pylbm Documentation

Release 0.4.1

Benjamin Graille, Loïc Gouarin

Jun 24, 2019

CONTENTS

pylbm is an all-in-one package for numerical simulations using Lattice Boltzmann solvers.

This package gives all the tools to describe your lattice Boltzmann scheme in 1D, 2D and 3D problems.

We choose the D'Humières formalism to describe the problem. You can have complex geometry with a set of simple shape like circle, sphere, ...

pylbm performs the numerical scheme using Cython, NumPy or Loo.py from the scheme and the domain given by the user. Pythran and Numba wiil be available soon. pylbm has MPI support with mpi4py.

You can install pylbm in several ways

With conda

```
conda install pylbm -c conda-forge
```

With Pypi

```
pip install pylbm
```

or

```
pip install pylbm --user
```

From source

You can also clone the project and install the latest version

```
git clone https://github.com/pylbm/pylbm
```

To install pylbm from source, we encourage you to create a fresh environment using conda.

```
conda create -n pylbm_env python=3.6
```

As mentioned at the end of the creation of this environment, you can activate it using the comamnd line

```
conda activate pylbm_env
```

Now, you just have to go into the pylbm directory that you cloned and install the dependencies

```
conda install --file requirements-dev.txt -c conda-forge
```

and then, install pylbm

```
python setup.py install
```

CHAPTER

ONE

GETTING STARTED

pylbm can be a simple way to make numerical simulations by using the Lattice Boltzmann method.

Once the package is installed you just have to understand how to build a dictionary that will be understood by pylbm to perform the simulation. The dictionary should contain all the needed informations as

- the geometry (see *here* for documentation)
- the scheme (see *here* for documentation)
- another informations like the space step, the scheme velocity, the generator of the functions...

To understand how to use pylbm, you have a lot of Python notebooks in the tutorial.

CHAPTER

TWO

DOCUMENTATION FOR USERS

2.1 The Geometry of the simulation

With pylbm, the numerical simulations can be performed in a domain with a complex geometry. This geometry is construct without considering a particular mesh but only with geometrical objects. All the geometrical informations are defined through a dictionary and put into an object of the class *Geometry*.

First, the domain is put into a box: a segment in 1D, a rectangle in 2D, and a rectangular parallelepipoid in 3D.

Then, the domain is modified by adding or deleting some elementary shapes. In 2D, the elementary shapes are

- a *Circle*
- an *Ellipse*
- a *Parallelogram*
- a *Triangle*

From version 0.2, the geometrical elements are implemented in 3D. The elementary shapes are

- a *Sphere*
- an *Ellipsoid*
- a *Parallelepiped*
- a Cylinder with a 2D-base
  - *Cylinder (Circle)*
  - *Cylinder (Ellipse)*
  - *Cylinder (Triangle)*

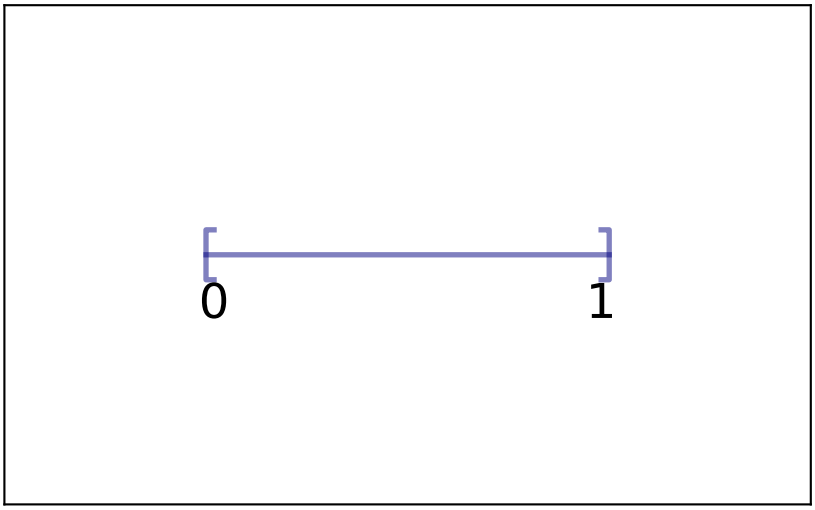
Several examples of geometries can be found in demo/examples/geometry/

2.1.1 Examples in 1D

`script`

The segment $[0, 1]$

```
d = {'box':{'x': [0, 1], 'label': [0, 1]}}
g = pylbm.Geometry(d)
g.visualize(viewlabel = True)
```



The segment $[0, 1]$ is created by the dictionary with the key `box`. We then add the labels 0 and 1 on the edges with the key `label`. The result is then visualized with the labels by using the method *visualize*. If no labels are given in the dictionary, the default value is -1.

2.1.2 Examples in 2D

`script`

The square $[0, 1]^2$

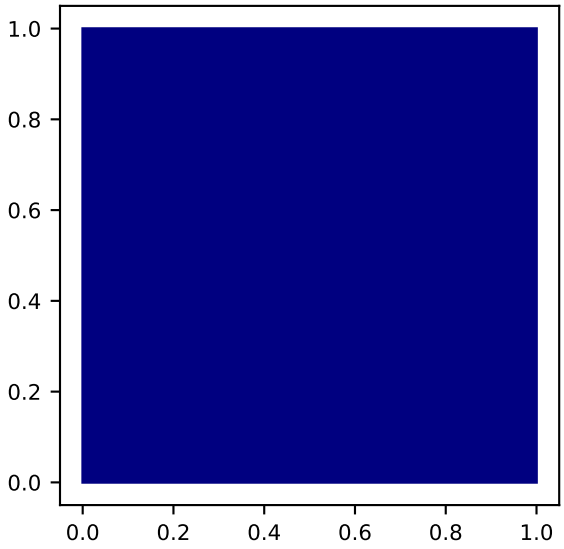
```
d = {'box':{'x': [0, 1], 'y': [0, 1]}}
g = pylbm.Geometry(d)
g.visualize()
```

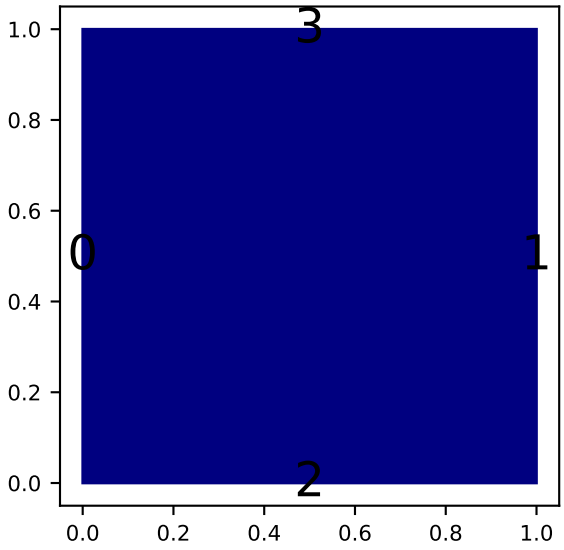
The square $[0, 1]^2$ is created by the dictionary with the key `box`. The result is then visualized by using the method *visualize*.

We then add the labels on each edge of the square through a list of integers with the conventions:

- first for the left ($x = x_{\min}$)
- third for the bottom ($y = y_{\min}$)
- second for the right ($x = x_{\max}$)
- fourth for the top ($y = y_{\max}$)

```
d = {'box':{'x': [0, 1], 'y': [0, 1], 'label':[0, 1, 2, 3]}}
g = pylbm.Geometry(d)
g.visualize(viewlabel = True)
```


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


3
0
1
2
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

If all the labels have the same value, a shorter solution is to give only the integer value of the label instead of the list. If no labels are given in the dictionary, the default value is -1.

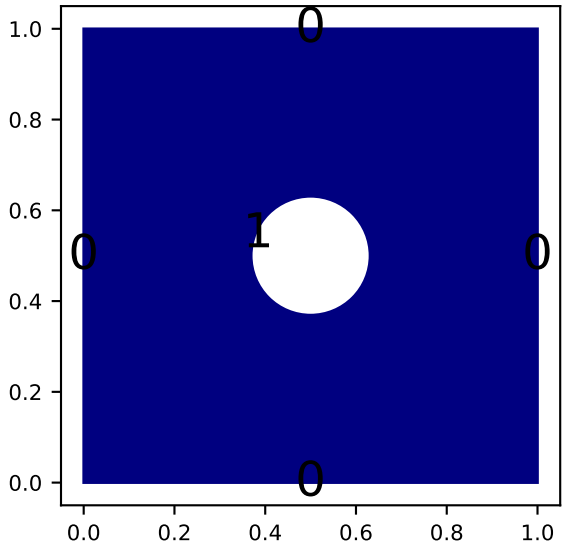
```
script 3 script 2 script 1
```

A square with a hole

The unit square $[0,1]^2$ can be holed with a circle (script 1) or with a triangular or with a parallelogram (script 3)

In the first example, a solid disc lies in the fluid domain defined by a *circle* with a center of (0.5, 0.5) and a radius of 0.125

```
d = {'box':{'x': [0, 1], 'y': [0, 1], 'label':0},
     'elements':[pylbm.Circle((.5, .5), .125, label = 1)],
}
g = pylbm.Geometry(d)
g.visualize(viewlabel=True)
```

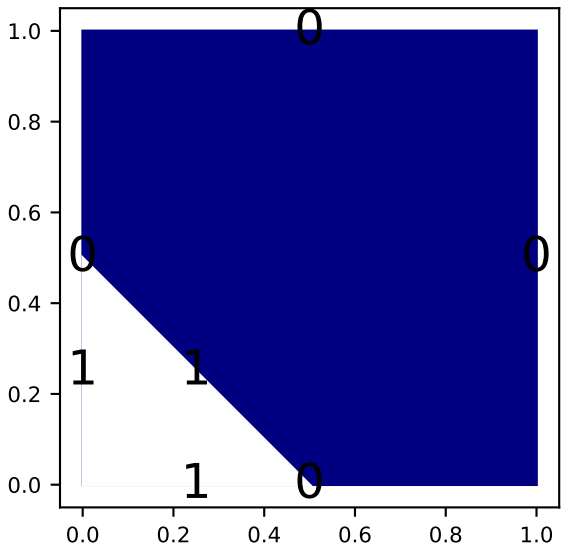


The dictionary of the geometry then contains an additional key `elements` that is a list of elements. In this example, the circle is labelized by 1 while the edges of the square by 0.

The element can be also a *triangle*

```
d = {'box':{'x': [0, 1], 'y': [0, 1], 'label':0},
      'elements':[pylbm.Triangle((0.,0.), (0.,.5), (.5, 0.), label = 1)],
}
g = pylbm.Geometry(d)
g.visualize(viewlabel=True)
```

or a *parallelogram*




```
d = {'box':{'x': [0, 3], 'y': [0, 1], 'label':[1, 2, 0, 0]},
     'elements':[pylbm.Parallelogram((0.,0.), (.5,0.), (0., .5), label = 0)],
}
g = pylbm.Geometry(d)
g.visualize()
```

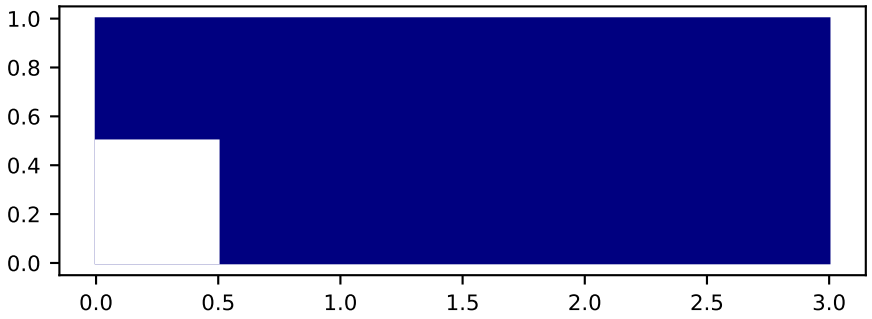
`script`

A complex cavity

A complex geometry can be build by using a list of elements. In this example, the box is fixed to the unit square $[0, 1]^2$. A square hole is added with the argument `isfluid=False`. A strip and a circle are then added with the argument `isfluid=True`. Finally, a square hole is put. The value of `elements` contains the list of all the previous elements. Note that the order of the elements in the list is relevant.

```
square = pylbm.Parallelogram((.1, .1), (.8, 0), (0, .8), isfluid=False)
strip = pylbm.Parallelogram((0, .4), (1, 0), (0, .2), isfluid=True)
circle = pylbm.Circle((.5, .5), .25, isfluid=True)
inner_square = pylbm.Parallelogram((.4, .5), (.1, .1), (.1, -.1), isfluid=False)
d = {'box':{'x': [0, 1], 'y': [0, 1], 'label':0},
     'elements':[square, strip, circle, inner_square],
}
g = pylbm.Geometry(d)
g.visualize()
```

Once the geometry is built, it can be modified by adding or deleting other elements. For instance, the four corners of the cavity can be rounded in this way.




```
g.add_elem(pylbm.Parallelogram((0.1, 0.9), (0.05, 0), (0, -0.05), isfluid=True))
g.add_elem(pylbm.Circle((0.15, 0.85), 0.05, isfluid=False))
g.add_elem(pylbm.Parallelogram((0.1, 0.1), (0.05, 0), (0, 0.05), isfluid=True))
g.add_elem(pylbm.Circle((0.15, 0.15), 0.05, isfluid=False))
g.add_elem(pylbm.Parallelogram((0.9, 0.9), (-0.05, 0), (0, -0.05), isfluid=True))
g.add_elem(pylbm.Circle((0.85, 0.85), 0.05, isfluid=False))
g.add_elem(pylbm.Parallelogram((0.9, 0.1), (-0.05, 0), (0, 0.05), isfluid=True))
g.add_elem(pylbm.Circle((0.85, 0.15), 0.05, isfluid=False))
g.visualize()
```

2.1.3 Examples in 3D

`script`

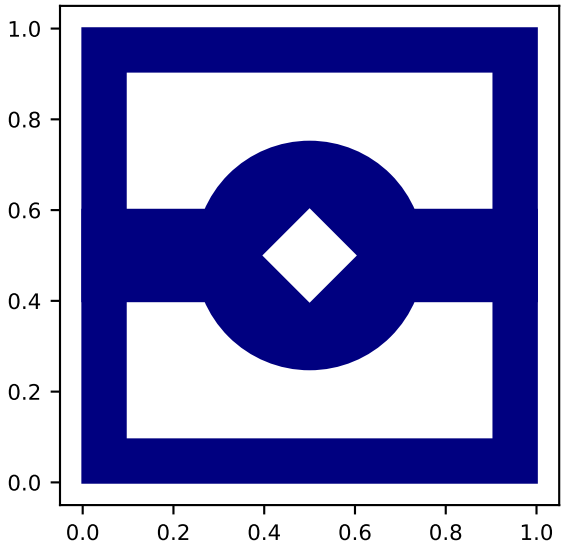
The cube $[0,1]^3$

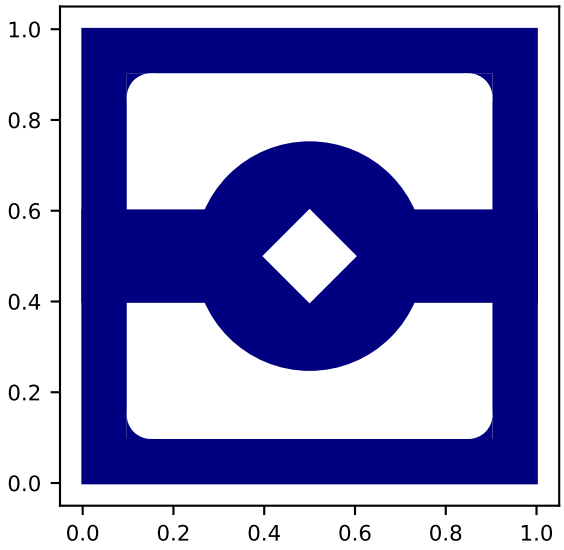
```
d = {'box':{'x': [0, 1], 'y': [0, 1], 'z':[0, 1], 'label':list(range(6))}}
g = pylbm.Geometry(d)
g.visualize(viewlabel=True)
```

The cube $[0,1]^3$ is created by the dictionary with the key `box`. The result is then visualized by using the method *`visualize`*.

We then add the labels on each edge of the square through a list of integers with the conventions:

- first for the left ($x = x_{\min}$)
- third for the bottom ($y = y_{\min}$)


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

- fifth for the front ($z = z_{\min}$)
- second for the right ($x = x_{\max}$)
- fourth for the top ($y = y_{\max}$)
- sixth for the back ($z = z_{\max}$)

If all the labels have the same value, a shorter solution is to give only the integer value of the label instead of the list. If no labels are given in the dictionary, the default value is -1.

The cube $[0, 1]^3$ with a hole

```
d = {
    'box':{'x': [0, 1], 'y': [0, 1], 'z':[0, 1], 'label':0},
    'elements':[pylbm.Sphere((.5,.5,.5), .25, label=1)],
}
g = pylbm.Geometry(d)
g.visualize(viewlabel=True)
```

The cube $[0, 1]^3$ and the spherical hole are created by the dictionary with the keys `box` and `elements`. The result is then visualized by using the method *visualize*.

2.2 The Domain of the simulation

With pylbm, the numerical simulations can be performed in a domain with a complex geometry. The creation of the geometry from a dictionary is explained here. All the informations needed to build the domain are defined through a dictionary and put in a object of the class *Domain*.

The domain is built from three types of informations:

- a geometry (class *Geometry*),
- a stencil (class *Stencil*),
- a space step (a float for the grid step of the simulation).

The domain is a uniform cartesian discretization of the geometry with a grid step dx. The whole box is discretized even if some elements are added to reduce the domain of the computation. The stencil is necessary in order to know the maximal velocity in each direction so that the corresponding number of phantom cells are added at the borders of the domain (for the treatment of the boundary conditions). The user can get the coordinates of the points in the domain by the fields `x`, `y`, and `z`. By convention, if the spatial dimension is one, `y=z=None`; and if it is two, `z=None`.

