\ 0 & 0 & -2 & 0 & 2 & 1 & 1 & -1 & -1 \\ 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \end{pmatrix}$$

where $m$ is the distribution function (for the fluid velocity) in moment space and $f$ is the distribution function in velocity space. We perform this operation in the section B code.

```
                    #__________ Section B __________#
def f2m(mloc, floc):
    mloc[0] = floc[0] + floc[1] + floc[2] + floc[3] + floc[4] + floc[5] + floc[6] +
     floc[7] + floc[8]
    mloc[1] = -4.0*floc[0] - floc[1] - floc[2] - floc[3] - floc[4] + 2.0*floc[5] +
    2.0*floc[6] + 2.0*floc[7] + 2.0*floc[8]
    mloc[2] = 4.0*floc[0] - 2.0*floc[1] - 2.0*floc[2] - 2.0*floc[3] - 2.0*floc[4] +
     floc[5] + floc[6] + floc[7] + floc[8]
    mloc[3] = floc[1] - floc[3] + floc[5] - floc[6] - floc[7] + floc[8]
    mloc[4] = -2.0*floc[1] + 2.0*floc[3] + floc[5] - floc[6] - floc[7] + floc[8]
    mloc[5] = floc[2] - floc[4] + floc[5] + floc[6] - floc[7] - floc[8]
    mloc[6] = -2.0*floc[2] + 2.0*floc[4] + floc[5] + floc[6] - floc[7] - floc[8]
    mloc[7] = floc[1] - floc[2] + floc[3] - floc[4]
    mloc[8] = floc[5] - floc[6] + floc[7] - floc[8]
```

To calculate the fraction of the liquid in the volume element $\varphi$, we use $\varphi = \phi fl$, where $\phi$ is the porosity and $fl$ the liquid fraction. We already know that the equilibrium distribution function for the fliud dynamics is as follows.

$$\mathbf{m}^{(eq)} = \begin{pmatrix} \rho \\ -2\rho + 3\rho_0 \left|\mathbf{u}\right|^2/\varphi \\ \rho - 3\rho_0 \left|\mathbf{u}\right|^2/\varphi \\ \rho_0 u_x \\ -\rho_0 u_x \\ \rho_0 u_y \\ -\rho_0 u_y \\ \rho_0(u_x^2 - u_y^2)/\varphi \\ \rho_0 u_x u_y/\varphi \end{pmatrix}$$

The external force term in the D2Q9 MRLBM has the following components ($\mathbf{S}$);

$$S_0 = 0,\ S_1 = \frac{6\rho_0 \mathbf{u}\cdot\mathbf{F}}{\varphi},\ S_2 = -\frac{6\rho_0 \mathbf{u}\cdot\mathbf{F}}{\varphi},\ S_3 = \rho_0 F_x,\ S_4 = -\rho_0 F_x,$$

$$S_5 = \rho_0 F_y,\ S_6 = -\rho_0 F_y,\ S_7 = \frac{2\rho_0(u_x F_x - u_y F_y)}{\varphi},\ S_8 = \frac{\rho_0(u_x F_y - u_y F_x)}{\varphi}$$

To solve our problem we make a linear transformation from the moment space to the velocity space $f = M^{-1}m$.

All these calculations are executed in section C of the code.

```
                    #__________ Section C __________#
def calc_fl_PCMloc(f_lloc, fl_PCMloc, porosloc): # liquid fraction
    if f_lloc[0] >= 0.5:
        fl_PCMloc[0] = porosloc[0]*f_lloc[0]
    else:
        fl_PCMloc[0] = 0.0

def calc_noruloc(nor_uloc, uxloc, uyloc):  # norm of velocity
    nor_uloc[0] = math.sqrt(uxloc[0]**2 + uyloc[0]**2)

# equilibrium distribution function:
```

```
def calc_m_eqloc(m_eqloc, denloc, nor_uloc, fl_PCMloc, uxloc, uyloc, f_lloc):
    if f_lloc[0] >= 0.5:              # equilibrium distribution function
        m_eqloc[0] = denloc[0]
        m_eqloc[1] = -2.0*denloc[0] + 3.0*1.0*nor_uloc[0]**2/fl_PCMloc[0]
        m_eqloc[2] = denloc[0]-3.0*1.0*nor_uloc[0]**2.0/fl_PCMloc[0]
        m_eqloc[3] = 1.0*uxloc[0]
        m_eqloc[4] = -1.0*uxloc[0]
        m_eqloc[5] = 1.0*uyloc[0]
        m_eqloc[6] = -1.0*uyloc[0]
        m_eqloc[7] = 1.0*(uxloc[0]**2-uyloc[0]**2)/fl_PCMloc[0]
        m_eqloc[8] = 1.0*uxloc[0]*uyloc[0]/fl_PCMloc[0]
    else:
        m_eqloc[0] = denloc[0]
        m_eqloc[1] = -2.0*denloc[0]
        m_eqloc[2] = denloc[0]
        m_eqloc[3] = 1.0*uxloc[0]
        m_eqloc[4] = -1.0*uxloc[0]
        m_eqloc[5] = 1.0*uyloc[0]
        m_eqloc[6] = -1.0*uyloc[0]
        m_eqloc[7] = 0.0
        m_eqloc[8] = 0.0

# components of forcing term:
def calc_Sloc(uxloc, uyloc, Fxloc, Fyloc, fl_PCMloc, Sloc, f_lloc):
    if f_lloc[0] >= 0.5:
        Sloc[0] = 0.0
        Sloc[1] = (6.0*1.0*(uxloc[0]*Fxloc[0] + uyloc[0]*Fyloc[0]))/fl_PCMloc[0]
        Sloc[2] = -(6.0*1.0*(uxloc[0]*Fxloc[0] + uyloc[0]*Fyloc[0]))/fl_PCMloc[0]
        Sloc[3] = 1.0*Fxloc[0]
        Sloc[4] = -1.0*Fxloc[0]
        Sloc[5] = 1.0*Fyloc[0]
        Sloc[6] = -1.0*Fyloc[0]
        Sloc[7] = (2.0*1.0*(uxloc[0]*Fxloc[0]-uyloc[0]*Fyloc[0]))/fl_PCMloc[0]
        Sloc[8] = (1.0*(uxloc[0]*Fyloc[0] + uyloc[0]*Fxloc[0]))/fl_PCMloc[0]
    else:
        Sloc[0] = 0.0
        Sloc[1] = 0.0
        Sloc[2] = 0.0
        Sloc[3] = 1.0*Fxloc[0]
        Sloc[4] = -1.0*Fxloc[0]
        Sloc[5] = 1.0*Fyloc[0]
        Sloc[6] = -1.0*Fyloc[0]
        Sloc[7] = 0.0
        Sloc[8] = 0.0

def operloc(res1loc, fuenloc, mloc, m_eqloc, Sloc, f_lloc): # parameter of
    if f_lloc[0] >= 0.5:                                     # relaxation term
        tau_v = 0.50544816327342
    else:
        tau_v = 0.5

    res1loc[0] = 1.0*(mloc[0]-m_eqloc[0])
    res1loc[1] = 1.1*(mloc[1]-m_eqloc[1])
    res1loc[2] = 1.1*(mloc[2]-m_eqloc[2])
    res1loc[3] = 1.0*(mloc[3]-m_eqloc[3])
    res1loc[4] = 1.2*(mloc[4]-m_eqloc[4])
    res1loc[5] = 1.0*(mloc[5]-m_eqloc[5])
    res1loc[6] = 1.2*(mloc[6]-m_eqloc[6])
    res1loc[7] = (1.0/tau_v)*(mloc[7]-m_eqloc[7])
    res1loc[8] = (1.0/tau_v)*(mloc[8]-m_eqloc[8])

    fuenloc[0] = (1.0 - 0.5*1.0)*(Sloc[0])
    fuenloc[1] = (1.0 - 0.5*1.1 )*(Sloc[1])
    fuenloc[2] = (1.0 - 0.5*1.1 )*(Sloc[2])
    fuenloc[3] = (1.0 - 0.5*1.0 )*(Sloc[3])
    fuenloc[4] = (1.0 - 0.5*1.2 )*(Sloc[4])
    fuenloc[5] = (1.0 - 0.5*1.0 )*(Sloc[5])
    fuenloc[6] = (1.0 - 0.5*1.2 )*(Sloc[6])
    fuenloc[7] = (1.0 - 0.5*1.0/tau_v)*(Sloc[7])
    fuenloc[8] = (1.0 - 0.5*1.0/tau_v)*(Sloc[8])

def colision(mloc, res1loc, fuenloc):         # collision step
    mloc[0] = mloc[0]-res1loc[0]+fuenloc[0]
    mloc[1]  = mloc[1]-res1loc[1]+fuenloc[1]
    mloc[2]  = mloc[2]-res1loc[2]+fuenloc[2]
    mloc[3]  = mloc[3]-res1loc[3]+fuenloc[3]
    mloc[4]  = mloc[4]-res1loc[4]+fuenloc[4]
    mloc[5]  = mloc[5]-res1loc[5]+fuenloc[5]
    mloc[6]  = mloc[6]-res1loc[6]+fuenloc[6]
    mloc[7]  = mloc[7]-res1loc[7]+fuenloc[7]
    mloc[8]  = mloc[8]-res1loc[8]+fuenloc[8]

def m2f(floc, mloc):  # transformation from moment space to velocity space
    floc[0]=1.11111111e-01*mloc[0]-1.11111111e-01*mloc[1]+1.11111111e-01*mloc
    [2]+6.93889390e-18*mloc[3]-6.93889390e-18*mloc[4]+6.93889390e-18*mloc
    [5]-6.93889390e-18*mloc[6]
    floc[1]=1.11111111e-01*mloc[0]-2.77777778e-02*mloc[1]-5.55555556e-02*mloc
    [2]+1.66666667e-01*mloc[3]-1.66666667e-01*mloc[4]+5.55111512e-17*mloc[5]+2.5e
    -01*mloc[7]
    floc[2]=1.11111111e-01*mloc[0]-2.77777778e-02*mloc[1]-5.55555556e-02*mloc
    [2]-1.11022302e-16*mloc[4]+1.66666667e-01*mloc[5]-1.66666667e-01*mloc[6]-2.5e
    -01*mloc[7]
    floc[3]=1.11111111e-01*mloc[0]-2.77777778e-02*mloc[1]-5.55555556e-02*mloc
    [2]-1.66666667e-01*mloc[3]+1.66666667e-01*mloc[4]+1.38777878e-17*mloc
    [5]-2.77555756e-17*mloc[6]+2.5e-01*mloc[7]
    floc[4]=1.11111111e-01*mloc[0]-2.77777778e-02*mloc[1]-5.55555556e-02*mloc
    [2]-1.66666667e-01*mloc[5]+1.66666667e-01*mloc[6]-2.5e-01*mloc[7]
    floc[5]=1.11111111e-01*mloc[0]+5.55555556e-02*mloc[1]+2.77777778e-02*mloc
    [2]+1.66666667e-01*mloc[3]+8.33333333e-02*mloc[4]+1.66666667e-01*mloc
    [5]+8.33333333e-02*mloc[6]+2.5e-01*mloc[8]
    floc[6]=1.11111111e-01*mloc[0]+5.55555556e-02*mloc[1]+2.77777778e-02*mloc
    [2]-1.66666667e-01*mloc[3]-8.33333333e-02*mloc[4]+1.66666667e-01*mloc
    [5]+8.33333333e-02*mloc[6]-2.5e-01*mloc[8]
    floc[7]=1.11111111e-01*mloc[0]+5.55555556e-02*mloc[1]+2.77777778e-02*mloc
    [2]-1.66666667e-01*mloc[3]-8.33333333e-02*mloc[4]-1.66666667e-01*mloc
    [5]-8.33333333e-02*mloc[6]+2.5e-01*mloc[8]
    floc[8]=1.11111111e-01*mloc[0]+5.55555556e-02*mloc[1]+2.77777778e-02*mloc
    [2]+1.66666667e-01*mloc[3]+8.33333333e-02*mloc[4]-1.66666667e-01*mloc
    [5]-8.33333333e-02*mloc[6]-2.5e-01*mloc[8]

def setf(d_f,floc,i,j):        # export distribution function as an array
    d_f[i,j,0]  = floc[0]
    d_f[i,j,1]  = floc[1]
    d_f[i,j,2]  = floc[2]
    d_f[i,j,3]  = floc[3]
    d_f[i,j,4]  = floc[4]
    d_f[i,j,5]  = floc[5]
    d_f[i,j,6]  = floc[6]
    d_f[i,j,7]  = floc[7]
    d_f[i,j,8]  = floc[8]
```