Several examples of domains can be found in demo/examples/domain/

2.2.1 Examples in 1D

`script`

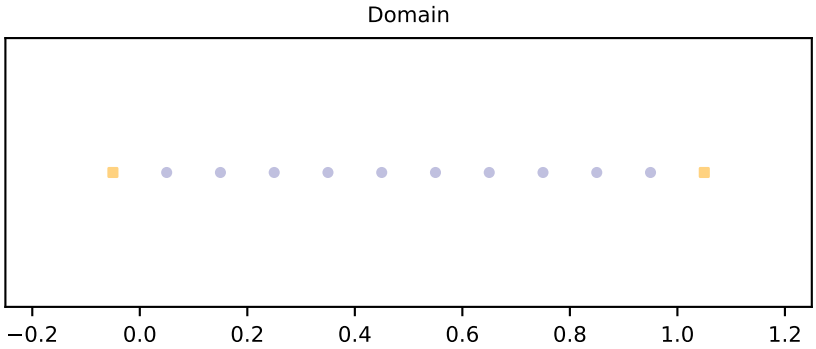
The segment $[0, 1]$ with a D_1Q_3

```
dico = {
    'box':{'x': [0, 1], 'label':0},
    'space_step':0.1,
    'schemes':[{'velocities':list(range(3))}],
}
```

(continues on next page)

(continued from previous page)

```
dom = pylbm.Domain(dico)
dom.visualize()
```



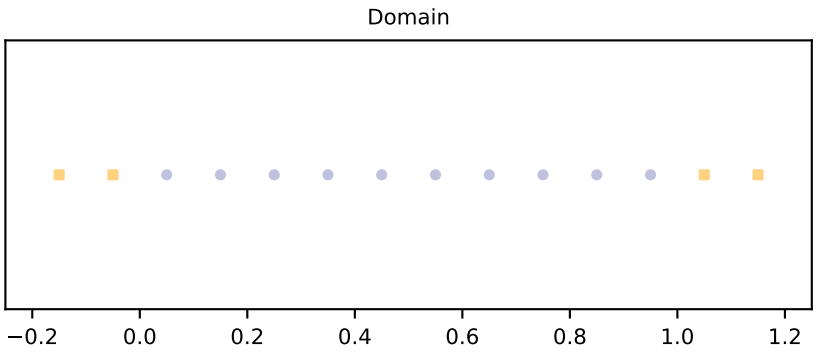
The segment $[0, 1]$ is created by the dictionary with the key `box`. The stencil is composed by the velocity $v_0 = 0$, $v_1 = 1$, and $v_2 = -1$. One phantom cell is then added at the left and at the right of the domain. The space step dx is taken to 0.1 to allow the visualization. The result is then visualized with the distance of the boundary points by using the method *visualize*.

`script`

The segment $[0, 1]$ with a D_1Q_5

```
dico = {
    'box':{'x': [0, 1], 'label':0},
    'space_step':0.1,
    'schemes':[{'velocities':list(range(5))}],
}
dom = pylbm.Domain(dico)
dom.visualize()
```

The segment $[0, 1]$ is created by the dictionary with the key `box`. The stencil is composed by the velocity $v_0 = 0$, $v_1 = 1$, $v_2 = -1$, $v_3 = 2$, $v_4 = -2$. Two phantom cells are then added at the left and at the right of the domain. The space step dx is taken to 0.1 to allow the visualization. The result is then visualized with the distance of the boundary points by using the method *visualize*.



2.2.2 Examples in 2D

`script`

The square $[0,1]^2$ with a D_2Q_9

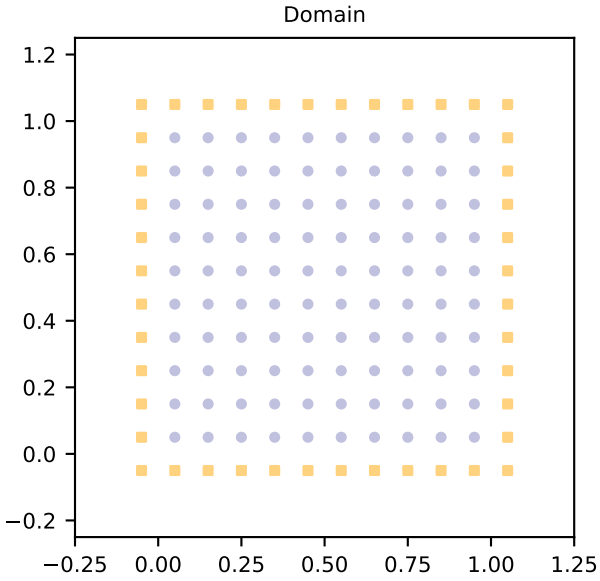
```
dico = {
    'box':{'x': [0, 1], 'y': [0, 1], 'label':0},
    'space_step':0.1,
    'schemes':[{'velocities':list(range(9))}],
}
dom = pylbm.Domain(dico)
dom.visualize()
dom.visualize(view_distance=True)
```

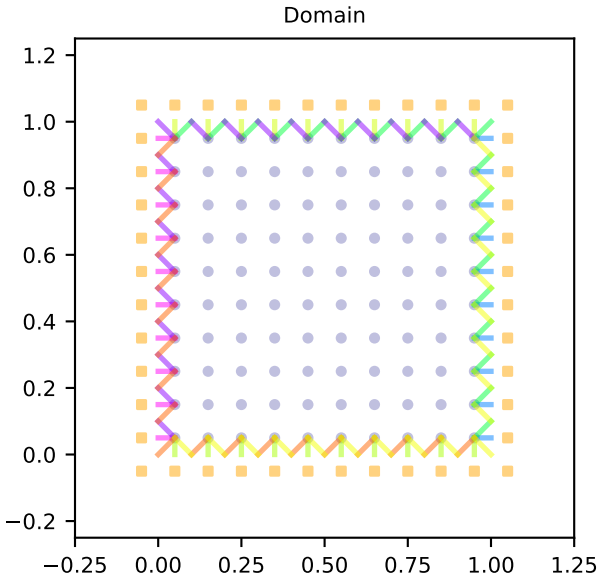
The square $[0,1]^2$ is created by the dictionary with the key `box`. The stencil is composed by the nine velocities

$$\begin{aligned} v_0 &= (0,0), \\ v_1 = (1,0), v_2 &= (0,1), v_3 = (-1,0), v_4 = (0,-1), \\ v_5 = (1,1), v_6 &= (-1,1), v_7 = (-1,-1), v_8 = (1,-1). \end{aligned} \tag{2.1}$$

One phantom cell is then added all around the square. The space step dx is taken to 0.1 to allow the visualization. The result is then visualized by using the method *visualize*. This method can be used without parameter: the domain is visualize in white for the fluid part (where the computation is done) and in black for the solid part (the phantom cells or the obstacles). An optional parameter view_distance can be used to visualize more precisely the points (a black circle inside the domain and a square outside). Color lines are added to visualize the position of the border: for each point that can reach the border for a given velocity in one time step, the distance to the border is computed.

`script 1`

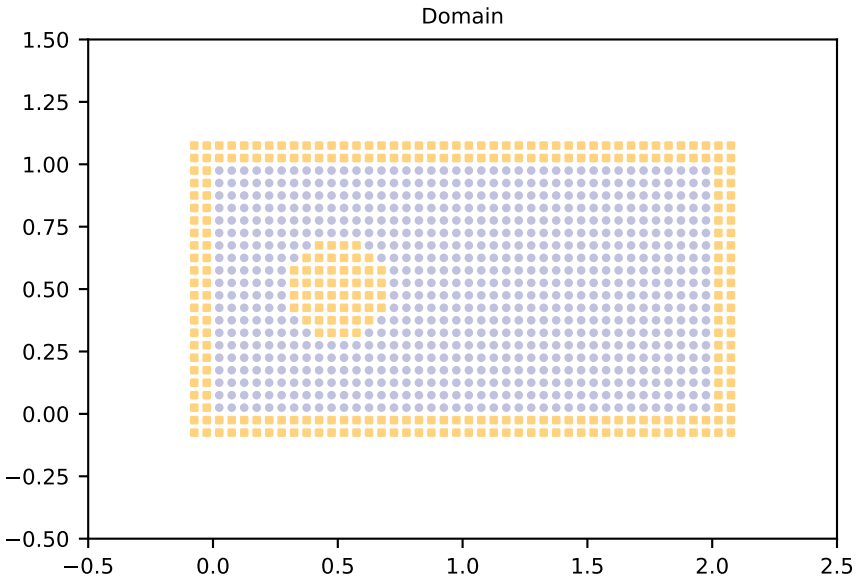

Domain
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.25
0.00
0.25
0.50
0.75
1.00
1.25


Domain
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.25
0.00
0.25
0.50
0.75
1.00
1.25

A square with a hole with a D_2Q_{13}

The unit square $[0,1]^2$ can be holed with a circle. In this example, a solid disc lies in the fluid domain defined by a `circle` with a center of (0.5, 0.5) and a radius of 0.125

```
dico = {
    'box':{'x': [0, 2], 'y': [0, 1], 'label':0},
    'elements':[pylbm.Circle((0.5,0.5), 0.2)],
    'space_step':0.05,
    'schemes':[{'velocities':list(range(13))}],
}
dom = pylbm.Domain(dico)
dom.visualize()
dom.visualize(view_distance=True)
```



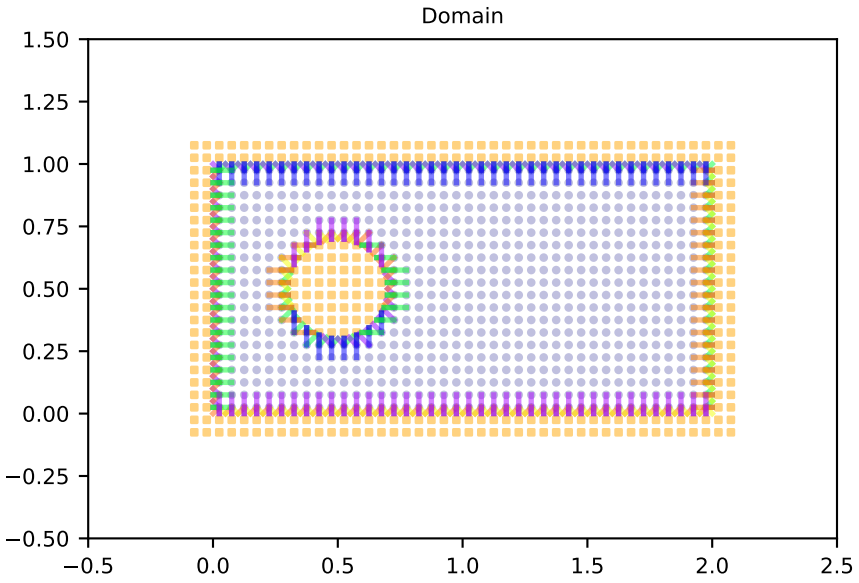
`script`

A step with a D_2Q_9

A step can be build by removing a rectangle in the left corner. For a D_2Q_9, it gives the following domain.

```
dico = {
    'box':{'x': [0, 3], 'y': [0, 1], 'label':0},
    'elements':[pylbm.Parallelogram((0.,0.), (.5,0.), (0., .5), label=1)],
    'space_step':0.125,
    'schemes':[{'velocities':list(range(9))}],
}
dom = pylbm.Domain(dico)
```

(continues on next page)



(continued from previous page)

```
dom.visualize()
dom.visualize(view_distance=True, label=1)
```

Note that the distance with the bound is visible only for the specified labels.

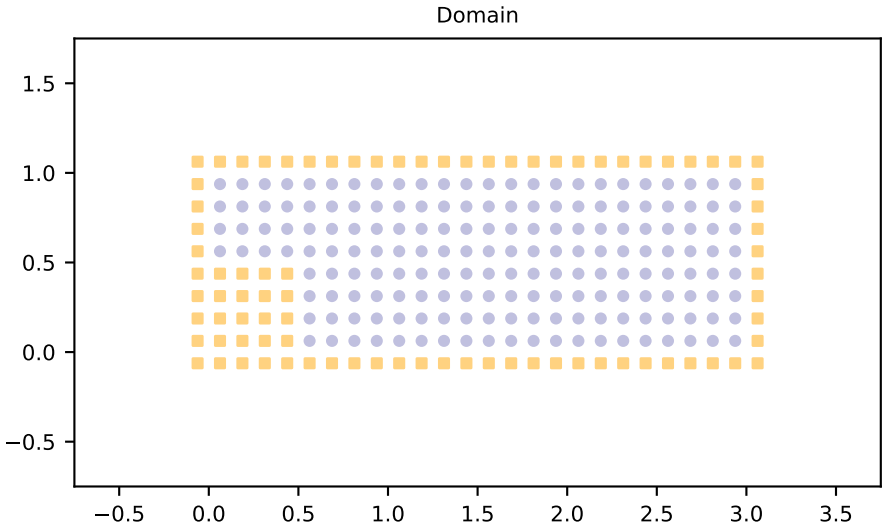
2.2.3 Examples in 3D

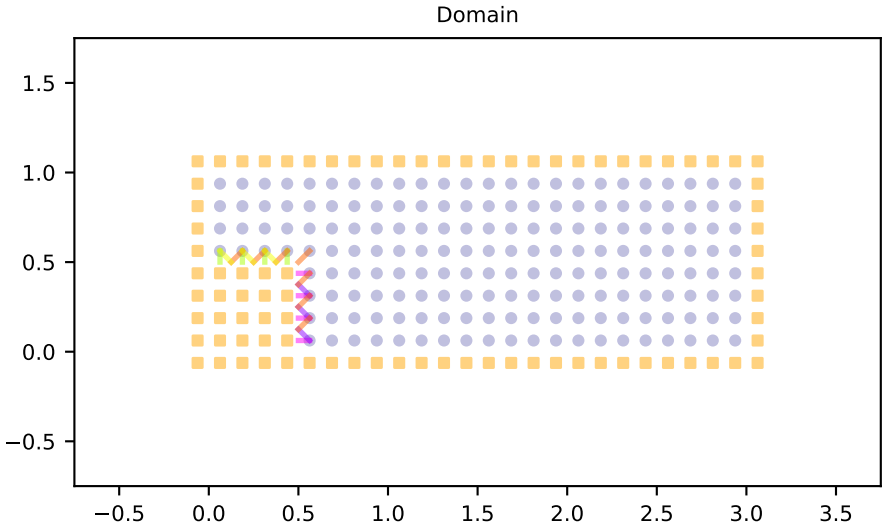
`script`

The cube $[0, 1]^3$ with a D_3Q_{19}

```
dico = {
    'box':{'x': [0, 2], 'y': [0, 2], 'z':[0, 2], 'label':0},
    'space_step':.5,
    'schemes':[{'velocities':list(range(19))}]
}
dom = pylbm.Domain(dico)
dom.visualize()
dom.visualize(view_distance=True)
```

The cube $[0, 1]^3$ is created by the dictionary with the key `box` and the first 19th velocities. The result is then visualized by using the method *`visualize`*.


Domain
1.5
1.0
0.5
0.0
−0.5
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5


Domain
1.5
1.0
0.5
0.0
−0.5
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5

The cube with a hole with a D_3Q_{19}

```
dico = {
    'box':{'x': [0, 2], 'y': [0, 2], 'z':[0, 2], 'label':0},
    'elements':[pylbm.Sphere((1,1,1), 0.5, label = 1)],
    'space_step':.5,
    'schemes':[{'velocities':list(range(19))}]
}
dom = pylbm.Domain(dico)
dom.visualize()
dom.visualize(view_distance=False, view_bound=True, label=1, view_in=False, view_
↪out=False)
```

2.3 The Scheme

With pylbm, elementary schemes can be gathered and coupled through the equilibrium in order to simplify the implementation of the vectorial schemes. Of course, the user can implement a single elementary scheme and then recover the classical framework of the d'Humières schemes.

For pylbm, the `scheme` is performed through a dictionary. The generalized d'Humières framework for vectorial schemes is used [dH92], [G14]. In the first section, we describe how build an elementary scheme. Then the vectorial schemes are introduced as coupled elementary schemes.

2.3.1 The elementary schemes

Let us first consider a regular lattice L in dimension d with a typical mesh size dx, and the time step dt. The scheme velocity λ is then defined by $\lambda = dx/dt$. We introduce a set of q velocities adapted to this lattice $\{v_0, \dots, v_{q-1}\}$, that is to say that, if x is a point of the lattice L, the point $x + v_j dt$ is on the lattice for every $j \in \{0, \dots, q-1\}$.

The aim of the $DdQq$ scheme is to compute a distribution function vector $\boldsymbol{f} = (f_0, \dots, f_{q-1})$ on the lattice L at discret values of time. The scheme splits into two phases: the relaxation and the transport. That is, the passage from the time t to the time $t + dt$ consists in the succession of these two phases.

- the relaxation phase

  This phase, also called collision, is local in space: on every site x of the lattice, the values of the vector $\boldsymbol{f}$ are modified, the result after the collision being denoted by $\boldsymbol{f}^\star$. The operator of collision is a linear operator of relaxation toward an equilibrium value denoted $\boldsymbol{f}^{\text{eq}}$.

  pylbm uses the framework of d'Humières: the linear operator of the collision is diagonal in a special basis called moments denoted by $\boldsymbol{m} = (m_0, \dots, m_{q-1})$. The change-of-basis matrix M is such that $\boldsymbol{m} = M\boldsymbol{f}$ and $\boldsymbol{f} = M^{-1}\boldsymbol{m}$. In the basis of the moments, the collision operator then just reads

  $$m_k^\star = m_k - s_k(m_k - m_k^{\text{eq}}), \qquad 0 \leqslant k \leqslant q-1,$$

  where s_k is the relaxation parameter associated to the kth moment. The kth moment is said conserved during the collision if the associated relaxation parameter $s_k = 0$.

  By analogy with the kinetic theory, the change-of-basis matrix M is defined by a set of polynomials with d variables $(P_0, \dots, P_{q-1})$ by

  $$M_{ij} = P_i(v_j).$$

- the transport phase

  This phase just consists in a shift of the indices and reads

  $$f_j(x, t + dt) = f_j^\star(x - v_j dt, t), \qquad 0 \leqslant j \leqslant q-1,$$

Notations

The `scheme` is defined and build through a dictionary in pylbm. Let us first list the several key words of this dictionary:

- `dim`: the spatial dimension. This argument is optional if the geometry is known, that is if the dimension can be computed through the list of the variables;
- `scheme_velocity`: the velocity of the scheme denoted by λ in the previous section and defined as the spatial step over the time step ($\lambda = dx/dt$) ;
- `schemes`: the list of the schemes. In pylbm, several coupled schemes can be used, the coupling being done through the equilibrium values of the moments. Some examples with only one scheme and with more than one schemes are given in the next sections. Each element of the list should be a dictionay with the following key words:
  - `velocities`: the list of the velocity indices,
  - `conserved_moments`: the list of the conserved moments (list of symbolic variables),
  - `polynomials`: the list of the polynomials that define the moments, the polynomials are built with the symbolic variables X, Y, and Z,
  - `equilibrium`: the list of the equilibrium value of the moments,
  - `relaxation_parameters`: the list of the relaxation parameters, (by convention, the relaxation parameter of a conserved moment is taken to 0).