In Section D we calculate of some macroscopic variables in local memory. In this code section we calculate the following.

1. The density

$$\rho = \sum_{i=0}^{8} f_i.$$

2. The enthalpy of the liquid and solid phases

$$H = C_p T + f_l L_a.$$

   Here $C_p$ is the heat capacity a constant pressure. $L_a$ is the phase change latent heat.

3. The inertial coefficient

$$C_F = \frac{1.75}{\sqrt{175\varphi^3}}.$$

4. The viscosity of liquid is calculate using the

$$P_r = \frac{\nu_l}{\alpha_l}; \quad \nu_l = Pr\alpha_l.$$

5. The bouyancy force $G$, based on the Boussinesq approximation, as

$$G = g\beta(T - T_0)\mathbf{j}.$$

   Here $g$ is the gravitational acceleration.$\beta$ is the thermal expansion coefficient. $T_0$ is the reference temperature, and $\mathbf{j}$ is the unit vector in the y-direction,

6. The macroscopic velocity.

   (a) The $u_x$ component

$$\mathbf{u}_x = \frac{\mathbf{v}_x}{l_0 + \sqrt{l_0^2 + l_1\,|\mathbf{v}|}}$$

   (b) The $u_y$ componenet

$$\mathbf{u}_y = \frac{\mathbf{v}_y}{l_0 + \sqrt{l_0^2 + l_1\,|\mathbf{v}|}}.$$

   where

$$l_0 = \frac{1}{2}\left(1 + \varphi\frac{\delta_t}{2}\frac{\nu_l}{K}\right)$$
$$l_1 = \varphi\frac{\delta_t}{2}\frac{C_F}{\sqrt{K}}.$$

   Here $K$ is the permeability.

7. The external force and all its components.

$$\mathbf{F}_i = -w_i\left(\frac{\varphi\nu_l}{K} * (\mathbf{u}_x\mathbf{e}_i + \mathbf{u}_y e_i) + \frac{\varphi C_F}{\sqrt{K}}\left(|\mathbf{u}_x|\mathbf{u}_x\mathbf{e}_i + |\mathbf{u}_y|\mathbf{u}_y\mathbf{e}_i\right)\right)$$

Here $\mathbf{e}_i$ is a direction in a unitary stencil. For example, for D2Q9 we have;

$$\mathbf{e}_i = \begin{cases} (0,0) & i = 0 \\ (cos[(i-1)\pi/2], sin[(i-1)\pi/2])\,c & i = 1,,2,3,4 \\ (cos[(2i-9)\pi/4], sin[(i-1)\pi/2])\sqrt{2}c & i = 5,6,7,8 \end{cases}$$

Note that permeability $K$ is obtained from an image of an actual packed medium.

```
                        #__________ Section D __________#
def calc_denloc(denloc, floc):
    denloc[0]=floc[0]+floc[1]+floc[2]+floc[3]+floc[4]+floc[5]+floc[6]+floc[7]+floc
    [8]      # calcualtion of density

def calc_Hlsloc(H_lloc, H_sloc):           # calculation of enthalpy
    Cpl=Cps=1.0
    T_m = 29.78
    La = Cpl*(45.0 - 20.0)/0.1241          # La = Cpl*(T_h-T_c)/St
    H_lloc[0] = Cpl*(T_m+0.5) + 1.0*La
    H_sloc[0] = Cps*(T_m-0.5)

def calc_cfloc(cfloc, fl_PCMloc, f_lloc): # inertial coefficient
    if (f_lloc[0] >= 0.5):
        cfloc[0] = (1.75/(math.sqrt(175.0*fl_PCMloc[0]**3.0)))
    else:
        cfloc[0] = 0.0

def calc_sigmaloc(Tloc, sigmaloc):          # thermal capacity ratio
    T_m = 29.78
    if Tloc[0] < T_m:
        sigmaloc[0] = 0.8352
    else:
        sigmaloc[0] = 0.8604

def calc_tau_alpha_vl(sigmaloc, tau_tloc, alf_eloc, alf_lloc, vlloc, f_lloc):
#    tau_v = 0.5 + (0.1*H*math.sqrt(3.0*0.0208))/(math.sqrt(8.409e5))
    if f_lloc[0] >= 0.5:
        tau_v = 0.50544816327342          # relaxation time parameter
    else:
        tau_v = 0.5

    tau_tloc[0] = 0.5 + (0.2719*(1.0/math.sqrt(3.0))**2*(tau_v-0.5))/(1.0*sigmaloc
    [0]*(math.sqrt(1.0/5.0))**2.0 *0.0208) # relaxing time parameter
    alf_eloc[0] = sigmaloc[0]*(1.0/5.0)*(tau_tloc[0]-0.5)*1.0 # effective
    diffusivity
    alf_lloc[0] = alf_eloc[0]/0.2719  # thermal diffusivity of liquid
    vlloc[0] = 0.0208*alf_lloc[0]     # viscosity of liquid vl=Prant*alpha_l

# parameters to calculate macroscopic velocity
def calc_lloc(fl_PCMloc, vlloc, cfloc, l_0loc, l_1loc, Ks):
    l_0loc[0] = 0.5*(1.0 + fl_PCMloc[0]*(1.0/2.0)*(vlloc[0]/Ks[0]))
    l_1loc[0] = fl_PCMloc[0]*(1.0/2.0)*(cfloc[0]/math.sqrt(Ks[0]))

# parameters to calculate buoyancy force
def calc_Gloc(vlloc, alf_lloc, Tloc, Gloc, f_lloc):
    H = 256.0           # characteristic length
    T_0 = 20.0          # reference temperature
    Ra = 8.409e5        # Rayleigh number
    if f_lloc[0] >= 0.5:# buoyancy force
        Gloc[0] = 0.0
        Gloc[1] = 0.0
        Gloc[2] = ((Ra*vlloc[0]*alf_lloc[0]*(Tloc[0] - T_0))/((45.0 - 20)*H**3))
    *1.0
        Gloc[3] = 0.0
        Gloc[4] = ((Ra*vlloc[0]*alf_lloc[0]*(Tloc[0] - T_0))/((45.0 - 20)*H**3))
    *-1.0
        Gloc[5] = ((Ra*vlloc[0]*alf_lloc[0]*(Tloc[0] - T_0))/((45.0 - 20)*H**3))
    *1.0
        Gloc[6] = ((Ra*vlloc[0]*alf_lloc[0]*(Tloc[0] - T_0))/((45.0 - 20)*H**3))
    *1.0
        Gloc[7] = ((Ra*vlloc[0]*alf_lloc[0]*(Tloc[0] - T_0))/((45.0 - 20)*H**3))
    *-1.0
        Gloc[8] = ((Ra*vlloc[0]*alf_lloc[0]*(Tloc[0] - T_0))/((45.0 - 20)*H**3))
    *-1.0
    else:
        Gloc[0]=0.0
        Gloc[1]=0.0
        Gloc[2]=0.0
        Gloc[3]=0.0
        Gloc[4]=0.0
        Gloc[5]=0.0
        Gloc[6]=0.0
        Gloc[7]=0.0
        Gloc[8]=0.0

# calculation of viscosity
def calc_Vloc(vxloc, vyloc, fl_PCMloc, nor_vloc, Gloc, floc):
    vxloc[0]=(floc[1]-floc[3]+floc[5]-floc[6]-floc[7]+floc[8])/1.0 + 0.5*fl_PCMloc
    [0]*(Gloc[5]-Gloc[6]+Gloc[7]-Gloc[8])
    vyloc[0]=(floc[2]-floc[4]+floc[5]+floc[6]-floc[7]-floc[8])/1.0 + 0.5*fl_PCMloc