Examples in 1D

`script`

D_1Q_2 for the advection

A velocity $c \in \mathbb{R}$ being given, the advection equation reads

$$\partial_t u(t,x) + c\partial_x u(t,x) = 0, \qquad t > 0, x \in \mathbb{R}.$$

Taken for instance $c = 0.5$, the following scheme can be used:

```
import sympy as sp
import pylbm
u, X = sp.symbols('u, X')

d = {
    'dim':1,
    'scheme_velocity':1.,
    'schemes':[
        {
            'velocities': [1, 2],
            'conserved_moments':u,
            'polynomials': [1, X],
            'equilibrium': [u, .5*u],
            'relaxation_parameters': [0., 1.9],
        },
    ],
}
s = pylbm.Scheme(d)
print(s)
```

The dictionary `d` is used to set the dimension to 1, the scheme velocity to 1. The used scheme has two velocities: the first one $v_0 = 1$ and the second one $v_1 = -1$. The polynomials that define the moments are $P_0 = 1$ and $P_1 = X$ so that the matrix of the moments is

$$M = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

with the convention $M_{ij} = P_i(v_j)$. Then, there are two distribution functions f_0 and f_1 that move at the velocities v_0 and v_1, and two moments $m_0 = f_0 + f_1$ and $m_1 = f_0 - f_1$. The first moment m_0 is conserved during the relaxation phase (as the associated relaxation parameter is set to 0), while the second moment m_1 relaxes to its equilibrium value $0.5m_0$ with a relaxation parameter 1.9 by the relation

$$m_1^\star = m_1 - 1.9(m_1 - 0.5m_0).$$

`script`

D_1Q_2 for Burger's

The Burger's equation reads

$$\partial_t u(t,x) + \tfrac{1}{2}\partial_x u^2(t,x) = 0, \qquad t > 0, x \in \mathbb{R}.$$

The following scheme can be used:

```
import sympy as sp
import pylbm
u, X = sp.symbols('u, X')

d = {
    'dim':1,
    'scheme_velocity':1.,
    'schemes':[
        {
            'velocities': [1, 2],
            'conserved_moments':u,
            'polynomials': [1, X],
            'equilibrium': [u, .5*u**2],
            'relaxation_parameters': [0., 1.9],
        },
    ],
}
s = pylbm.Scheme(d)
print(s)
```

The same dictionary has been used for this application with only one modification: the equilibrium value of the second moment m_1^{eq} is taken to $\frac{1}{2}m_0^2$.

`script`

D_1Q_3 for the wave equation

The wave equation is rewritten into the system of two partial differential equations

$$\begin{cases} \partial_t u(t,x) + \partial_x v(t,x) = 0, & t > 0, x \in \mathbb{R}, \\ \partial_t v(t,x) + c^2 \partial_x u(t,x) = 0, & t > 0, x \in \mathbb{R}. \end{cases}$$

The following scheme can be used:

```
import sympy as sp
import pylbm
u, v, X = sp.symbols('u, v, X')

c = 0.5
d = {
  'dim':1,
  'scheme_velocity':1.,
  'schemes':[{
    'velocities': [0, 1, 2],
    'conserved_moments':[u, v],
    'polynomials': [1, X, 0.5*X**2],
    'equilibrium': [u, v, .5*c**2*u],
    'relaxation_parameters': [0., 0., 1.9],
    },
  ],
}
s = pylbm.Scheme(d)
print(s)
```

Examples in 2D

`script`

D_2Q_4 for the advection

A velocity $(c_x, c_y) \in \mathbb{R}^2$ being given, the advection equation reads

$$\partial_t u(t,x,y) + c_x \partial_x u(t,x,y) + c_y \partial_y u(t,x,y) = 0, \qquad t > 0, x, y \in \mathbb{R}.$$

Taken for instance $c_x = 0.1, c_y = 0.2$, the following scheme can be used:

```
import sympy as sp
import pylbm
u, X, Y = sp.symbols('u, X, Y')

d = {
  'dim':2,
  'scheme_velocity':1.,
  'schemes':[{
    'velocities': [1, 2, 3, 4],
    'conserved_moments':u,
    'polynomials': [1, X, Y, X**2-Y**2],
    'equilibrium': [u, .1*u, .2*u, 0.],
    'relaxation_parameters': [0., 1.9, 1.9, 1.4],
    },
  ],
}
s = pylbm.Scheme(d)
print(s)
```

The dictionary `d` is used to set the dimension to 2, the scheme velocity to 1. The used scheme has four velocities: $v_0 = (1,0)$, $v_1 = (0,1)$, $v_2 = (-1,0)$, and $v_3 = (0,-1)$. The polynomials that define the moments are $P_0 = 1$,

$P_1 = X$, $P_2 = Y$, and $P_3 = X^2 - Y^2$ so that the matrix of the moments is

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

with the convention $M_{ij} = P_i(v_j)$. Then, there are four distribution functions $f_j, 0 \leq j \leq 3$ that move at the velocities v_j, and four moments $m_k = \sum_{j=0}^{3} M_{kj} f_j$. The first moment m_0 is conserved during the relaxation phase (as the associated relaxation parameter is set to 0), while the other moments $m_k, 1 \leq k \leq 3$ relaxe to their equilibrium values by the relations

$$\begin{cases} m_1^\star = m_1 - 1.9(m_1 - 0.1 m_0), \\ m_2^\star = m_2 - 1.9(m_2 - 0.2 m_0), \\ m_3^\star = (1 - 1.4) m_3. \end{cases}$$

`script`

D_2Q_9 for Navier-Stokes

The system of the compressible Navier-Stokes equations reads

$$\begin{cases} \partial_t \rho + \nabla\cdot(\rho \boldsymbol{u}) = 0, \\ \partial_t(\rho \boldsymbol{u}) + \nabla\cdot(\rho \boldsymbol{u} \otimes \boldsymbol{u}) + \nabla p = \kappa \nabla(\nabla\cdot\boldsymbol{u}) + \eta \nabla\cdot\big(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T - \nabla\cdot\boldsymbol{u}\mathbb{I}\big), \end{cases}$$

where we removed the dependency of all unknown on the variables (t, x, y). The vector $\boldsymbol{x}$ stands for (x, y) and, if ψ is a scalar function of $\boldsymbol{x}$ and $\boldsymbol{\phi} = (\phi_x, \phi_y)$ is a vectorial function of $\boldsymbol{x}$, the usual partial differential operators read

$$\begin{aligned} \nabla\psi &= (\partial_x\psi, \partial_y\psi), \\ \nabla\cdot\boldsymbol{\phi} &= \partial_x\phi_x + \partial_y\phi_y, \\ \nabla\cdot(\boldsymbol{\phi}\otimes\boldsymbol{\phi}) &= (\nabla\cdot(\phi_x\boldsymbol{\phi}), \nabla\cdot(\phi_y\boldsymbol{\phi})). \end{aligned}$$

The coefficients κ and η are the bulk and the shear viscosities.

The following dictionary can be used to simulate the system of Navier-Stokes equations in the limit of small velocities:

```
from six.moves import range
import sympy as sp
import pylbm
rho, qx, qy, X, Y = sp.symbols('rho, qx, qy, X, Y')

dx = 1./256    # space step
eta = 1.25e-5  # shear viscosity
kappa = 10*eta # bulk viscosity
sb = 1./(.5+kappa*3./dx)
ss = 1./(.5+eta*3./dx)
d = {
    'dim':2,
    'scheme_velocity':1.,
    'schemes':[{
        'velocities':list(range(9)),
        'conserved_moments':[rho, qx, qy],
        'polynomials':[
            1, X, Y,
            3*(X**2+Y**2)-4,
            (9*(X**2+Y**2)**2-21*(X**2+Y**2)+8)/2,
```

(continues on next page)

(continued from previous page)

```
            3*X*(X**2+Y**2)-5*X, 3*Y*(X**2+Y**2)-5*Y,
            X**2-Y**2, X*Y
        ],
        'relaxation_parameters':[0., 0., 0., sb, sb, sb, sb, ss, ss],
        'equilibrium':[
            rho, qx, qy,
            -2*rho + 3*qx**2 + 3*qy**2,
            rho + 3/2*qx**2 + 3/2*qy**2,
            -qx, -qy,
            qx**2 - qy**2, qx*qy
        ],
    },],
}
s = pylbm.Scheme(d)
print(s)
```

The scheme generated by the dictionary is the 9 velocities scheme with orthogonal moments introduced in [QdHL92]

Examples in 3D

`script`

D_3Q_6 for the advection

A velocity $(c_x, c_y, c_z) \in \mathbb{R}^2$ being given, the advection equation reads

$$\partial_t u(t,x,y,z) + c_x \partial_x u(t,x,y,z) + c_y \partial_y u(t,x,y,z) + c_z \partial_z u(t,x,y,z) = 0, \quad t > 0, x, y, z \in \mathbb{R}.$$

Taken for instance $c_x = 0.1, c_y = -0.1, c_z = 0.2$, the following scheme can be used:

```
from six.moves import range
import sympy as sp
import pylbm
u, X, Y, Z = sp.symbols('u, X, Y, Z')

cx, cy, cz = .1, -.1, .2
d = {
    'dim':3,
    'scheme_velocity':1.,
    'schemes':[{
        'velocities': list(range(1,7)),
        'conserved_moments':u,
        'polynomials': [1, X, Y, Z, X**2-Y**2, X**2-Z**2],
        'equilibrium': [u, cx*u, cy*u, cz*u, 0., 0.],
        'relaxation_parameters': [0., 1.5, 1.5, 1.5, 1.5, 1.5],
    },],
}
s = pylbm.Scheme(d)
print(s)
```

The dictionary `d` is used to set the dimension to 3, the scheme velocity to 1. The used scheme has six velocities: $v_0 = (0,0,1)$, $v_1 = (0,0,-1)$, $v_2 = (0,1,0)$, $v_3 = (0,-1,0)$, $v_4 = (1,0,0)$, and $v_5 = (-1,0,0)$. The polynomials that define the moments are $P_0 = 1$, $P_1 = X$, $P_2 = Y$, $P_3 = Z$, $P_4 = X^2 - Y^2$, and $P_5 = X^2 - Z^2$ so that the

matrix of the moments is

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

with the convention $M_{ij} = P_i(v_j)$. Then, there are six distribution functions $f_j, 0 \leq j \leq 5$ that move at the velocities v_j, and six moments $m_k = \sum_{j=0}^{5} M_{kj} f_j$. The first moment m_0 is conserved during the relaxation phase (as the associated relaxation parameter is set to 0), while the other moments $m_k, 1 \leq k \leq 5$ relaxe to their equilibrium values by the relations

$$\begin{cases} m_1^\star = m_1 - 1.5(m_1 - 0.1m_0), \\ m_2^\star = m_2 - 1.5(m_2 + 0.1m_0), \\ m_3^\star = m_3 - 1.5(m_3 - 0.2m_0), \\ m_4^\star = (1 - 1.5)m_4, \\ m_5^\star = (1 - 1.5)m_5. \end{cases}$$

2.3.2 The vectorial schemes

With pylbm, vectorial schemes can be built easily by using a list of elementary schemes. Each elementary scheme is given by a dictionary as in the previous section. The conserved moments of all the elementary schemes can be used in the equilibrium values of the non conserved moments, in order to couple the schemes. For more details on the vectorial schemes, the reader can refer to [G14].

Examples in 1D

`script`

$D_1Q_{2,2}$ for the shallow water equation

A constant $g \in \mathbb{R}$ being given, the shallow water system reads

$$\begin{aligned} &\partial_t h(t,x) + \partial_x q(t,x) = 0, && t > 0, x \in \mathbb{R}, \\ &\partial_t q(t,x) + \partial_x \left(q^2(t,x)/h(t,x) + g h^2(t,x)/2 \right) = 0, && t > 0, x \in \mathbb{R}. \end{aligned}$$

Taken for instance $g = 1$, the following scheme can be used:

```
import sympy as sp
import pylbm

# parameters
h, q, X, LA, g = sp.symbols('h, q, X, LA, g')
la = 2.              # velocity of the scheme
s_h, s_q = 1.7, 1.5  # relaxation parameters

d = {
    'dim': 1,
    'scheme_velocity': la,
    'schemes':[
        {
            'velocities': [1, 2],
```

(continues on next page)

(continued from previous page)

```
            'conserved_moments': h,
            'polynomials': [1, LA*X],
            'relaxation_parameters': [0, s_h],
            'equilibrium': [h, q],
        },
        {
            'velocities': [1, 2],
            'conserved_moments': q,
            'polynomials': [1, LA*X],
            'relaxation_parameters': [0, s_q],
            'equilibrium': [q, q**2/h+.5*g*h**2],
        },
    ],
    'parameters': {LA: la, g: 1.},
}
s = pylbm.Scheme(d)
print(s)
```

Two elementary schemes have been built, these two schemes are identical except for the equilibrium values of the non conserved moment and of the relaxation parameter: The first one is used to simulate the equation on h and the second one to simulate the equation on q. For each scheme, the equilibrium value of the non conserved moment is equal to the flux of the corresponding equation: the equilibrium value of the kth scheme can so depend on all the conserved moments (and not only on those of the kth scheme).

Examples in 2D

`script`

$D_2Q_{4,4,4}$ for the shallow water equation

A constant $g \in \mathbb{R}$ being given, the shallow water system reads

$$
\begin{aligned}
&\partial_t h(t,x,y) + \partial_x q_x(t,x,y) + \partial_y q_y(t,x,y) = 0, && t>0, x, y \in \mathbb{R},\\
&\partial_t q_x(t,x,y) + \partial_x\big(q_x^2(t,x,y)/h(t,x,y) + gh^2(t,x,y)/2\big)\\
&\qquad + \partial_y\big(q_x(t,x,y)q_y(t,x,y)/h(t,x,y)\big) = 0, && t>0, x, y \in \mathbb{R},\\
&\partial_t q_y(t,x,y) + \partial_x\big(q_x(t,x,y)q_y(t,x,y)/h(t,x,y)\big)\\
&\qquad + \partial_y\big(q_y^2(t,x,y)/h(t,x,y) + gh^2(t,x,y)/2\big) = 0, && t>0, x, y \in \mathbb{R}.
\end{aligned}
$$

Taken for instance $g = 1$, the following scheme can be used:

```
import sympy as sp
import pylbm

X, Y, LA, g = sp.symbols('X, Y, LA, g')
h, qx, qy = sp.symbols('h, qx, qy')

# parameters
la = 4 # velocity of the scheme
s_h  = [0., 2.,  2.,  1.5]
s_q  = [0., 1.5, 1.5, 1.2]

vitesse = [1,2,3,4]
```

(continues on next page)

(continued from previous page)

```
polynomes = [1, LA*X, LA*Y, X**2-Y**2]

d = {
    'dim': 2,
    'scheme_velocity': la,
    'schemes':[
        {
            'velocities': vitesse,
            'conserved_moments': h,
            'polynomials': polynomes,
            'relaxation_parameters': s_h,
            'equilibrium': [h, qx, qy, 0.],
        },
        {
            'velocities': vitesse,
            'conserved_moments': qx,
            'polynomials': polynomes,
            'relaxation_parameters': s_q,
            'equilibrium': [qx, qx**2/h + 0.5*g*h**2, qx*qy/h, 0.],
        },
        {
            'velocities': vitesse,
            'conserved_moments': qy,
            'polynomials': polynomes,
            'relaxation_parameters': s_q,
            'equilibrium': [qy, qy*qx/h, qy**2/h + 0.5*g*h**2, 0.],
        },
    ],
    'parameters': {LA: la, g: 1.},
}

s = pylbm.Scheme(d)
print(s)
```

Three elementary schemes have been built, these three schemes are identical except for the equilibrium values of the non conserved moment and of the relaxation parameter: The first one is used to simulate the equation on h and the others to simulate the equation on q_x and q_y. For each scheme, the equilibrium value of the non conserved moment is equal to the flux of the corresponding equation: the equilibrium value of the kth scheme can so depend on all the conserved moments (and not only on those of the kth scheme).

2.4 Analyze your scheme

Two of the biggest problems encountered when starting to use lattice Bolzmann methods are

- what are the physical equations we're trying to solve?
- how to set the parameters of the scheme (equilibrium values, relaxation parameters,...) so that it is stable and solves what you want?

pylbm tries to give you some ideas to solve them.

For the first one, pylbm can give you the first and second order coefficients of your physical equation (in a next release, it will be possible to have also the third and the fourth order terms). To have better results, it is important to

keep SymPy symbols in your scheme as long as possible. Thus, we can see the influence of theses parameters on the physical equations.

For the second one, once you know that you have the good physical equation, pylbm allows to check the linear stability region for these parameters arround a given linearized state. We provide widgets inside a notebook or for a Python script to modify easily these parameters and show the result on the figure interactively.

We believe that these two tools will make it easier for as many people as possible to become familiar with the lattice Boltzmann methods and, in the end, allow them to implement their own schemes.

Let's take a simple example to illustrate how it works.

Assume that you want to solve the advection equation for 1D problem

$$\begin{aligned} &\partial_t u = c\partial_x u, && t>0, \quad x\in(0,1),\\ &u(t=0,x) = u_0(x), && x\in(0,1)\\ &u(t,x=0) = u(t,x=1), && t>0. \end{aligned}$$

And you already have a lattice Boltzmann scheme that you want to try: the $\mathrm{D}_1\mathrm{Q}_3$ scheme given by

- three velocities $v_0=0$, $v_1=1$, and $v_2=-1$, with associated distribution functions f_0, f_1, and f_2,
- the scheme velocity λ,
- the three moments

$$m_0 = \sum_{i=0}^{2} f_i, \quad m_1 = \sum_{i=0}^{2} v_i\, f_i, \quad m_2 = \frac{1}{2}\sum_{i=0}^{2} v_i^2\, f_i,$$

  and their equilibrium values m_0^e, m_1^e, and m_2^e,
- and finaly the two relaxation parameters s_1 and s_2 lying in $[0,2]$.

We can write this scheme into pylbm as

[1]:
```
import sympy as sp

# Symbolic variables definitions
U, X = sp.symbols('u, X')
C, LA, S0, S1 = sp.symbols('c, lambda, s_0, s_1', constants=True)

# The D1Q3 LBM scheme
adv_scheme = {
    'dim': 1,
    'scheme_velocity': LA,
    'schemes': [
        {
            'velocities': list(range(3)),
            'conserved_moments': U,
            'polynomials': [1, X, X**2/2],
            'relaxation_parameters': [0., S0, S1],
            'equilibrium': [U, C*U, C**2*U/2],
        },
    ],
    'parameters': {LA: 1,
                   S0: 1.9,
                   S1: 1.9,
                   C: 1},
}
```

Let's create the scheme and look at the information given by pylbm

[2]:
```
import pylbm

scheme = pylbm.Scheme(adv_scheme)
```

[3]:
```
print(scheme)
```

```
+-------------+
| Scheme information |
+-------------+
    - spatial dimension: 1
    - number of schemes: 1
    - number of velocities: 3
    - conserved moments: [u]

    +-------+
    | Scheme 0 |
    +-------+
        - velocities
            (0: 0)
            (1: 1)
            (2: -1)

        - polynomials

            1

            X

             2
            X
            ──
            2

        - equilibria

             u

            cu

             2
            c u
            ────
             2

        - relaxation parameters

            0.0

            s

            s

    - moments matrices

        1  1   1

        0     -
```

(continues on next page)

(continued from previous page)

```
           2   2

       0  ──  ──
          2   2

   - inverse of moments matrices

               -2
       1   0   ──
                 2


           1   1
       0   ──  ──
           2   2


          -1   1
       0  ──  ──
          2   2
```

We can see here that we have described one scheme with three 1D velocities. The moment matrix gives how to find the moments from the distribution functions.

Let's check now if we solve the good physical equations.