    [0]*(Gloc[2]+Gloc[4]+Gloc[5]+Gloc[6]+Gloc[7]+Gloc[8])
    nor_vloc[0] = math.sqrt(vxloc[0]**2 + vyloc[0]**2)

# calculation of macroscopic velocity
def calc_Uloc(l_0loc, l_1loc, vxloc, vyloc, nor_vloc, uxloc, uyloc, nor_uloc,
    fl_PCMloc, i, j):
    uxloc[0]=vxloc[0]/(l_0loc[0] + math.sqrt(l_0loc[0]**2 + l_1loc[0]*nor_vloc[0]))
    uyloc[0]=vyloc[0]/(l_0loc[0] + math.sqrt(l_0loc[0]**2 + l_1loc[0]*nor_vloc[0]))

    if fl_PCMloc[0]==0: # liquid fraction of phase change material
        uxloc[0]=0.0
        uyloc[0]=0.0
    if i ==0 or j==0 or i==255 or j==255:
        uxloc[0]=0.0
        uyloc[0]=0.0

    nor_uloc[0] = math.sqrt(uxloc[0]**2 + uyloc[0]**2) # velocity norm

# calculation of force in all directions
def calc_Floc(fl_PCMloc, uxloc, uyloc, cfloc, vlloc, Gloc, TFloc, Fxloc, Fyloc, Ks,
     i, j):
    H = 256
    TFloc[0] = 0.0 # force in direction 0

    TFloc[1]=-(1.0/9.0)*((fl_PCMloc[0]*vlloc[0]/Ks[0])*(uxloc[0]*1.0 + uyloc
    [0]*0.0)+(fl_PCMloc[0]*cfloc[0]/math.sqrt(Ks[0]))*(abs(uxloc[0])*uxloc[0]*1.0+
    abs(uyloc[0])*uyloc[0]*0.0)) + fl_PCMloc[0]*Gloc[1] # force in direction 1

    TFloc[2]=-(1.0/9.0)*((fl_PCMloc[0]*vlloc[0]/Ks[0])*(uxloc[0]*0.0 + uyloc
    [0]*1.0)+(fl_PCMloc[0]*cfloc[0]/math.sqrt(Ks[0]))*(abs(uxloc[0])*uxloc[0]*0.0+
    abs(uyloc[0])*uyloc[0]*1.0)) + fl_PCMloc[0]*Gloc[2] # force in direction 2

    TFloc[3]=-(1.0/9.0)*((fl_PCMloc[0]*vlloc[0]/Ks[0])*(uxloc[0]*-1.0 + uyloc
    [0]*0.0)+(fl_PCMloc[0]*cfloc[0]/math.sqrt(Ks[0]))*(abs(uxloc[0])*uxloc[0]*-1.0+
    abs(uyloc[0])*uyloc[0]*0.0)) + fl_PCMloc[0]*Gloc[3] # force in direction 3

    TFloc[4]=-(1.0/9.0)*((fl_PCMloc[0]*vlloc[0]/Ks[0])*(uxloc[0]*0.0 + uyloc
    [0]*-1.0)+(fl_PCMloc[0]*cfloc[0]/math.sqrt(Ks[0]))*(abs(uxloc[0])*uxloc[0]*0.0+
    abs(uyloc[0])*uyloc[0]*-1.0)) + fl_PCMloc[0]*Gloc[4] # force in direction 4

    TFloc[5]=-(1.0/36.0)*((fl_PCMloc[0]*vlloc[0]/Ks[0])*(uxloc[0]*1.0 + uyloc
    [0]*1.0)+(fl_PCMloc[0]*cfloc[0]/math.sqrt(Ks[0]))*(abs(uxloc[0])*uxloc[0]*1.0+
    abs(uyloc[0])*uyloc[0]*1.0)) + fl_PCMloc[0]*Gloc[5] # force in direction 5

    TFloc[6]=-(1.0/36.0)*((fl_PCMloc[0]*vlloc[0]/Ks[0])*(uxloc[0]*-1.0 + uyloc
    [0]*1.0)+(fl_PCMloc[0]*cfloc[0]/math.sqrt(Ks[0]))*(abs(uxloc[0])*uxloc[0]*-1.0+
    abs(uyloc[0])*uyloc[0]*1.0)) + fl_PCMloc[0]*Gloc[6] # force in direction 6

    TFloc[7]=-(1.0/36.0)*((fl_PCMloc[0]*vlloc[0]/Ks[0])*(uxloc[0]*-1.0 + uyloc
    [0]*-1.0)+(fl_PCMloc[0]*cfloc[0]/math.sqrt(Ks[0]))*(abs(uxloc[0])*uxloc
    [0]*-1.0+abs(uyloc[0])*uyloc[0]*-1.0)) + fl_PCMloc[0]*Gloc[7] # force in 7

    TFloc[8]=-(1.0/36.0)*((fl_PCMloc[0]*vlloc[0]/Ks[0])*(uxloc[0]*1.0 + uyloc
    [0]*-1.0)+(fl_PCMloc[0]*cfloc[0]/math.sqrt(Ks[0]))*(abs(uxloc[0])*uxloc[0]*1.0+
    abs(uyloc[0])*uyloc[0]*-1.0)) + fl_PCMloc[0]*Gloc[8] # force in direction 8

# force in X and Y directions
    Fxloc[0]= TFloc[1]-TFloc[3]+TFloc[5]-TFloc[6]-TFloc[7]+TFloc[8] # X
    Fyloc[0]= TFloc[2]-TFloc[4]+TFloc[5]+TFloc[6]-TFloc[7]-TFloc[8] # Y
    if i==0 or j==0 or i==H or j==H:
        Fxloc[0] = 0.0
        Fyloc[0] = 0.0

def setvar2D(d_den, denloc, i, j): # export density as an array
    d_den[i,j]=denloc[0]
```