[4]:
```
pde = pylbm.EquivalentEquation(scheme)
```

[5]:
```
print(pde)
```

```
+---------------+
| Equivalent Equations |
+---------------+
    The equation is


    d         d             d
    ──(Fx) + ──(U) = ──Bxx──(U)
    dx        dt       x    dx

    where


    U = [u]

    Fx = [cu]

    Bxx = [0]
```

pylbm gives the first and second order terms. In the next release, you will also have access to the third and fourth terms. Our scheme solves the advection equation as exepected.

We can now study the stability of this scheme. Many notions of stability exist and can be used. In pylbm, we focus on a linear notion by computing the eigenvalues of the linear operator corresponding to one time step. The scheme will

be considered as stable if all these eigenvalues stay inside the unit circle (as complex values). This notion is sufficient for linear scheme but just gives partial informations for non-linear scheme.

[6]:
```
stab = pylbm.Stability(scheme)

# linearization around a state
uo = 1.

stab.visualize(
    {
        'linearization': {
            U: uo,
        },
        'parameters': {
            LA: {
                'range': [1, 20],
                'init': 1,
                'step': .1
            },
            U: {
                'range': [0, 20],
                'init': uo,
                'step': .1
            },
            C: {
                'range': [0, 20],
                'init': 1,
                'step': .1
            },
            S0: {
                'range': [0, 2],
                'init': 1.9,
                'step': .1
            },
            S1: {
                'range': [0, 2],
                'init': 1.9,
                'step': .1
            },
        },
        'number_of_wave_vectors': 1024,
    }
)
```

```
Output()
```

```
FloatSlider(value=1.0, continuous_update=False, description='', layout=Layout(width=
↪'80%'), max=20.0, min=1.0...
```

```
FloatSlider(value=1.0, continuous_update=False, description='u', layout=Layout(width=
↪'80%'), max=20.0)
```

```
FloatSlider(value=1.0, continuous_update=False, description='c', layout=Layout(width=
↪'80%'), max=20.0)
```

```
FloatSlider(value=1.9, continuous_update=False, description='s₀', layout=Layout(width=
↪'80%'), max=2.0)
```

```
FloatSlider(value=1.9, continuous_update=False, description='s₁', layout=Layout(width=
↪'80%'), max=2.0)
```

2.5 The storage

When you use pylbm, a generated code is performed using the descritpion of the scheme(s) (the velocities, the polynomials, the conserved moments, the equilibriums, ...). There are several generators already implemented

- NumPy
- Cython
- Pythran (work in progress)
- Loo.py (work in progress)

To have best performance following the generator, you need a specific storage of the moments and distribution functions arrays. For example, it is preferable to have a storage like $[n_v, n_x, n_y, n_z]$ in NumPy n_v is the number of velocities and n_x, n_y and n_z the grid size. It is due to the vectorized form of the algorithm. Whereas for Cython, it is preferable to have the storage $[n_x, n_y, n_z, n_v]$ using the pull algorithm.

So, we have implemented a storage class that always gives to the user the same access to the moments and disribution functions arrays but with a different storage in memory for the generator. This class is called *Array*.

It is really simple to create an array. You just need to give

- the number of velocities,
- the global grid size,
- the size of the fictitious point in each direction,
- the order of $[n_v, n_x, n_y, n_z]$ with the following indices
  - 0: n_v
  - 1: n_x
  - 2: n_y
  - 3: n_z

  The default order is $[n_v, n_x, n_y, n_z]$.
- the mpi topology (optional)
- the type of the data (optional)

  The default is double

2.5.1 2D example

Suppose that you want to create an array with a grid size $[5, 10]$ and 9 velocities with 1 cell in each direction for the fictitious domain.

[25]:
```
from pylbm.storage import Array
import numpy as np
a = Array(9, [5, 10], [1, 1])
```

[28]:
```
for i in range(a.nv):
    a[i] = i
```

[29]:
```
print(a[:])
```

```
[[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]

 [[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
  [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
  [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
  [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
  [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]]

 [[ 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.]
  [ 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.]
  [ 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.]
  [ 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.]
  [ 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.]]

 [[ 3.  3.  3.  3.  3.  3.  3.  3.  3.  3.]
  [ 3.  3.  3.  3.  3.  3.  3.  3.  3.  3.]
  [ 3.  3.  3.  3.  3.  3.  3.  3.  3.  3.]
  [ 3.  3.  3.  3.  3.  3.  3.  3.  3.  3.]
  [ 3.  3.  3.  3.  3.  3.  3.  3.  3.  3.]]

 [[ 4.  4.  4.  4.  4.  4.  4.  4.  4.  4.]
  [ 4.  4.  4.  4.  4.  4.  4.  4.  4.  4.]
  [ 4.  4.  4.  4.  4.  4.  4.  4.  4.  4.]
  [ 4.  4.  4.  4.  4.  4.  4.  4.  4.  4.]
  [ 4.  4.  4.  4.  4.  4.  4.  4.  4.  4.]]

 [[ 5.  5.  5.  5.  5.  5.  5.  5.  5.  5.]
  [ 5.  5.  5.  5.  5.  5.  5.  5.  5.  5.]
  [ 5.  5.  5.  5.  5.  5.  5.  5.  5.  5.]
  [ 5.  5.  5.  5.  5.  5.  5.  5.  5.  5.]
  [ 5.  5.  5.  5.  5.  5.  5.  5.  5.  5.]]

 [[ 6.  6.  6.  6.  6.  6.  6.  6.  6.  6.]
  [ 6.  6.  6.  6.  6.  6.  6.  6.  6.  6.]
  [ 6.  6.  6.  6.  6.  6.  6.  6.  6.  6.]
  [ 6.  6.  6.  6.  6.  6.  6.  6.  6.  6.]
  [ 6.  6.  6.  6.  6.  6.  6.  6.  6.  6.]]

 [[ 7.  7.  7.  7.  7.  7.  7.  7.  7.  7.]
  [ 7.  7.  7.  7.  7.  7.  7.  7.  7.  7.]
  [ 7.  7.  7.  7.  7.  7.  7.  7.  7.  7.]
  [ 7.  7.  7.  7.  7.  7.  7.  7.  7.  7.]
  [ 7.  7.  7.  7.  7.  7.  7.  7.  7.  7.]]

 [[ 8.  8.  8.  8.  8.  8.  8.  8.  8.  8.]
  [ 8.  8.  8.  8.  8.  8.  8.  8.  8.  8.]
  [ 8.  8.  8.  8.  8.  8.  8.  8.  8.  8.]
  [ 8.  8.  8.  8.  8.  8.  8.  8.  8.  8.]
  [ 8.  8.  8.  8.  8.  8.  8.  8.  8.  8.]]]
```

[30]:
```
b = Array(9, [5, 10], [1, 1], sorder=[2, 1, 0])
for i in range(b.nv):
    b[i] = i
```

[31]:
```
print(b[:])
```

```
[[[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]

 [[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
  [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
  [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
  [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
  [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]]

 [[ 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.]
  [ 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.]
  [ 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.]
  [ 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.]
  [ 2.  2.  2.  2.  2.  2.  2.  2.  2.  2.]]

 [[ 3.  3.  3.  3.  3.  3.  3.  3.  3.  3.]
  [ 3.  3.  3.  3.  3.  3.  3.  3.  3.  3.]
  [ 3.  3.  3.  3.  3.  3.  3.  3.  3.  3.]
  [ 3.  3.  3.  3.  3.  3.  3.  3.  3.  3.]
  [ 3.  3.  3.  3.  3.  3.  3.  3.  3.  3.]]

 [[ 4.  4.  4.  4.  4.  4.  4.  4.  4.  4.]
  [ 4.  4.  4.  4.  4.  4.  4.  4.  4.  4.]
  [ 4.  4.  4.  4.  4.  4.  4.  4.  4.  4.]
  [ 4.  4.  4.  4.  4.  4.  4.  4.  4.  4.]
  [ 4.  4.  4.  4.  4.  4.  4.  4.  4.  4.]]

 [[ 5.  5.  5.  5.  5.  5.  5.  5.  5.  5.]
  [ 5.  5.  5.  5.  5.  5.  5.  5.  5.  5.]
  [ 5.  5.  5.  5.  5.  5.  5.  5.  5.  5.]
  [ 5.  5.  5.  5.  5.  5.  5.  5.  5.  5.]
  [ 5.  5.  5.  5.  5.  5.  5.  5.  5.  5.]]

 [[ 6.  6.  6.  6.  6.  6.  6.  6.  6.  6.]
  [ 6.  6.  6.  6.  6.  6.  6.  6.  6.  6.]
  [ 6.  6.  6.  6.  6.  6.  6.  6.  6.  6.]
  [ 6.  6.  6.  6.  6.  6.  6.  6.  6.  6.]
  [ 6.  6.  6.  6.  6.  6.  6.  6.  6.  6.]]

 [[ 7.  7.  7.  7.  7.  7.  7.  7.  7.  7.]
  [ 7.  7.  7.  7.  7.  7.  7.  7.  7.  7.]
  [ 7.  7.  7.  7.  7.  7.  7.  7.  7.  7.]
  [ 7.  7.  7.  7.  7.  7.  7.  7.  7.  7.]
  [ 7.  7.  7.  7.  7.  7.  7.  7.  7.  7.]]

 [[ 8.  8.  8.  8.  8.  8.  8.  8.  8.  8.]
  [ 8.  8.  8.  8.  8.  8.  8.  8.  8.  8.]
  [ 8.  8.  8.  8.  8.  8.  8.  8.  8.  8.]
  [ 8.  8.  8.  8.  8.  8.  8.  8.  8.  8.]
  [ 8.  8.  8.  8.  8.  8.  8.  8.  8.  8.]]]
```

You can see that the access of the data is the same for a et b whereas the sorder is not the same.

If we look at the *array* attribute which is the real storage of our data

[32]:
```
a.array.shape
```
[32]:
```
(9, 5, 10)
```

[33]:
```
b.array.shape
```
[33]:
```
(10, 5, 9)
```

you can see that it is not the same and it is exactly what we want. To do that, we use the swapaxes of numpy and we use this representation to have an access to our data.

2.5.2 Access to the data with the conserved moments

When you discribe your scheme, you define the conserved moments. It is usefull to have a direct acces to these moments by giving their name and not their indices in the array. So, it is possible to specify where are the conserved moments in the array.

Let define conserved moments using sympy symbol.

[35]:
```
import sympy
rho, u, v = sympy.symbols("rho, u, v")
```

We indicate to pylbm where are located these conserved moments in our array by giving a list of two elements: the first one is the scheme number and the second one the index in this scheme.

[45]:
```
a.set_conserved_moments({rho: [0, 0], u: [0, 2], v: [0, 1]})
```

[46]:
```
a[rho]
```
[46]:
```
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

[47]:
```
a[u]
```
[47]:
```
array([[ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.,  2.]])
```

[48]:
```
a[v]
```
[48]:
```
array([[ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.]])
```

[ ]:
```
```

2.6 Tutorial

2.6.1 Transport in 1D

In this tutorial, we test the most simple lattice Boltzmann scheme $\mathrm{D}_1\mathrm{Q}_2$ on two classical hyperbolic scalar equations: the advection equation and the Burger's equation.

The advection equation

The problem reads

$$\partial_t u + c\partial_x u = 0, \quad t > 0, \quad x \in (0,1),$$

where c is a constant scalar (typically $c = 1$). Additional boundary and initial conditions will be given in the following.

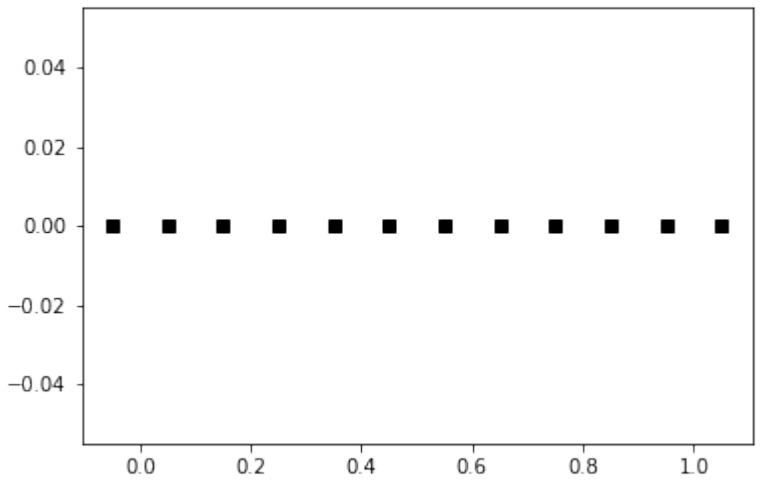
The numerical simulation of this equation by a lattice Boltzmann scheme consists in the approximation of the solution on discret points of $(0,1)$ at discret instants.

The spatial mesh is defined by using a numpy array. To simplify, the mesh is supposed to be uniform.

First, we import the package numpy and we create the spatial mesh. One phantom cell has to be added at each edge of the domain for the treatment of the boundary conditions.

[1]:
```
%matplotlib inline
```

[2]:
```
import numpy as np
import pylab as plt

def mesh(N):
    xmin, xmax = 0., 1.
    dx = 1./N
    x = np.linspace(xmin-.5*dx, xmax+.5*dx, N+2)
    return x

x = mesh(10)
plt.plot(x, 0.*x, 'sk')
plt.show()
```



To simulate this equation, we use the $\mathrm{D}_1\mathrm{Q}_2$ scheme given by

- two velocities $v_0 = -1$, $v_1 = 1$, with associated distribution functions f_0 and f_1,
- a space step Δx and a time step Δt, the ration $\lambda = \Delta x/\Delta t$ is called the scheme velocity,
- two moments $m_0 = \sum_{i=0}^{1} f_i$ and $m_1 = \lambda \sum_{i=0}^{1} v_i f_i$ and their equilibrium values $m_0^e = m_0$, $m_1^e = c\, m_0$,
- a relaxation parameter s lying in $[0, 2]$.

In order to prepare the formalism of the package pylbm, we introduce the two polynomials that define the moments: $P_0 = 1$ and $P_1 = \lambda X$, such that

$$m_k = \sum_{i=0}^{1} P_k(v_i)\, f_i.$$

The transformation $(f_0, f_1) \mapsto (m_0, m_1)$ is invertible if, and only if, the polynomials (P_0, P_1) is a free set over the stencil of velocities.

The lattice Boltzmann method consists to compute the distribution functions f_0 and f_1 in each point of the lattice x and at each time $t^n = n\Delta t$. A step of the scheme can be read as a splitting between the relaxation phase and the transport phase:

- relaxation:

$$m_1^\star(t, x) = (1 - s)\, m_1(t, x) + s\, m_1^e(t, x).$$

- m2f:

$$\begin{aligned} f_0^\star(t, x) &= (m_0(t, x) - m_1^\star(t, x)/\lambda)/2, \\ f_1^\star(t, x) &= (m_0(t, x) + m_1^\star(t, x)/\lambda)/2. \end{aligned}$$

- transport:

$$f_0(t + \Delta t, x) = f_0^\star(t, x + \Delta x), \qquad f_1(t + \Delta t, x) = f_1^\star(t, x - \Delta x).$$

- f2m:

$$\begin{aligned} m_0(t + \Delta t, x) &= f_0(t + \Delta t, x) + f_1(t + \Delta t, x), \\ m_1(t + \Delta t, x) &= -\lambda\, f_0(t + \Delta t, x) + \lambda\, f_1(t + \Delta t, x). \end{aligned}$$

The moment of order 0, m_0, being the only one conserved during the relaxation phase, the equivalent equation of this scheme reads at first order

$$\partial_t\, m_0 + \partial_x\, m_1^e = \mathcal{O}(\Delta t).$$

We implement a function equilibrium that computes the equilibrium value m_1^e, the moment of order 0, m_0, and the velocity c being given in argument.

[3]:
```
def equilibrium(m0, c):
    return c*m0
```

Then, we create two vectors m_0 and m_1 with shape the shape of the mesh and initialize them. The moment of order 0 should contain the initial value of the unknown u and the moment of order 1 the corresponding equilibrium value.

We create also two vectors f_0 and f_1.

[4]:
```
def initialize(mesh, c, la):
    m0 = np.zeros(mesh.shape)
    m0[np.logical_and(mesh<0.5, mesh>0.25)] = 1.
    m1 = equilibrium(m0, c)
    f0, f1 = np.empty(m0.shape), np.empty(m0.shape)
    m2f(f0, f1, m0, m1, la)
    return f0, f1, m0, m1
```

And finally, we implement the four elementary functions f2m, relaxation, m2f, and transport. In the transport function, the boundary conditions should be implemented: we will use periodic conditions by copying the informations in the phantom cells.

[5]:
```
def f2m(f0, f1, m0, m1, la):
    m0[:] = f0 + f1
    m1[:] = la*(f1 - f0)

def m2f(f0, f1, m0, m1, la):
    f0[:] = 0.5*(m0-m1/la)
    f1[:] = 0.5*(m0+m1/la)

def relaxation(m0, m1, c, s):
    m1[:] = (1-s)*m1 + s*equilibrium(m0, c)

def transport(f0, f1):
    #periodical boundary conditions
    f0[-1] = f0[1]
    f1[0] = f1[-2]
    #transport
    f0[1:-1] = f0[2:]
    f1[1:-1] = f1[:-2]
```

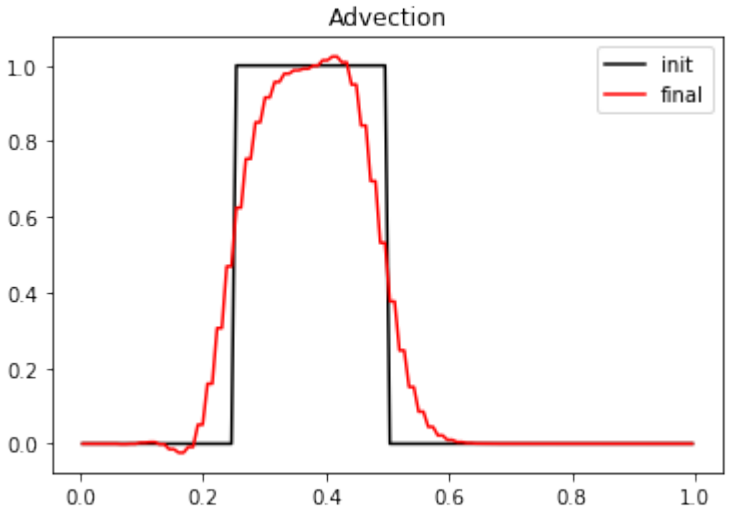
We compute and we plot the numerical solution at time $T_f = 2$.

[6]:
```
# parameters
c = .5  # velocity for the transport equation
Tf = 2. # final time
N = 128 # number of points in space
la = 1. # scheme velocity
s = 1.8 # relaxation parameter
# initialization
x = mesh(N)
f0, f1, m0, m1 = initialize(x, c, la)
t = 0
dt = (x[1]-x[0])/la
plt.figure(1)
plt.clf()
plt.plot(x[1:-1], m0[1:-1], 'k', label='init')
while t<Tf:
    t += dt
    relaxation(m0, m1, c, s)
    m2f(f0, f1, m0, m1, la)
    transport(f0, f1)
    f2m(f0, f1, m0, m1, la)
plt.plot(x[1:-1], m0[1:-1], 'r', label='final')
plt.legend()
plt.title('Advection')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



The Burger's equation

The problem reads

$$\partial_t u + \tfrac{1}{2}\partial_x u^2 = 0, \quad t > 0, \quad x \in (0,1).$$

The previous D_1Q_2 scheme can simulate the Burger's equation by modifying the equilibrium value of the moment of order 1 m_1^e. It now reads $m_1^e = {m_0}^2/2$.

More generaly, the simulated equation is into the conservative form

$$\partial_t u + \partial_x \varphi(u) = 0, \quad t > 0, \quad x \in (0,1),$$

the equilibrium has to be taken to $m_1^e = \varphi(m_0)$.

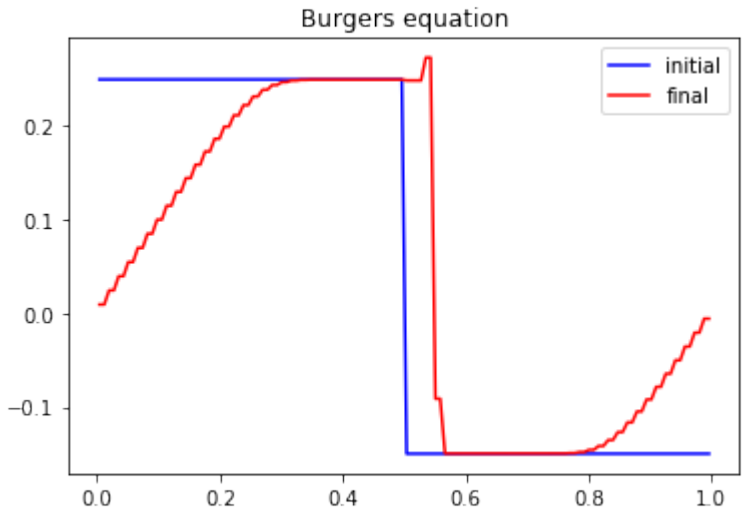
We just have to modify the equilibrium and the initialization of the previous example to simulate the Burger's equation. The initial condition can be a discontinuous function in order to simulate Riemann problems. Note that the function f2m, m2f, relaxation, and transport are unchanged.

[7]:
```
def equilibrium(m0):
    return .5*m0**2

def initialize(mesh, la):
    ug, ud = 0.25, -0.15
    xmin, xmax = .5*np.sum(mesh[:2]), .5*np.sum(mesh[-2:])
    xc = xmin + .5*(xmax-xmin)
    m0 = ug*(mesh<xc) + ud*(mesh>xc) + .5*(ug+ud)*(mesh==xc)
    m1 = equilibrium(m0)
    f0 = np.empty(m0.shape)
    f1 = np.empty(m0.shape)
    return f0, f1, m0, m1

def relaxation(m0, m1, s):
    m1[:] = (1-s)*m1 + s*equilibrium(m0)
```

(continues on next page)

(continued from previous page)

```
# parameters
Tf = 1. # final time
N = 128 # number of points in space
la = 1. # scheme velocity
s = 1.8 # relaxation parameter
# initialization
x = mesh(N)      # mesh
dx = x[1]-x[0]   # space step
dt = dx/la       # time step
f0, f1, m0, m1 = initialize(x, la)
plt.figure(1)
plt.plot(x[1:-1], m0[1:-1], 'b', label='initial')
# time loops
t = 0.
while (t<Tf):
    t += dt
    relaxation(m0, m1, s)
    m2f(f0, f1, m0, m1, la)
    transport(f0, f1)
    f2m(f0, f1, m0, m1, la)
plt.plot(x[1:-1], m0[1:-1], 'r', label='final')
plt.title('Burgers equation')
plt.legend(loc='best')
plt.show()
```

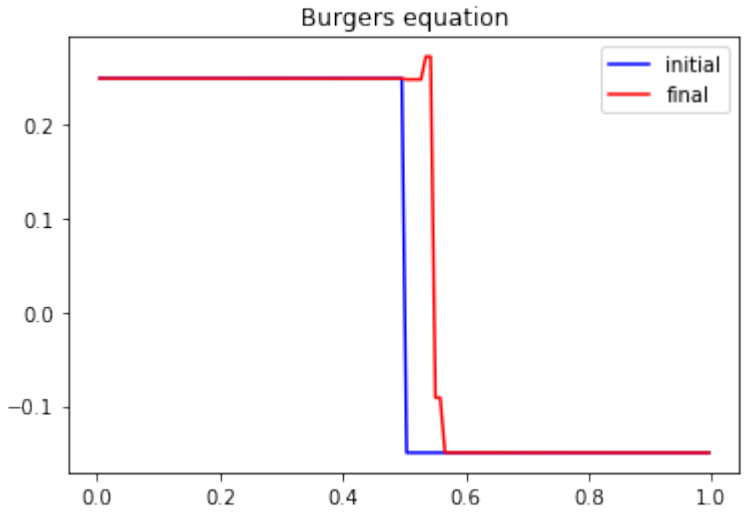


We can test different values of the relaxation parameter s. In particular, we observe that the scheme remains stable if $s \in [0, 2]$. More s is small, more the numerical diffusion is important and if s is close to 2, oscillations appear behind the shock.

In order to simulate a Riemann problem, the boundary conditions have to be modified. A classical way is to impose entry conditions for hyperbolic problems. The lattice Boltzmann methods lend themselves very well to that conditions: the scheme only needs the distributions corresponding to a velocity that goes inside the domain. Nevertheless, on a physical edge where the flux is going outside, a non physical distribution that goes inside has to be imposed. A first simple way is to leave the initial value: this is correct while the discontinuity does not reach the edge. A second way is to impose Neumann condition by repeating the inner value.

We modify the previous script to take into account these new boundary conditions.

[8]:
```
def transport(f0, f1):
    # Neumann boundary conditions
    f0[-1] = f0[-2]
    f1[0] = f1[1]
    # transport
    f0[1:-1] = f0[2:]
    f1[1:-1] = f1[:-2]

# parameters
Tf = 1. # final time
N = 128 # number of points in space
la = 1. # scheme velocity
s = 1.8 # relaxation parameter

# initialization
x = mesh(N)      # mesh
dx = x[1]-x[0]   # space step
dt = dx/la       # time step
f0, f1, m0, m1 = initialize(x, la)
plt.figure(1)
plt.plot(x[1:-1], m0[1:-1], 'b', label='initial')
# time loops
t = 0.
while (t<Tf):
    t += dt
    relaxation(m0, m1, s)
    m2f(f0, f1, m0, m1, la)
    transport(f0, f1)
    f2m(f0, f1, m0, m1, la)
plt.plot(x[1:-1], m0[1:-1], 'r', label='final')
plt.title('Burgers equation')
plt.legend(loc='best')
plt.show()
```



2.6.2 The wave equation in 1D

In this tutorial, we test a very classical lattice Boltzmann scheme $\mathrm{D}_1\mathrm{Q}_3$ on the wave equation.

The problem reads

$$\partial_{tt}\rho = c^2\partial_{xx}\rho, \qquad t > 0, \quad x \in (0, 2\pi),$$

where c is a constant scalar. In this session, two different kinds of boundary conditions will be considered:

- periodic conditions $\rho(0) = \rho(2\pi)$,
- Homogeneous Dirichlet conditions $\rho(0) = \rho(2\pi) = 0$.

The problem is transformed into a one order system:

$$\begin{aligned} \partial_t\rho + \partial_x q = 0, &\qquad t > 0, \quad x \in (0, 2\pi), \\ \partial_t q + c^2\partial_x\rho = 0, &\qquad t > 0, \quad x \in (0, 2\pi). \end{aligned}$$

The scheme $\mathrm{D}_1\mathrm{Q}_3$

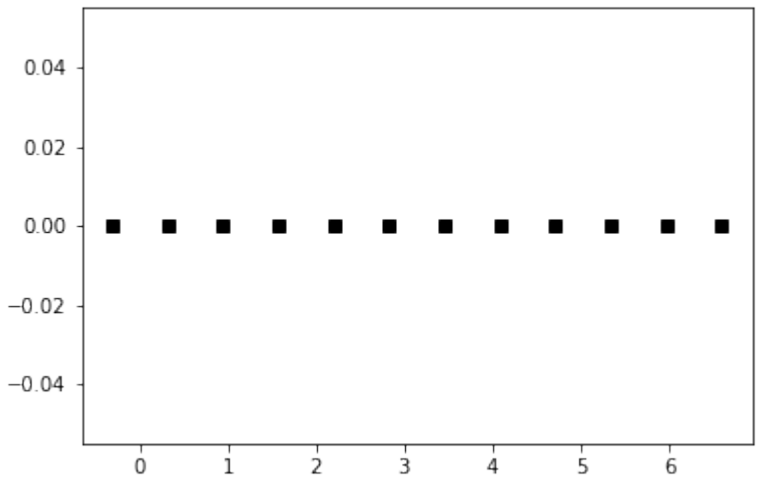
The numerical simulation of this equation by a lattice Boltzmann scheme consists in the approximation of the solution on discret points of $(0, 2\pi)$ at discret instants.

The spatial mesh is defined by using a numpy array. To simplify, the mesh is supposed to be uniform.

First, we import the package numpy and we create the spatial mesh. One phantom cell has to be added at each bound for the treatment of the boundary conditions.

[1]:
```
%matplotlib inline
```

[2]:
```
import numpy as np
import pylab as plt

def mesh(N):
    xmin, xmax = 0., 2.*np.pi
    dx = (xmax-xmin)/N
    x = np.linspace(xmin-.5*dx, xmax+.5*dx, N+2)
    return x

x = mesh(10)
plt.plot(x, 0.*x, 'sk')
plt.show()
```



To simulate this system of equations, we use the $\mathrm{D_1Q_3}$ scheme given by

- three velocities $v_0 = 0$, $v_1 = 1$, and $v_2 = -1$, with associated distribution functions f_0, f_1, and f_2,
- a space step Δx and a time step Δt, the ration $\lambda = \Delta x/\Delta t$ is called the scheme velocity,
- three moments

$$m_0 = \sum_{i=0}^{2} f_i, \quad m_1 = \lambda \sum_{i=0}^{2} v_i\, f_i, \quad m_2 = \frac{\lambda^2}{2} \sum_{i=0}^{2} v_i^2\, f_i,$$

and their equilibrium values $m_0^e = m_0$, $m_1^e = m_1$, and $m_2^e = c^2/2\, m_0$.

- a relaxation parameter s lying in $[0, 2]$.

In order to prepare the formalism of the package pylbm, we introduce the three polynomials that define the moments: $P_0 = 1$, $P_1 = \lambda X$, and $P_2 = \lambda^2/2X^2$, such that

$$m_k = \sum_{i=0}^{2} P_k(v_i)\, f_i.$$

The transformation $(f_0, f_1, f_2) \mapsto (m_0, m_1, m_2)$ is invertible if, and only if, the polynomials (P_0, P_1, P_2) is a free set over the stencil of velocities.

The lattice Boltzmann method consists to compute the distribution functions f_0, f_1, and f_2 in each point of the lattice x and at each time $t^n = n\Delta t$. A step of the scheme can be read as a splitting between the relaxation phase and the transport phase:

- relaxation:

$$m_2^\star(t, x) = (1 - s)\, m_2(t, x) + s\, m_2^e(t, x).$$

- m2f:

$$\begin{aligned} f_0^\star(t, x) &= m_0(t, x) - 2\, m_2^\star(t, x)/\lambda^2, \\ f_1^\star(t, x) &= m_1(t, x)/(2\lambda) + m_2^\star(t, x)/\lambda^2, \\ f_2^\star(t, x) &= -\, m_1(t, x)/(2\lambda) + m_2^\star(t, x)/\lambda^2. \end{aligned}$$

- transport:

$$\begin{aligned} f_0(t + \Delta t, x) &= f_0^\star(t, x), \\ f_1(t + \Delta t, x) &= f_1^\star(t, x - \Delta x), \\ f_2(t + \Delta t, x) &= f_2^\star(t, x + \Delta x). \end{aligned}$$

- f2m:

$$\begin{aligned} m_0(t + \Delta t, x) &= f_0(t + \Delta t, x) + f_1(t + \Delta t, x) + f_2(t + \Delta t, x), \\ m_1(t + \Delta t, x) &= \lambda\, f_1(t + \Delta t, x) - \lambda\, f_2(t + \Delta t, x), \\ m_2(t + \Delta t, x) &= \tfrac{1}{2}\lambda^2\, f_1(t + \Delta t, x) + \tfrac{1}{2}\lambda^2\, f_2(t + \Delta t, x). \end{aligned}$$

The moments of order 0, m_0, and of order 1, m_1, being conserved during the relaxation phase, the equivalent equations of this scheme read at first order

$$\begin{aligned} &\partial_t\, m_0 + \partial_x\, m_1 = \mathcal{O}(\Delta t), \\ &\partial_t\, m_1 + 2\partial_x\, m_2^e = \mathcal{O}(\Delta t). \end{aligned}$$

We implement a function equilibrium that computes the equilibrium value m_2^e, the moment of order 0, m_0, and the velocity c being given in argument.

[3]:
```
def equilibrium(m0, c):
    return .5*c**2*m0
```

We create three vectors m_0, m_1, and m_2 with shape the shape of the mesh and initialize them. The moments of order 0 and 1 should contain the initial value of the unknowns ρ and q, and the moment of order 2 the corresponding equilibrium value.

We create also three vectors f_0, f_1 and f_2.

[4]:
```
def initialize(mesh, c, la):
    m0 = np.sin(mesh)
    m1 = np.zeros(mesh.shape)
    m2 = equilibrium(m0, c)
    f0 = np.empty(m0.shape)
    f1 = np.empty(m0.shape)
    f2 = np.empty(m0.shape)
    return f0, f1, f2, m0, m1, m2
```

Periodic boundary conditions

We implement the four elementary functions f2m, relaxation, m2f, and transport. In the transport function, the boundary conditions should be implemented: we will use periodic conditions by copying the informations in the phantom cells.

[5]:
```
def f2m(f0, f1, f2, m0, m1, m2, la):
    m0[:] = f0 + f1 + f2
    m1[:] = la * (f2 - f1)
    m2[:] = .5* la**2 * (f1 + f2)

def m2f(f0, f1, f2, m0, m1, m2, la):
    f0[:] = m0 - 2./la**2 * m2
    f1[:] = -.5/la * m1 + 1/la**2 * m2
    f2[:] = .5/la * m1 + 1/la**2 * m2

def relaxation(m0, m1, m2, c, s):
    m2[:] *= (1-s)
    m2[:] += s*equilibrium(m0, c)

def transport(f0, f1, f2):
    # periodic boundary conditions
    f1[-1] = f1[1]
    f2[0] = f2[-2]
    # transport
    f1[1:-1] = f1[2:]
    f2[1:-1] = f2[:-2]
```

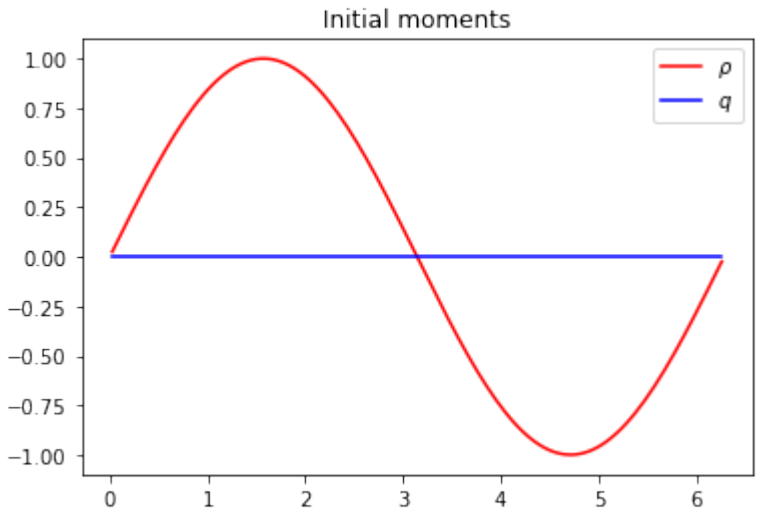
We compute and we plot the numerical solution at time $T_f = 2\pi$.

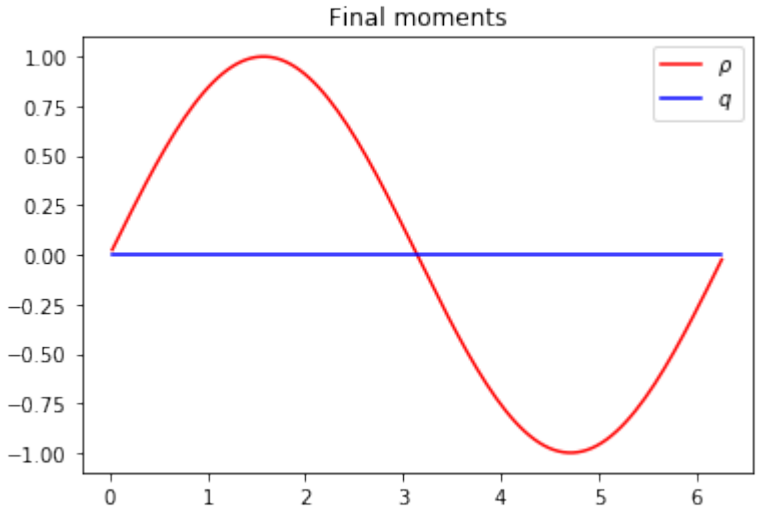
[6]:
```
# parameters
c = 1   # velocity for the transport equation
Tf = 2.*np.pi # final time
N = 128 # number of points in space
la = 1. # scheme velocity
s = 2.  # relaxation parameter
# initialization
x = mesh(N)      # mesh
```

(continues on next page)

(continued from previous page)

```
dx = x[1]-x[0]  # space step
dt = dx/la      # time step
f0, f1, f2, m0, m1, m2 = initialize(x, c, la)
plt.figure(1)
plt.plot(x[1:-1], m0[1:-1], 'r', label=r'$\rho$')
plt.plot(x[1:-1], m1[1:-1], 'b', label=r'$q$')
plt.title('Initial moments')
plt.legend(loc='best')
# time loops
nt = int(Tf/dt)
m2f(f0, f1, f2, m0, m1, m2, la)
for k in range(nt):
    transport(f0, f1, f2)
    f2m(f0, f1, f2, m0, m1, m2, la)
    relaxation(m0, m1, m2, c, s)
    m2f(f0, f1, f2, m0, m1, m2, la)
plt.figure(2)
plt.plot(x[1:-1], m0[1:-1], 'r', label=r'$\rho$')
plt.plot(x[1:-1], m1[1:-1], 'b', label=r'$q$')
plt.title('Final moments')
plt.legend(loc='best')
plt.show()
```





Anti bounce back conditions

In order to take into account homogenous Dirichlet conditions over ρ, we introduce the bounce back conditions. At edge $x = 0$, two points are involved: $x_0 = -\Delta x/2$ and $x_1 = \Delta x/2$. We impose $f_1(x_0) = -f_2(x_1)$. And at edge $x = 2\pi$, the two involved points are x_N and x_{N+1}. We impose $f_2(x_{N+1}) = -f_1(x_N)$.