In the Section E of this code we execute the collision step for the temperature field as follows.

$$\mathbf{n}^{+}(\mathbf{x},t)=\mathbf{n}(\mathbf{x},t)-\mathbf{\Theta}\left[\mathbf{n}(\mathbf{x},t)-\mathbf{n}^{(eq)}(\mathbf{x},t)\right]+\delta_t\tilde{\mathbf{S}}.$$

Again, we need to perform a linear transformation from velocity space to moment space:

$$n=Ng$$

where

$$\mathbf{N}=\begin{pmatrix}1&1&1&1&1\\0&1&0&-1&0\\0&0&1&0&-1\\-4&1&1&1&1\\0&1&-1&1&-1\end{pmatrix}.$$

With the corresponding D2Q5 equilibrium distribution function.

$$\mathbf{n}^{(eq)}=\begin{pmatrix}T\\u_xT/\sigma\\u_yT/\sigma\\\tilde{\omega}T\\0\end{pmatrix}.$$

The relaxation parameter used in this code is

$$\tau_T=0.5+\alpha_e/\left(\sigma C_{sT}^2\delta_t\right),$$

where

$$\alpha_e = \alpha_l f_l + \alpha_s(1 - f_l).$$

The source term vector is now

$$\tilde{S}_0 = -\frac{\phi La}{\sigma C_{pl}}\frac{\Delta f_l}{\delta_t}, \quad \tilde{S}_1 = 0, \quad \tilde{S}_2 = 0, \quad \tilde{S}_3 = -\varpi\frac{\phi La}{\sigma C_{pl}}\frac{\Delta f_l}{\delta_t}, \quad \tilde{S}_4 = 0.$$

Finally the linear transformation of the temperature distribution function from moment space to velocity space is

$$g = N^{-1} n$$

```
                    #__________ Section E __________#
def getf_2l(d_f2l, f_2lloc, i, j): # obtain liquid fraction
    f_2lloc[0] = d_f2l[i,j]

def getg(d_g, gloc, i, j):          # obtain distribution function
    gloc[0] = d_g[i, j, 0]
    gloc[1] = d_g[i, j, 1]
    gloc[2] = d_g[i, j, 2]
    gloc[3] = d_g[i, j, 3]
    gloc[4] = d_g[i, j, 4]

# transformation of distribution function from velocity space to moment space
def g2n(nloc, gloc):
    nloc[0] = gloc[0] + gloc[1] + gloc[2] + gloc[3] + gloc[4]
    nloc[1] = gloc[1] - gloc[3]
    nloc[2] = gloc[2] - gloc[4]
    nloc[3] = -4.0*gloc[0] + gloc[1] + gloc[2] + gloc[3] + gloc[4]
    nloc[4] = gloc[1] - gloc[2] + gloc[3] - gloc[4]

# calculation of equilibrium distribution function in moment space
def calc_neqloc(Tloc, uxloc, uyloc, neqloc):
    w_test = -2.0
    if Tloc[0] >= 29.78:
        neqloc[0] = Tloc[0]
        neqloc[1] = uxloc[0]*Tloc[0]/0.8604      #sigma = 0.8352 in the liquid
        neqloc[2] = uyloc[0]*Tloc[0]/0.8604
        neqloc[3] = w_test*Tloc[0]
        neqloc[4] = 0.0
    else:
        neqloc[0] = Tloc[0]
        neqloc[1] = uxloc[0]*Tloc[0]/0.8352      #sigma = 0.8352 in the solid
        neqloc[2] = uyloc[0]*Tloc[0]/0.8352
        neqloc[3] = w_test*Tloc[0]
        neqloc[4] = 0.0

def calc_tautloc(tautloc, Tloc, f_lloc): # tau is a relaxation parameter
    tau_v = 0.50544816327342
    Lambda = 0.2719
    c_s = 1.0/math.sqrt(3.0)
    c_st = math.sqrt(1.0/5.0)
    Pr = 0.0208    # Prandtl number
    if Tloc[0] < 29.78:
        tautloc[0] = 0.5 + (Lambda*c_s**2*(tau_v-0.5))/(1.0*0.8352*c_st**2.0 * Pr)
    else:
        tautloc[0] = 0.5 + (Lambda*c_s**2*(tau_v-0.5))/(1.0*0.8604*c_st**2.0 * Pr)

def calc_relaxloc(relaxloc, tautloc): # calculation of relaxation vector
    relaxloc[0] = 1.0
    relaxloc[1] = 1.0/tautloc[0]
    relaxloc[2] = 1.0/tautloc[0]
    relaxloc[3] = 1.5
    relaxloc[4] = 1.5

# vector of source term
def calc_Ssurceloc(Ssurceloc, f_lloc, f_2lloc, Tloc, porosloc):
    Cpl = 1.0
    La = Cpl*(45.0 - 20.0)/0.1241    # La = Cpl*(T_h-T_c)/St
    w_test = -2.00
    if Tloc[0] > 29.78:              # liquid region
```

```
        Ssurceloc[0] = -((porosloc[0]*La)/(0.8604*Cpl))*(f_1loc[0] - f_2lloc[0])
    /1.0
        Ssurceloc[1] = 0.0
        Ssurceloc[2] = 0.0
        Ssurceloc[3] = -w_test*((porosloc[0]*La)/(0.8604*Cpl))*(f_1loc[0] - f_2lloc
    [0])/1.0
        Ssurceloc[4] = 0.0
    else:                              # solid region
        Ssurceloc[0] = -((porosloc[0]*La)/(0.8352*Cpl))*(f_1loc[0] - f_2lloc[0])
    /1.0
        Ssurceloc[1] = 0.0
        Ssurceloc[2] = 0.0
        Ssurceloc[3] = -w_test*((porosloc[0]*La)/(0.8352*Cpl))*(f_1loc[0] - f_2lloc
    [0])/1.0
        Ssurceloc[4] = 0.0

# collision step of temperature field
def colis_g(nloc, neqloc, Ssurceloc, relaxloc):
    nloc[0] = nloc[0] - relaxloc[0]*(nloc[0] - neqloc[0]) + Ssurceloc[0]
    nloc[1] = nloc[1] - relaxloc[1]*(nloc[1] - neqloc[1]) + Ssurceloc[1]
    nloc[2] = nloc[2] - relaxloc[2]*(nloc[2] - neqloc[2]) + Ssurceloc[2]
    nloc[3] = nloc[3] - relaxloc[3]*(nloc[3] - neqloc[3]) + Ssurceloc[3]
    nloc[4] = nloc[4] - relaxloc[4]*(nloc[4] - neqloc[4]) + Ssurceloc[4]

# linear transformation of distribution function from moment space to velocity
    space
def n2g(gloc, nloc):
    gloc[0] = 0.2*nloc[0] - 0.2*nloc[3]
    gloc[1] = 0.2*nloc[0] + 0.5*nloc[1] + 0.05*nloc[3] + 0.25*nloc[4]
    gloc[2] = 0.2*nloc[0] + 0.5*nloc[2] + 0.05*nloc[3] - 0.25*nloc[4]
    gloc[3] = 0.2*nloc[0] - 0.5*nloc[1] + 0.05*nloc[3] + 0.25*nloc[4]
    gloc[4] = 0.2*nloc[0] - 0.5*nloc[2] + 0.05*nloc[3] - 0.25*nloc[4]

def setg(d_g, gloc, i, j): # export distribution function as an array
    d_g[i, j, 0] = gloc[0]
    d_g[i, j, 1] = gloc[1]
    d_g[i, j, 2] = gloc[2]
    d_g[i, j, 3] = gloc[3]
    d_g[i, j, 4] = gloc[4]
```