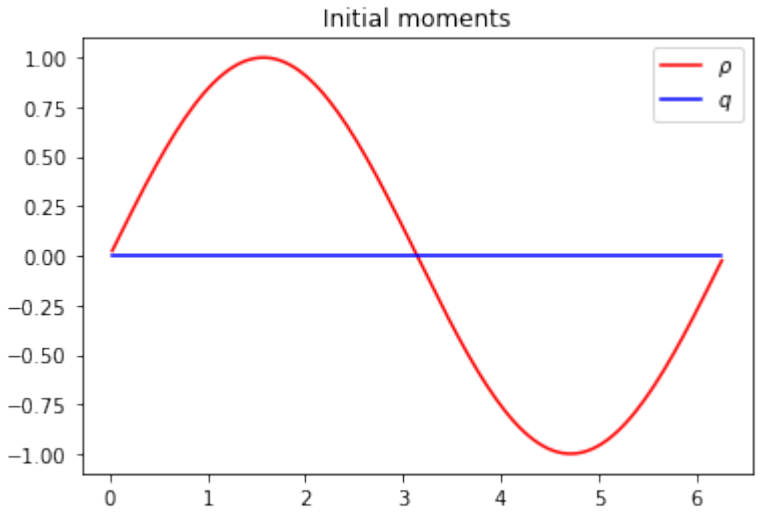
We modify the transport function to impose anti bounce back conditions. We can compare the solutions obtained with the two different boundary conditions.

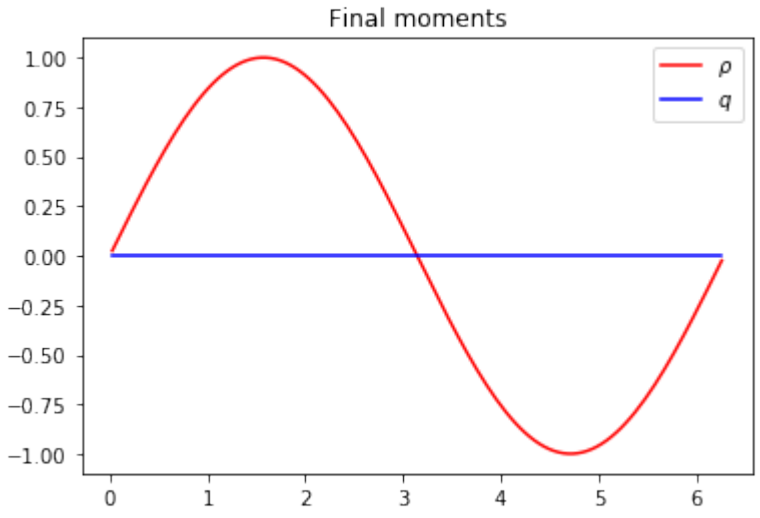
[7]:
```
def transport(f0, f1, f2):
    # anti bounce back boundary conditions
    f1[-1] = -f2[-2]
    f2[0] = -f1[1]
    # transport
    f1[1:-1] = f1[2:]
    f2[1:-1] = f2[:-2]


# parameters
c = 1   # velocity for the transport equation
Tf = 2*np.pi # final time
N = 128 # number of points in space
la = 1. # scheme velocity
s = 2.  # relaxation parameter
# initialization
x = mesh(N)      # mesh
dx = x[1]-x[0]   # space step
dt = dx/la       # time step
f0, f1, f2, m0, m1, m2 = initialize(x, c, la)
plt.figure(1)
plt.plot(x[1:-1], m0[1:-1], 'r', label=r'$\rho$')
plt.plot(x[1:-1], m1[1:-1], 'b', label=r'$q$')
plt.title('Initial moments')
plt.legend(loc='best')
# time loops
nt = int(Tf/dt)
m2f(f0, f1, f2, m0, m1, m2, la)
```

(continues on next page)

(continued from previous page)

```
for k in range(nt):
    transport(f0, f1, f2)
    f2m(f0, f1, f2, m0, m1, m2, la)
    relaxation(m0, m1, m2, c, s)
    m2f(f0, f1, f2, m0, m1, m2, la)
plt.figure(2)
plt.plot(x[1:-1], m0[1:-1], 'r', label=r'$\rho$')
plt.plot(x[1:-1], m1[1:-1], 'b', label=r'$q$')
plt.title('Final moments')
plt.legend(loc='best')
plt.show()
```





[ ]:

2.6.3 The heat equation in 1D

In this tutorial, we test a very classical lattice Boltzmann scheme $\mathrm{D_1Q_3}$ on the heat equation.

The problem reads

$$\partial_t u = \mu \partial_{xx} u, \quad t > 0, \quad x \in (0,1),$$
$$u(0) = u(1) = 0,$$

where μ is a constant scalar.

[1]:
```
%matplotlib inline
```

The scheme $\mathrm{D_1Q_3}$

The numerical simulation of this equation by a lattice Boltzmann scheme consists in the approximatation of the solution on discret points of $(0,1)$ at discret instants.

To simulate this system of equations, we use the $\mathrm{D_1Q_3}$ scheme given by

- three velocities $v_0 = 0$, $v_1 = 1$, and $v_2 = -1$, with associated distribution functions f_0, f_1, and f_2,
- a space step Δx and a time step Δt, the ration $\lambda = \Delta x/\Delta t$ is called the scheme velocity,
- three moments

  $$m_0 = \sum_{i=0}^{2} f_i, \quad m_1 = \sum_{i=0}^{2} v_i\, f_i, \quad m_2 = \frac{1}{2}\sum_{i=0}^{2} v_i^2\, f_i,$$

  and their equilibrium values m_0^e, m_1^e, and m_2^e.
- two relaxation parameters s_1 and s_2 lying in $[0,2]$.

In order to use the formalism of the package pylbm, we introduce the three polynomials that define the moments: $P_0 = 1$, $P_1 = X$, and $P_2 = X^2/2$, such that

$$m_k = \sum_{i=0}^{2} P_k(v_i)\, f_i.$$

The transformation $(f_0, f_1, f_2) \mapsto (m_0, m_1, m_2)$ is invertible if, and only if, the polynomials (P_0, P_1, P_2) is a free set over the stencil of velocities.

The lattice Boltzmann method consists to compute the distribution functions f_0, f_1, and f_2 in each point of the lattice x and at each time $t^n = n\Delta t$. A step of the scheme can be read as a splitting between the relaxation phase and the transport phase:

- relaxation:

  $$m_1^\star(t,x) = (1-s_1)\, m_1(t,x) + s_1\, m_1^e(t,x),$$
  $$m_2^\star(t,x) = (1-s_2)\, m_2(t,x) + s_2\, m_2^e(t,x).$$

- m2f:

  $$\begin{aligned} f_0^\star(t,x) &= m_0(t,x) - 2\, m_2^\star(t,x), \\ f_1^\star(t,x) &= m_1^\star(t,x)/2 + m_2^\star(t,x), \\ f_2^\star(t,x) &= -m_1^\star(t,x)/2 + m_2^\star(t,x). \end{aligned}$$

- transport:

  $$\begin{aligned} f_0(t+\Delta t,x) &= f_0^\star(t,x), \\ f_1(t+\Delta t,x) &= f_1^\star(t,x-\Delta x), \\ f_2(t+\Delta t,x) &= f_2^\star(t,x+\Delta x). \end{aligned}$$

- f2m:

$$\begin{aligned} m_0(t+\Delta t,x) &= f_0(t+\Delta t,x) + f_1(t+\Delta t,x) + f_2(t+\Delta t,x),\\ m_1(t+\Delta t,x) &= f_1(t+\Delta t,x) - f_2(t+\Delta t,x),\\ m_2(t+\Delta t,x) &= \tfrac{1}{2} f_1(t+\Delta t,x) + \tfrac{1}{2} f_2(t+\Delta t,x). \end{aligned}$$

The moment of order 0, m_0, being conserved during the relaxation phase, a diffusive scaling $\Delta t = \Delta x^2$, yields to the following equivalent equation

$$\partial_t m_0 = 2\big(\tfrac{1}{s_1} - \tfrac{1}{2}\big)\partial_{xx} m_2^e + \mathcal{O}(\Delta x^2),$$

if $m_1^e = 0$. In order to be consistent with the heat equation, the following choice is done:

$$m_2^e = \tfrac{1}{2}u, \qquad s_1 = \frac{2}{1+2\mu}, \qquad s_2 = 1.$$

Using pylbm

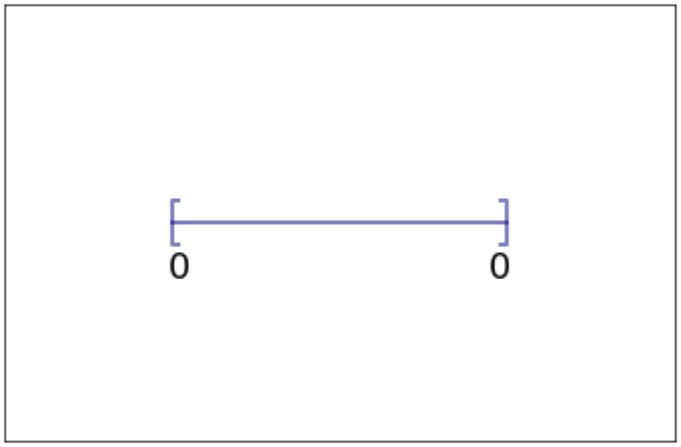
pylbm uses Python dictionary to describe the simulation. In the following, we will build this dictionary step by step.

The geometry

In pylbm, the geometry is defined by a box and a label for the boundaries.

[2]:
```
import pylbm
import numpy as np

xmin, xmax = 0., 1.
dico_geom = {
    'box': {'x': [xmin, xmax], 'label':0},
}
geom = pylbm.Geometry(dico_geom)
print(geom)
geom.visualize(viewlabel=True);
```

```
+---------------+
| Geometry information |
+---------------+
    - spatial dimension: 1
    - bounds of the box: [0. 1.]
    - labels: [0, 0]
```

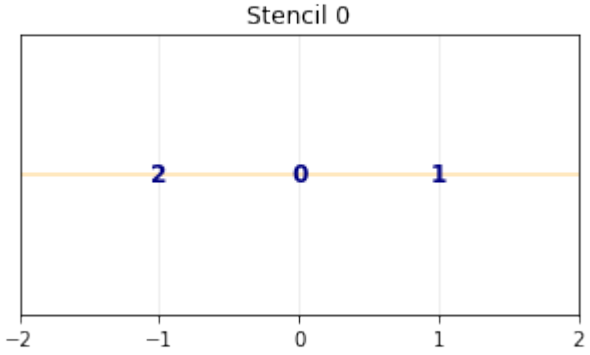


The stencil

pylbm provides a class stencil that is used to define the discret velocities of the scheme. In this example, the stencil is composed by the velocities $v_0 = 0$, $v_1 = 1$ and $v_2 = -1$ numbered by $[0, 1, 2]$.

[3]:
```
dico_sten = {
    'dim': 1,
    'schemes':[{'velocities': list(range(3))}],
}
sten = pylbm.Stencil(dico_sten)
print(sten)
sten.visualize();
```

```
+--------------+
| Stencil information |
+--------------+
    - spatial dimension: 1
    - minimal velocity in each direction: [-1]
    - maximal velocity in each direction: [1]
    - information for each elementary stencil:
        stencil 0
            - number of velocities: 3
            - velocities
                (0: 0)
                (1: 1)
                (2: -1)
```



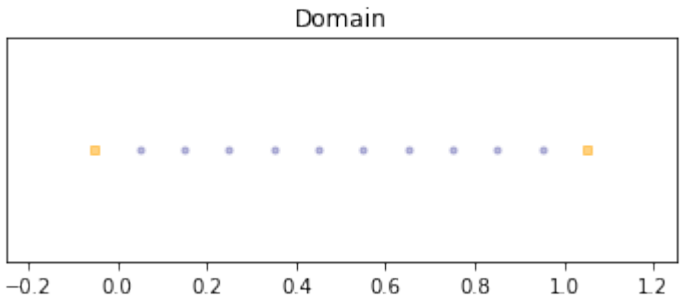
The domain

In order to build the domain of the simulation, the dictionary should contain the space step Δx and the stencils of the velocities (one for each scheme).

We construct a domain with $N = 10$ points in space.

[4]:
```
N = 10
dx = (xmax-xmin)/N
dico_dom = {
    'box': {'x': [xmin, xmax], 'label':0},
    'space_step': dx,
    'schemes': [
        {
            'velocities': list(range(3)),
        }
    ],
}
dom = pylbm.Domain(dico_dom)
print(dom)
dom.visualize();
```

```
+-------------+
| Domain information |
+-------------+
    - spatial dimension: 1
    - space step: 0.1
    - with halo:
        bounds of the box: [-0.05] x [1.05]
        number of points: [12]
    - without halo:
        bounds of the box: [0.05] x [0.95]
        number of points: [10]

    +---------------+
    | Geometry information |
    +---------------+
        - spatial dimension: 1
        - bounds of the box: [0. 1.]
        - labels: [0, 0]
```



The scheme

In pylbm, a simulation can be performed by using several coupled schemes. In this example, a single scheme is used and defined through a list of one single dictionary. This dictionary should contain:

- 'velocities': a list of the velocities
- 'conserved_moments': a list of the conserved moments as sympy variables
- 'polynomials': a list of the polynomials that define the moments
- 'equilibrium': a list of the equilibrium value of all the moments
- 'relaxation_parameters': a list of the relaxation parameters (0 for the conserved moments)
- 'init': a dictionary to initialize the conserved moments

(see the documentation for more details)

The scheme velocity could be taken to $1/\Delta x$ and the inital value of u to

$$u(t=0,x) = \sin(\pi x).$$

[5]:
```
import sympy as sp

def solution(x, t):
    return np.sin(np.pi*x)*np.exp(-np.pi**2*mu*t)

# parameters
mu = 1.
la = 1./dx
s1 = 2./(1+2*mu)
s2 = 1.
u, X = sp.symbols('u, X')

dico_sch = {
    'dim': 1,
    'scheme_velocity': la,
    'schemes':[
        {
            'velocities': list(range(3)),
            'conserved_moments': u,
            'polynomials': [1, X, X**2/2],
            'equilibrium': [u, 0., .5*u],
            'relaxation_parameters': [0., s1, s2],
        }
    ],
}

sch = pylbm.Scheme(dico_sch)
print(sch)
```

```
+-------------+
| Scheme information |
+-------------+
    - spatial dimension: 1
    - number of schemes: 1
    - number of velocities: 3
    - conserved moments: [u]

    +-------+
    | Scheme 0 |
    +-------+
        - velocities
```

(continues on next page)

(continued from previous page)

```
            (0: 0)
            (1: 1)
            (2: -1)

        - polynomials

            1

            X

             2
            X
            ──
            2

        - equilibria

              u

             0.0

            0.5u

        - relaxation parameters

                    0.0

            0.666666666666667

                    1.0

    - moments matrices

        1  1    1

        0  10  -10

        0  50  50

    - inverse of moments matrices

        1    0    -1/50

        0  1/20   1/100

        0  -1/20  1/100
```

The simulation

A simulation is built by defining a correct dictionary.

We combine the previous dictionaries to build a simulation. In order to impose the homogeneous Dirichlet conditions in $x = 0$ and $x = 1$, the dictionary should contain the key 'boundary_conditions' (we use pylbm.bc.Anti_bounce_back function).

[6]:
```
dico = {
    'box': {'x':[xmin, xmax], 'label':0},
    'space_step': dx,
    'scheme_velocity': la,
    'schemes':[
        {
            'velocities': list(range(3)),
            'conserved_moments': u,
            'polynomials': [1, X, X**2/2],
            'equilibrium': [u, 0., .5*u],
            'relaxation_parameters': [0., s1, s2],
        }
    ],
    'init': {u:(solution,(0.,))},
    'boundary_conditions': {
        0: {'method': {0: pylbm.bc.AntiBounceBack,}},
    },
    'generator': 'numpy'
}

sol = pylbm.Simulation(dico)
print(sol)
```

```
+----------------+
| Simulation information |
+----------------+

    +-------------+
    | Domain information |
    +-------------+
        - spatial dimension: 1
        - space step: 0.1
        - with halo:
            bounds of the box: [-0.05] x [1.05]
            number of points: [12]
        - without halo:
            bounds of the box: [0.05] x [0.95]
            number of points: [10]

        +---------------+
        | Geometry information |
        +---------------+
            - spatial dimension: 1
            - bounds of the box: [0. 1.]
            - labels: [0, 0]

    +-------------+
    | Scheme information |
    +-------------+
        - spatial dimension: 1
        - number of schemes: 1
        - number of velocities: 3
        - conserved moments: [u]

        +-------+
        | Scheme 0 |
        +-------+
            - velocities
```

(continues on next page)

(continued from previous page)

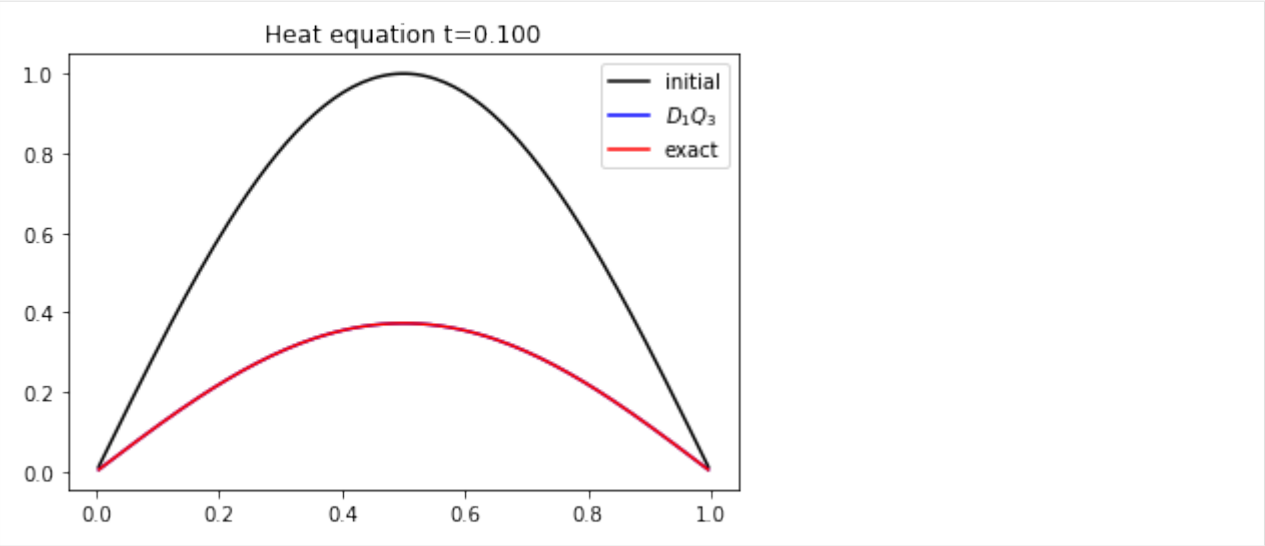
```
            (0: 0)
            (1: 1)
            (2: -1)

        - polynomials

            1

            X

             2
            X
            ──
            2

        - equilibria

              u

             0.0

            0.5u

        - relaxation parameters

                    0.0

            0.666666666666667

                    1.0

    - moments matrices

        1  1    1

        0  10  -10

        0  50  50

    - inverse of moments matrices

        1    0    -1/50

        0  1/20   1/100

        0  -1/20  1/100
```

Run a simulation

Once the simulation is initialized, one time step can be performed by using the function one_time_step.