In the section F of the code we calculate the macroscopic variables related to the temperature field executed in local memory.

1. Temperature:

$$T = \sum_{i=0}^{4} g_i.$$

2. Enthalpy:

$$H = C_p T + f_l L_a.$$

3. Liquid fraction

$$f_l = \begin{cases} 0, & \text{if } H_k \leq H_s, \\ \frac{H_k - H_s}{H_l - H_s}, & \text{if } H_s < H_k < H_l, \\ 1, & \text{if } H_k \geq H_l, \end{cases} .$$

```
                         #__________ Section F __________#
def calc_Tloc(gloc, Tloc):          # calculation of temperature
    Tloc[0] = gloc[0] + gloc[1] + gloc[2] + gloc[3] + gloc[4]

def calc_cpfl(f_1loc, f_2lloc):
    f_2lloc[0] = f_1loc[0]

def calc_Hk(Tloc, Hkloc, f_1loc):  # calculation of enthalpy
    Cps = 1.0    #calor especifico del solido y calor latente
    La = Cps*(45.0 - 20.0)/0.1241  # La = Cpl*(T_h-T_c)/St
    Hkloc[0] = Cps*Tloc[0] + f_1loc[0]*La

def calc_fl(f_1loc, Hkloc):         # calculation of liquid fraction
```

```
    T_m = 29.78
    La = 1.0*(45.0 - 20.0)/0.1241 # La = Cpl*(T_h-T_c)/St
    Hl = 1.0*(T_m+0.5) + 1.0*La # entalpia del liquido
    Hs = 1.0*(T_m-0.5) # entalpia del solido  Ts=-0.02
    if (Hkloc[0] <= Hs):
        f_lloc[0] = 0.0
    elif ((Hkloc[0] > Hs) and (Hkloc[0] < Hl)):
        f_lloc[0] = (Hkloc[0] - Hs)/(Hl - Hs)
    else:
        f_lloc[0] = 1.0
```