We compute the solution of the heat equation at $t = 0.1$. And, on the same graphic, we plot the initial condition, the exact solution and the numerical solution.

[7]:
```
import numpy as np
import sympy as sp
import pylab as plt
import pylbm

u, X, LA = sp.symbols('u, X, LA')

def solution(x, t):
    return np.sin(np.pi*x)*np.exp(-np.pi**2*mu*t)

xmin, xmax = 0., 1.
N = 128
mu = 1.
Tf = .1
dx = (xmax-xmin)/N # spatial step
la = 1./dx
s1 = 2./(1+2*mu)
s2 = 1.
dico = {
    'box':{'x': [xmin, xmax], 'label': 0},
    'space_step': dx,
    'scheme_velocity': la,
    'schemes': [
        {
            'velocities': list(range(3)),
            'conserved_moments': u,
            'polynomials': [1, X/LA, X**2/(2*LA**2)],
            'equilibrium': [u, 0., .5*u],
            'relaxation_parameters': [0., s1, s2],
        }
    ],
    'init': {u: (solution, (0.,))},
    'boundary_conditions': {
        0: {'method': {0: pylbm.bc.AntiBounceBack,}},
    },
    'parameters': {LA: la},
    'generator': 'numpy'
}

sol = pylbm.Simulation(dico)
x = sol.domain.x
y = sol.m[u]

plt.figure(1)
plt.plot(x, y, 'k', label='initial')

while sol.t < 0.1:
    sol.one_time_step()

plt.plot(x, sol.m[u], 'b', label=r'$D_1Q_3$')
plt.plot(x, solution(x, sol.t),'r', label='exact')
plt.title('Heat equation t={0:5.3f}'.format(sol.t))
plt.legend();
```



[1]:
```
%matplotlib inline
```

2.6.4 The heat equation in 2D

In this tutorial, we test a very classical lattice Boltzmann scheme $\mathrm{D_2Q_5}$ on the heat equation.

The problem reads

$$\partial_t u = \mu(\partial_{xx} + \partial_{yy})u, \quad t > 0, \quad (x, y) \in (0, 1)^2,$$
$$u(0) = u(1) = 0,$$

where μ is a constant scalar.

The scheme $\mathrm{D_2Q_5}$

The numerical simulation of this equation by a lattice Boltzmann scheme consists in the approximatation of the solution on discret points of $(0, 1)^2$ at discret instants.

To simulate this system of equations, we use the $\mathrm{D_2Q_5}$ scheme given by

- five velocities $v_0 = (0, 0)$, $v_1 = (1, 0)$, $v_2 = (0, 1)$, $v_3 = (-1, 0)$, and $v_4 = (0, -1)$ with associated distribution functions f_i, $0 \leq i \leq 4$,
- a space step Δx and a time step Δt, the ration $\lambda = \Delta x/\Delta t$ is called the scheme velocity,
- five moments

  $$m_0 = \sum_{i=0}^{4} f_i, \quad m_1 = \sum_{i=0}^{4} v_{ix}\, f_i, \quad m_2 = \sum_{i=0}^{4} v_{iy}\, f_i, \quad m_3 = \frac{1}{2}\sum_{i=0}^{5}(v_{ix}^2 + v_{iy}^2)\, f_i, \quad m_4 = \frac{1}{2}\sum_{i=0}^{5}(v_{ix}^2 - v_{iy}^2)\, f_i,$$

  and their equilibrium values m_k^e, $0 \leq k \leq 4$.
- two relaxation parameters s_1 and s_2 lying in $[0, 2]$ (s_1 for the odd moments and s_2 for the odd ones).

In order to use the formalism of the package pylbm, we introduce the five polynomials that define the moments: $P_0 = 1$, $P_1 = X$, $P_2 = Y$, $P_3 = (X^2 + Y^2)/2$, and $P_4 = (X^2 - Y^2)/2$, such that

$$m_k = \sum_{i=0}^{4} P_k(v_{ix}, v_{iy})\, f_i.$$

The transformation $(f_0, f_1, f_2, f_3, f_4) \mapsto (m_0, m_1, m_2, m_3, m_4)$ is invertible if, and only if, the polynomials $(P_0, P_1, P_2, P_3, P_4)$ is a free set over the stencil of velocities.

The lattice Boltzmann method consists to compute the distribution functions f_i, $0 \leq i \leq 4$ in each point of the lattice x and at each time $t^n = n\Delta t$. A step of the scheme can be read as a splitting between the relaxation phase and the transport phase:

- relaxation:

$$\begin{aligned}
m_1^\star(t,x,y) &= (1-s_1)\, m_1(t,x,y) + s_1\, m_1^e(t,x,y),\\
m_2^\star(t,x,y) &= (1-s_1)\, m_2(t,x,y) + s_1\, m_2^e(t,x,y),\\
m_3^\star(t,x,y) &= (1-s_2)\, m_3(t,x,y) + s_2\, m_3^e(t,x,y),\\
m_4^\star(t,x,y) &= (1-s_2)\, m_4(t,x,y) + s_2\, m_4^e(t,x,y).
\end{aligned}$$

- m2f:

$$\begin{aligned}
f_0^\star(t,x,y) &= m_0(t,x,y) - 2\, m_3^\star(t,x,y),\\
f_1^\star(t,x,y) &= \tfrac{1}{2}(\ \ m_1^\star(t,x,y) + m_3^\star(t,x,y) + m_4^\star(t,x,y)),\\
f_2^\star(t,x,y) &= \tfrac{1}{2}(\ \ m_2^\star(t,x,y) + m_3^\star(t,x,y) - m_4^\star(t,x,y)),\\
f_3^\star(t,x,y) &= \tfrac{1}{2}(-m_1^\star(t,x,y) + m_3^\star(t,x,y) + m_4^\star(t,x,y)),\\
f_4^\star(t,x,y) &= \tfrac{1}{2}(-m_2^\star(t,x,y) + m_3^\star(t,x,y) - m_4^\star(t,x,y)).
\end{aligned}$$

- transport:

$$\begin{aligned}
f_0(t+\Delta t,x,y) &= f_0^\star(t,x,y),\\
f_1(t+\Delta t,x,y) &= f_1^\star(t,x-\Delta x,y),\\
f_2(t+\Delta t,x,y) &= f_2^\star(t,x,y-\Delta x),\\
f_3(t+\Delta t,x,y) &= f_3^\star(t,x+\Delta x,y),\\
f_4(t+\Delta t,x,y) &= f_4^\star(t,x,y+\Delta x).
\end{aligned}$$

- f2m:

$$\begin{aligned}
m_0(t+\Delta t,x,y) &= f_0(t+\Delta t,x,y) + f_1(t+\Delta t,x,y) + f_2(t+\Delta t,x,y)\\
&\quad + f_3(t+\Delta t,x,y) + f_4(t+\Delta t,x,y),\\
m_1(t+\Delta t,x,y) &= f_1(t+\Delta t,x,y) - f_3(t+\Delta t,x,y),\\
m_2(t+\Delta t,x,y) &= f_2(t+\Delta t,x,y) - f_4(t+\Delta t,x,y),\\
m_3(t+\Delta t,x,y) &= \tfrac{1}{2}(f_1(t+\Delta t,x,y) + f_2(t+\Delta t,x,y) + f_3(t+\Delta t,x,y) + f_4(t+\Delta t,x,y)),\\
m_4(t+\Delta t,x,y) &= \tfrac{1}{2}(f_1(t+\Delta t,x,y) - f_2(t+\Delta t,x,y) + f_3(t+\Delta t,x,y) - f_4(t+\Delta t,x,y)).
\end{aligned}$$

The moment of order 0, m_0, being conserved during the relaxation phase, a diffusive scaling $\Delta t = \Delta x^2$, yields to the following equivalent equation

$$\partial_t\, m_0 = \big(\tfrac{1}{s_1} - \tfrac{1}{2}\big)\big(\partial_{xx}(m_3^e + m_4^e) + \partial_{yy}(m_3^e - m_4^e)\big) + \mathcal{O}(\Delta x^2),$$

if $m_1^e = 0$. In order to be consistent with the heat equation, the following choice is done:

$$m_3^e = \tfrac{1}{2}u, \qquad m_4^e = 0, \qquad s_1 = \frac{2}{1+4\mu}, \qquad s_2 = 1.$$

Using pylbm

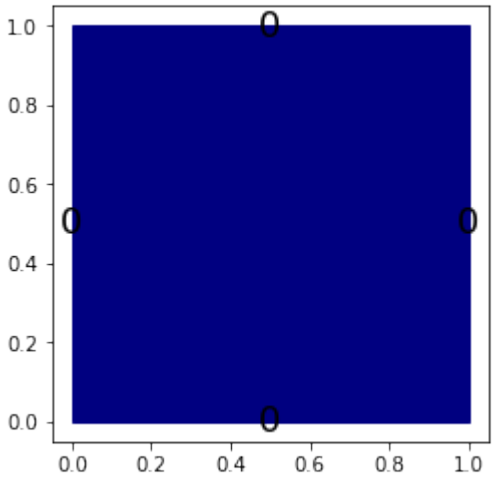
pylbm uses Python dictionary to describe the simulation. In the following, we will build this dictionary step by step.

The geometry

In pylbm, the geometry is defined by a box and a label for the boundaries. We define here a square $(0,1)^2$.

[2]:
```
import pylbm
import numpy as np
import pylab as plt
xmin, xmax, ymin, ymax = 0., 1., 0., 1.
dico_geom = {
    'box': {'x': [xmin, xmax],
            'y': [ymin, ymax],
            'label': 0
           },
}
geom = pylbm.Geometry(dico_geom)
print(geom)
geom.visualize(viewlabel=True);
```

```
+---------------+
| Geometry information |
+---------------+
    - spatial dimension: 2
    - bounds of the box: [0. 1.] x [0. 1.]
    - labels: [0, 0, 0, 0]
```



The stencil

pylbm provides a class stencil that is used to define the discret velocities of the scheme. In this example, the stencil is composed by the velocities $v_0 = (0,0)$, $v_1 = (1,0)$, $v_2 = (-1,0)$, $v_3 = (0,1)$, and $v_4 = (0,-1)$ numbered by $[0,1,2,3,4]$.

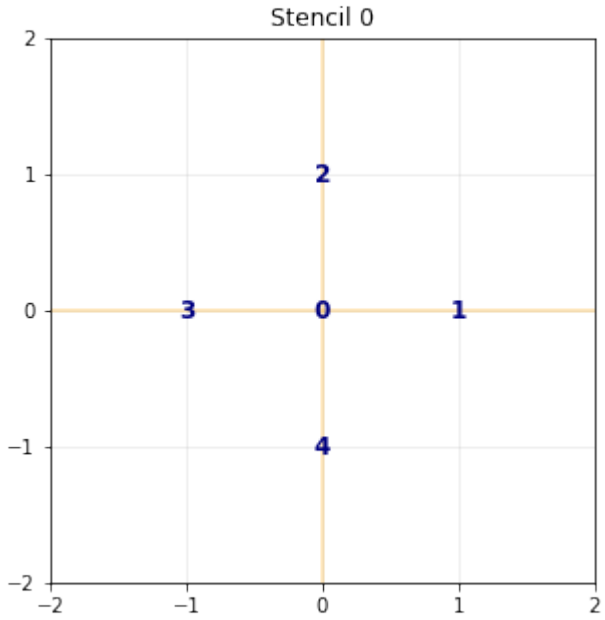
[3]:
```
dico_sten = {
    'dim':2,
```

(continues on next page)

(continued from previous page)

```
    'schemes': [
        {'velocities': list(range(5))}
    ],
}
sten = pylbm.Stencil(dico_sten)
print(sten)
sten.visualize();
```

```
+--------------+
| Stencil information |
+--------------+
    - spatial dimension: 2
    - minimal velocity in each direction: [-1 -1]
    - maximal velocity in each direction: [1 1]
    - information for each elementary stencil:
        stencil 0
            - number of velocities: 5
            - velocities
                (0: 0, 0)
                (1: 1, 0)
                (2: 0, 1)
                (3: -1, 0)
                (4: 0, -1)
```



The domain

In order to build the domain of the simulation, the dictionary should contain the space step Δx and the stencils of the velocities (one for each scheme).

We construct a domain with $N = 10$ points in space.

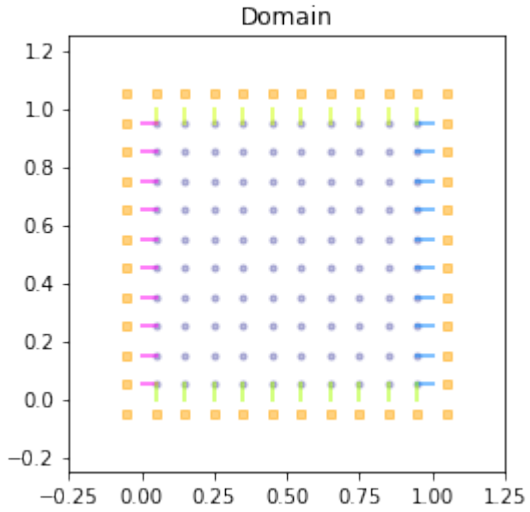
[4]:
```
N = 10
dx = (xmax-xmin)/N
dico_dom = {
    'box': {'x': [xmin, xmax],
```

(continues on next page)

(continued from previous page)

```
            'y': [ymin, ymax],
            'label': 0
           },
    'space_step': dx,
    'schemes': [
        {'velocities': list(range(5)),}
    ],
}
dom = pylbm.Domain(dico_dom)
print(dom)
dom.visualize(view_distance=True);
```

```
+-------------+
| Domain information |
+-------------+
    - spatial dimension: 2
    - space step: 0.1
    - with halo:
        bounds of the box: [-0.05 -0.05] x [1.05 1.05]
        number of points: [12, 12]
    - without halo:
        bounds of the box: [0.05 0.05] x [0.95 0.95]
        number of points: [10, 10]

    +---------------+
    | Geometry information |
    +---------------+
        - spatial dimension: 2
        - bounds of the box: [0. 1.] x [0. 1.]
        - labels: [0, 0, 0, 0]
```



The scheme

In pylbm, a simulation can be performed by using several coupled schemes. In this example, a single scheme is used and defined through a list of one single dictionary. This dictionary should contain:

- 'velocities': a list of the velocities

- 'conserved_moments': a list of the conserved moments as sympy variables
- 'polynomials': a list of the polynomials that define the moments
- 'equilibrium': a list of the equilibrium value of all the moments
- 'relaxation_parameters': a list of the relaxation parameters (0 for the conserved moments)
- 'init': a dictionary to initialize the conserved moments

(see the documentation for more details)

The scheme velocity could be taken to $1/\Delta x$ and the inital value of u to

$$u(t=0,x) = \sin(\pi x)\sin(\pi y).$$

[5]:
```
import sympy as sp

def solution(x, y, t):
    return np.sin(np.pi*x)*np.sin(np.pi*y)*np.exp(-2*np.pi**2*mu*t)

# parameters
mu = 1.
la = 1./dx
s1 = 2./(1+4*mu)
s2 = 1.
u, X, Y, LA = sp.symbols('u, X, Y, LA')

dico_sch = {
    'dim': 2,
    'scheme_velocity': la,
    'schemes': [
        {
            'velocities': list(range(5)),
            'conserved_moments': u,
            'polynomials': [1, X/LA, Y/LA, (X**2+Y**2)/(2*LA**2), (X**2-Y**2)/
↪(2*LA**2)],
            'equilibrium': [u, 0., 0., .5*u, 0.],
            'relaxation_parameters': [0., s1, s1, s2, s2],
        }
    ],
    'parameters': {LA: la},
}

sch = pylbm.Scheme(dico_sch)
print(sch)
```

```
+-------------+
| Scheme information |
+-------------+
    - spatial dimension: 2
    - number of schemes: 1
    - number of velocities: 5
    - conserved moments: [u]

    +-------+
    | Scheme 0 |
    +-------+
        - velocities
```

(continues on next page)

(continued from previous page)

```
            (0: 0, 0)
            (1: 1, 0)
            (2: 0, 1)
            (3: -1, 0)
            (4: 0, -1)

        - polynomials

               1

              X
              ──
              LA

              Y
              ──
              LA

             2    2
            X  + Y
            ──────
                2
             2LA

             2    2
            X  - Y
            ──────
                2
             2LA

        - equilibria

               u

             0.0

             0.0

            0.5u

             0.0

        - relaxation parameters

            0.0

            0.4

            0.4

            1.0

            1.0

    - moments matrices

        1   1    1     1     1
```

(continues on next page)

(continued from previous page)

```
          10          -10
    0   ──     0    ─────   0
        LA           LA

              10          -10
    0   0     ──     0    ─────
              LA           LA

         50    50    50    50
    0   ──── ──── ──── ────
          2     2     2     2
        LA    LA    LA    LA

         50   -50    50   -50
    0   ──── ───── ──── ─────
          2     2     2     2
        LA    LA    LA    LA
```

- inverse of moments matrices

```
                        2
                     -LA
    1   0     0     ──────    0
                       50

                        2       2
         LA           LA      LA
    0    ──     0     ──      ──
         20           200     200

                        2       2
               LA     LA     -LA
    0   0      ──     ──     ────
               20     200     200

                        2       2
        -LA            LA      LA
    0   ────   0       ──      ──
         20            200     200

                        2       2
              -LA      LA     -LA
    0   0     ────     ──     ────
               20     200     200
```

The simulation

A simulation is built by defining a correct dictionary.

We combine the previous dictionaries to build a simulation. In order to impose the homogeneous Dirichlet conditions in $x = 0$, $x = 1$, $y = 0$, and $y = 1$, the dictionary should contain the key 'boundary_conditions' (we use pylbm.bc.Anti_bounce_back function).