### 1.4.4 Graphics.

In this section we show the code to obtain the corresponding graphics. this code shows the stream lines and temperature field in the same graphic and can be edited for the reader by other preferences.

```
                      #_________ Section G _________#
import numpy as np
from StringIO import StringIO
import matplotlib.pyplot as plt

# import the text files (.txt) to save solution as a matrix
pfile=open('T.txt','r')
pfile_1=open('vel_ux.txt','r')
pfile_2=open('vel_uy.txt','r')
pfile_3=open('den.txt','r')

# read and transform from text to a matrix
data=pfile.read()
pfile.close()
T=np.genfromtxt(StringIO(data))   # temperature

data_1=pfile_1.read()
pfile_1.close()
u=np.genfromtxt(StringIO(data_1)) # velocity in x direction

data_2=pfile_2.read()
pfile_2.close()
v=np.genfromtxt(StringIO(data_2)) # velocity in y direction

data_3=pfile_3.read()
pfile_3.close()
rho=np.genfromtxt(StringIO(data_3)) # density

# obtain the shape of the matrix, then the stream lines are obtained (strf).
n, m = rho.shape
strf=np.zeros([n,m])
strf[0,0]=0.0
for j in range(m):
    rhoav=0.5*(rho[j-1,0]+rho[j,0])
    if j != 0.0: strf[j,0] = strf[j-1,0]-rhoav*0.5*(v[j-1,0]+v[j,0])
    for i in range(1,n):
        rhom=0.5*(rho[j,i]+rho[j,i-1])
        strf[j,i]=strf[j,i-1]+rhom*0.5*(u[j,i-1]+u[j,i])

strf = np.transpose(strf)
T = np.transpose(T)

strf_=np.zeros([n,m])
T_=np.zeros([n,m])
for j in range(m):
    k=0
    for i in range(n-1,-1,-1):
        strf_[k,j] = abs(strf[i,j])  # strf is the stream lines matrix
        T_[k,j] = T[i,j]
        k=k+1
xlist = np.linspace(0, n, 256)  # x-size in domain
ylist = np.linspace(0, m, 256)  # y-size in domain
X, Y = np.meshgrid(xlist, ylist)# mesh dimension
```

```
plt.figure()
cp = plt.contour(X, Y, strf_)   # plot
plt.clabel(cp, inline=True,
          fontsize=9)

fig = plt.figure(1)
#plt.title('Temperature and stream lines')
plt.xlabel('X')
plt.ylabel('Y')
cs1 = plt.contourf((T_), 500)
plt.colorbar()
plt.savefig("strf.jpg")      # export image as .jpg format
#plt.savefig("strf.eps")     # export image as postscript format
plt.show()                   # Show the image
```

## 1.5 Performance Tuning

Melting_Ga uses default settings that should make most simulations run at a reasonable speed in modern GPUs. There are however several tunable parameters that can be used to improve the performance of specific simulations. Finding the optimal values of these parameters requires some experimentation on a case-by-case basis.

### 1.5.1 General tuning strategies

**Adjusting the block size**

This is the simplest optimization you should apply when trying to increase the speed of a simulation. Every lattice Boltzmann node in Melting_Ga is processed in a separate GPU thread. These threads are grouped into 2-dimensional blocks. You can adjust this to a different value:

```
threads = 256,1          #1024#512
    blocks = (lattice_x/threads[0]+(0!=lattice_x%threads[0]),
              lattice_y/threads[1]+(0!=lattice_y%threads[1]) )
```

## 1.6 Examples

Several simulations are included in the Melting_Ga code repository as examples. They serve to illustrate the capabilities of the solver, as well as to test the correctness of the simulations by comparing their results to data from the literature. Every item is a different example, where the user can change the parameters and perform a different simulation.

1. Case 1: Phase change simulation of a simple bar of a solid material (caused by thermal phenomena), the results can be verified in [5].

    - Serial implementation performed only by Numpy.
    - Serial implementation performed by Numba and features of Numpy (explained in this tutorial).
    - Parallel implementation performed by Numba-CUDA, streaming step, boundary conditions and calculation of macroscopic variables are performed in shares memory.
    - Parallel implementation performed by Numba-CUDA, collision step and calculation of macroscopic variables are performed in local memory (explained in this tutorial).
    - Comparison of the analytical solution (when possible) and numerics, mostly via graphics.

2. Case 2: Phase change simulation of a material (in our case Ga) immersed in a porous media, performed by Numba-CUDA, this simulation uses MRLBM (with unitary stencil D2Q5) to solve temperature field and MRLBM (with unitary stencil D2Q9) to solve velocity field. Natural convection is taken into account via the Boussinesq approach.

    - Simulation with homogeneous porosity, $\phi = 0.368$.
    - Simulation with acquisition of porous media through an image (see Fig. 6), where the porosity is calculated in all subregions. This is explained in this tutorial.

## References

[1] C. Beckermann and R. Viskanta. Natural convection solid/liquid phase change in porous media. *International Journal of Heat and Mass Transfer*, 31(1):35–46, January 1988.

[2] Wen-Jeng Chang and Dong-Fang Yang. Natural convection for the melting of ice in porous media in a rectangular enclosure. *International Journal of Heat and Mass Transfer*, 39(11):2333–2348, July 1996.

[3] Mohsen Eshraghi and Sergio D. Felicelli. An implicit lattice Boltzmann model for heat conduction with phase change. *International Journal of Heat and Mass Transfer*, 55(9):2420–2428, April 2012.

[4] A. A. Mohamad. *Lattice Boltzmann Method: Fundamentals and Engineering Applications with Computer Codes*. Springer-Verlag, London, 2011.

[5] Qing Liu and Ya-Ling He. Double multiple-relaxation-time lattice Boltzmann model for solid–liquid phase change with natural convection in porous media. *Physica A: Statistical Mechanics and its Applications*, 438(Supplement C):94–106, November 2015.

[6] Hiroaki Yoshida and Makoto Nagaoka. Multiple-relaxation-time lattice Boltzmann model for the convection and anisotropic diffusion equation. *Journal of Computational Physics*, 229(20):7774–7795, October 2010.

[7] 2.1. Types and signatures — Numba 0.37.0+189.g35d2977-py2.7-linux-x86_64.egg documentation.

[8] 42ummemfilt.gif (512×512).