[6]:
```
dico = {
    'box': {'x': [xmin, xmax],
            'y': [ymin, ymax],
            'label': 0},
    'space_step': dx,
    'scheme_velocity': la,
    'schemes': [
        {
            'velocities': list(range(5)),
            'conserved_moments': u,
            'polynomials': [1, X/LA, Y/LA, (X**2+Y**2)/(2*LA**2), (X**2-Y**2)/
↪(2*LA**2)],
            'equilibrium': [u, 0., 0., .5*u, 0.],
            'relaxation_parameters': [0., s1, s1, s2, s2],
        }
    ],
    'init': {u: (solution, (0.,))},
    'boundary_conditions': {
        0: {'method': {0: pylbm.bc.AntiBounceBack,}},
    },
    'parameters': {LA: la},
}

sol = pylbm.Simulation(dico)
print(sol)
```

```
+----------------+
| Simulation information |
+----------------+

    +-------------+
    | Domain information |
    +-------------+
        - spatial dimension: 2
        - space step: 0.1
        - with halo:
            bounds of the box: [-0.05 -0.05] x [1.05 1.05]
            number of points: [12, 12]
        - without halo:
            bounds of the box: [0.05 0.05] x [0.95 0.95]
            number of points: [10, 10]

        +---------------+
        | Geometry information |
        +---------------+
            - spatial dimension: 2
            - bounds of the box: [0. 1.] x [0. 1.]
            - labels: [0, 0, 0, 0]

    +-------------+
    | Scheme information |
    +-------------+
        - spatial dimension: 2
        - number of schemes: 1
        - number of velocities: 5
        - conserved moments: [u]

        +-------+
```

(continues on next page)

(continued from previous page)

```
        | Scheme 0 |
        +-------+
            - velocities
                (0: 0, 0)
                (1: 1, 0)
                (2: 0, 1)
                (3: -1, 0)
                (4: 0, -1)

            - polynomials

                   1

                  X
                  ──
                  LA

                  Y
                  ──
                  LA

                 2    2
                X  + Y
                ────────
                     2
                 2LA

                 2    2
                X  - Y
                ────────
                     2
                 2LA

            - equilibria

                  u

                 0.0

                 0.0

                0.5u

                 0.0

            - relaxation parameters

                0.0

                0.4

                0.4

                1.0

                1.0
```

(continues on next page)

(continued from previous page)

```
- moments matrices

    1   1    1     1     1

        10          -10
    0   ──    0     ────   0
        LA          LA

              10          -10
    0   0     ──    0     ────
              LA          LA

         50   50    50    50
    0   ──── ──── ──── ────
          2    2     2     2
        LA   LA    LA    LA

         50  -50    50   -50
    0   ──── ──── ──── ────
          2    2     2     2
        LA   LA    LA    LA

- inverse of moments matrices

                        2
                     -LA
    1   0     0      ────     0
                       50

                        2       2
         LA             LA      LA
    0    ──    0       ──      ──
         20             200     200

                        2       2
               LA       LA     -LA
    0   0      ──       ──     ────
               20       200     200

                        2       2
        -LA             LA      LA
    0   ────    0       ──      ──
         20             200     200

                        2       2
              -LA       LA     -LA
    0   0     ────      ──     ────
               20       200     200
```

Run a simulation

Once the simulation is initialized, one time step can be performed by using the function one_time_step.

We compute the solution of the heat equation at $t = 0.1$. On the same graphic, we plot the initial condition, the exact

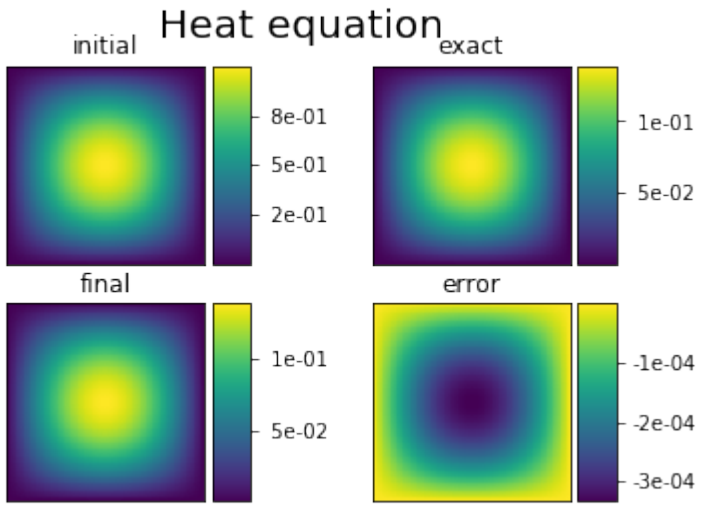
solution and the numerical solution.

[7]:
```
import numpy as np
import sympy as sp
import pylab as plt
%matplotlib inline
from mpl_toolkits.axes_grid1 import make_axes_locatable
import pylbm

u, X, Y = sp.symbols('u, X, Y')

def solution(x, y, t, k, l):
    return np.sin(k*np.pi*x)*np.sin(l*np.pi*y)*np.exp(-(k**2+l**2)*np.pi**2*mu*t)

def plot(i, j, z, title):
    im = axarr[i,j].imshow(z)
    divider = make_axes_locatable(axarr[i, j])
    cax = divider.append_axes("right", size="20%", pad=0.05)
    cbar = plt.colorbar(im, cax=cax, format='%6.0e')
    axarr[i, j].xaxis.set_visible(False)
    axarr[i, j].yaxis.set_visible(False)
    axarr[i, j].set_title(title)

# parameters
xmin, xmax, ymin, ymax = 0., 1., 0., 1.
N = 128
mu = 1.
Tf = .1
dx = (xmax-xmin)/N # spatial step
la = 1./dx
s1 = 2./(1+4*mu)
s2 = 1.
k, l = 1, 1 # number of the wave

dico = {
    'box': {'x':[xmin, xmax],
            'y':[ymin, ymax],
            'label': 0},
    'space_step': dx,
    'scheme_velocity': la,
    'schemes':[
        {
            'velocities': list(range(5)),
            'conserved_moments': u,
            'polynomials': [1, X/LA, Y/LA, (X**2+Y**2)/(2*LA**2), (X**2-Y**2)/
↪(2*LA**2)],
            'equilibrium': [u, 0., 0., .5*u, 0.],
            'relaxation_parameters': [0., s1, s1, s2, s2],
        }
    ],
    'init': {u: (solution, (0., k, l))},
    'boundary_conditions': {
        0: {'method': {0: pylbm.bc.AntiBounceBack,}},
    },
    'generator': 'cython',
    'parameters': {LA: la},
}
```

(continues on next page)

(continued from previous page)

```
sol = pylbm.Simulation(dico)
x = sol.domain.x
y = sol.domain.y

f, axarr = plt.subplots(2, 2)
f.suptitle('Heat equation', fontsize=20)

plot(0, 0, sol.m[u].copy(), 'initial')

while sol.t < Tf:
    sol.one_time_step()

sol.f2m()
z = sol.m[u]
ze = solution(x[:, np.newaxis], y[np.newaxis, :], sol.t, k, l)
plot(1, 0, z, 'final')
plot(0, 1, ze, 'exact')
plot(1, 1, z-ze, 'error')

plt.show()
```



2.6.5 Poiseuille flow

In this tutorial, we consider the classical $\mathrm{D_2Q_9}$ to simulate a Poiseuille flow modeling by the Navier-Stokes equations.

[1]:
```
%matplotlib inline
```

The $\mathrm{D_2Q_9}$ for Navier-Stokes

The $\mathrm{D_2Q_9}$ is defined by:

- a space step Δx and a time step Δt related to the scheme velocity λ by the relation $\lambda = \Delta x/\Delta t$,
- nine velocities $\{(0,0),(\pm 1,0),(0,\pm 1),(\pm 1,\pm 1)\}$, identified in pylbm by the numbers 0 to 8,

- nine polynomials used to build the moments

$$\{1, \lambda X, \lambda Y, 3E-4, (9E^2-21E+8)/2, 3XE-5X, 3YE-5Y, X^2-Y^2, XY\},$$

where $E = X^2+Y^2$.

- three conserved moments ρ, q_x, and q_y,
- nine relaxation parameters (three are 0 corresponding to conserved moments): $\{0, 0, 0, s_\mu, s_\mu, s_\eta, s_\eta, s_\eta, s_\eta\}$, where s_μ and s_η are in $(0, 2)$,
- equilibrium value of the non conserved moments

$$\begin{aligned}
m_3^e &= -2\rho + 3(q_x^2+q_y^2)/(\rho_0\lambda^2),\\
m_4^e &= \rho - 3(q_x^2+q_y^2)/(\rho_0\lambda^2),\\
m_5^e &= -q_x/\lambda,\\
m_6^e &= -q_y/\lambda,\\
m_7^e &= (q_x^2-q_y^2)/(\rho_0\lambda^2),\\
m_8^e &= q_xq_y/(\rho_0\lambda^2),
\end{aligned}$$

where ρ_0 is a given scalar.

This scheme is consistant at second order with the following equations (taken $\rho_0 = 1$)

$$\begin{aligned}
\partial_t\rho + \partial_x q_x + \partial_y q_y &= 0,\\
\partial_t q_x + \partial_x(q_x^2+p) + \partial_y(q_xq_y) &= \mu\partial_x(\partial_x q_x + \partial_y q_y) + \eta(\partial_{xx}+\partial_{yy})q_x,\\
\partial_t q_y + \partial_x(q_xq_y) + \partial_y(q_y^2+p) &= \mu\partial_y(\partial_x q_x + \partial_y q_y) + \eta(\partial_{xx}+\partial_{yy})q_y,
\end{aligned}$$

with $p = \rho\lambda^2/3$.

Build the simulation with pylbm

In the following, we build the dictionary of the simulation step by step.

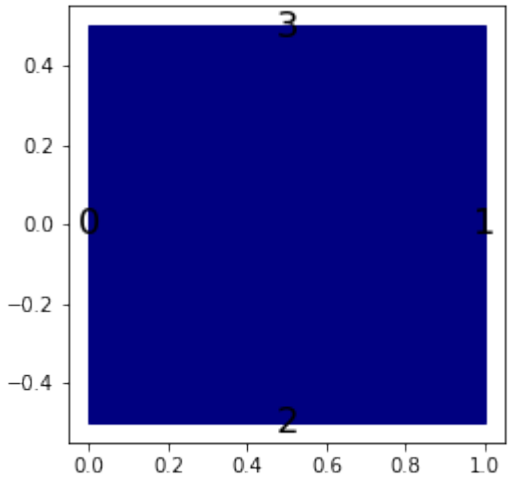
The geometry

The simulation is done on a rectangle of length L and width W. We can use $L = W = 1$.

We propose a dictionary that build the geometry of the domain. The labels of the bounds can be specified to different values for the moment.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

import pylbm

L, W = 1., 1.
dico_geom = {
    'box': {'x': [0, L],
            'y': [-.5*W, .5*W],
            'label':list(range(4))
           }
}
geom = pylbm.Geometry(dico_geom)
print(geom)
geom.visualize(viewlabel=True);
```

```
+---------------+
| Geometry information |
+---------------+
    - spatial dimension: 2
    - bounds of the box: [0. 1.] x [-0.5  0.5]
    - labels: [0, 1, 2, 3]
```



The stencil

The stencil of the D_2Q_9 is composed by the nine following velocities in 2D:

$$v_0 = (0,0),$$
$$v_1 = (1,0), \quad v_2 = (0,1), \quad v_3 = (-1,0), \quad v_4 = (0,-1),$$
$$v_5 = (1,1), \quad v_6 = (-1,1), \quad v_7 = (-1,-1), \quad v_8 = (1,-1).$$

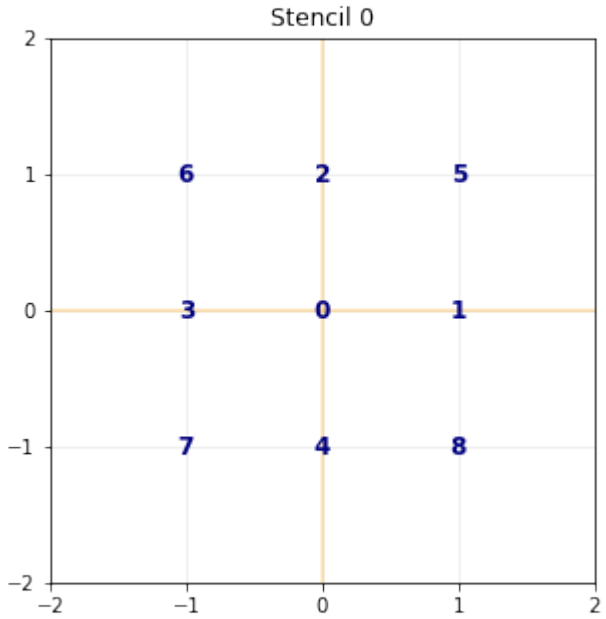
[3]:
```
dico_sten = {
    'dim': 2,
    'schemes': [
        {'velocities': list(range(9))}
    ],
}
sten = pylbm.Stencil(dico_sten)
print(sten)
sten.visualize();
```

```
+--------------+
| Stencil information |
+--------------+
    - spatial dimension: 2
    - minimal velocity in each direction: [-1 -1]
    - maximal velocity in each direction: [1 1]
    - information for each elementary stencil:
        stencil 0
            - number of velocities: 9
            - velocities
                (0: 0, 0)
```

(continues on next page)

(continued from previous page)

```
                (1: 1, 0)
                (2: 0, 1)
                (3: -1, 0)
                (4: 0, -1)
                (5: 1, 1)
                (6: -1, 1)
                (7: -1, -1)
                (8: 1, -1)
```



The domain

In order to build the domain of the simulation, the dictionary should contain the space step Δx and the stencils of the velocities (one for each scheme).

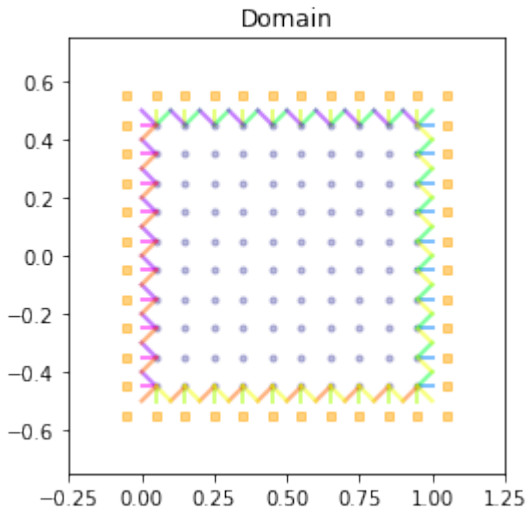
[4]:
```
dico_dom = {
    'space_step': .1,
    'box': {'x': [0, L],
            'y': [-.5*W, .5*W],
            'label': list(range(4))
           },
    'schemes': [
        {'velocities': list(range(9))}
    ],
}
dom = pylbm.Domain(dico_dom)
print(dom)
dom.visualize(view_distance=True);
```

```
+-------------+
| Domain information |
+-------------+
    - spatial dimension: 2
    - space step: 0.1
```

(continues on next page)

(continued from previous page)

```
    - with halo:
        bounds of the box: [-0.05 -0.55] x [1.05 0.55]
        number of points: [12, 12]
    - without halo:
        bounds of the box: [ 0.05 -0.45] x [0.95 0.45]
        number of points: [10, 10]

    +---------------+
    | Geometry information |
    +---------------+
        - spatial dimension: 2
        - bounds of the box: [0. 1.] x [-0.5  0.5]
        - labels: [0, 1, 2, 3]
```



The scheme

In pylbm, a simulation can be performed by using several coupled schemes. In this example, a single scheme is used and defined through a list of one single dictionary. This dictionary should contain:

- 'velocities': a list of the velocities
- 'conserved_moments': a list of the conserved moments as sympy variables
- 'polynomials': a list of the polynomials that define the moments
- 'equilibrium': a list of the equilibrium value of all the moments
- 'relaxation_parameters': a list of the relaxation parameters (0 for the conserved moments)
- 'init': a dictionary to initialize the conserved moments

(see the documentation for more details)

In order to fix the bulk (μ) and the shear (η) viscosities, we impose

$$s_\eta = \frac{2}{1+\eta d}, \qquad s_\mu = \frac{2}{1+\mu d}, \qquad d = \frac{6}{\lambda \rho_0 \Delta x}.$$

The scheme velocity could be taken to 1 and the inital value of ρ to $\rho_0 = 1$, q_x and q_y to 0.

In order to accelerate the simulation, we can use another generator. The default generator is Numpy (pure python). We can use for instance Cython that generates a more efficient code. This generator can be activated by using 'generator': pylbm.generator.CythonGenerator in the dictionary.

[5]:
```
import sympy as sp
X, Y, rho, qx, qy, LA = sp.symbols('X, Y, rho, qx, qy, LA')

# parameters
dx = 1./128  # spatial step
la = 1.      # velocity of the scheme
L = 1        # length of the domain
W = 1        # width of the domain
rhoo = 1.    # mean value of the density
mu   = 1.e-3 # shear viscosity
eta = 1.e-1 # bulk viscosity
# initialization
xmin, xmax, ymin, ymax = 0.0, L, -0.5*W, 0.5*W
dummy = 3.0/(la*rhoo*dx)
s_mu = 1.0/(0.5+mu*dummy)
s_eta = 1.0/(0.5+eta*dummy)
s_q = s_eta
s_es = s_mu
s  = [0.,0.,0.,s_mu,s_es,s_q,s_q,s_eta,s_eta]
dummy = 1./(LA**2*rhoo)
qx2 = dummy*qx**2
qy2 = dummy*qy**2
q2  = qx2+qy2
qxy = dummy*qx*qy

dico_sch = {
    'box': {'x': [xmin, xmax],
            'y': [ymin, ymax],
            'label':0
           },
    'space_step': dx,
    'scheme_velocity': la,
    'parameters': {LA: la},
    'schemes': [
        {
            'velocities': list(range(9)),
            'conserved_moments': [rho, qx, qy],
            'polynomials': [
                1, LA*X, LA*Y,
                3*(X**2+Y**2)-4,
                (9*(X**2+Y**2)**2-21*(X**2+Y**2)+8)/2,
                3*X*(X**2+Y**2)-5*X, 3*Y*(X**2+Y**2)-5*Y,
                X**2-Y**2, X*Y
            ],
            'relaxation_parameters': s,
            'equilibrium': [
                rho, qx, qy,
                -2*rho + 3*q2,
                rho-3*q2,
                -qx/LA, -qy/LA,
                qx2-qy2, qxy
            ],
        },
    ],
```

(continues on next page)

(continued from previous page)

```
}
sch = pylbm.Scheme(dico_sch)
print(sch)
```

```
+-------------+
| Scheme information |
+-------------+
    - spatial dimension: 2
    - number of schemes: 1
    - number of velocities: 9
    - conserved moments: [, qx, qy]

    +-------+
    | Scheme 0 |
    +-------+
        - velocities
            (0: 0, 0)
            (1: 1, 0)
            (2: 0, 1)
            (3: -1, 0)
            (4: 0, -1)
            (5: 1, 1)
            (6: -1, 1)
            (7: -1, -1)
            (8: 1, -1)

        - polynomials

                              1

                             LAX

                             LAY

                         2       2
                      3X  + 3Y  - 4

                                              2
                  2        2       2     2
              21X     21Y     9X  + Y
          - ──────  - ──────  + ──────────── + 4
              2        2            2

                         2    2
                      3XX  + Y  - 5X

                         2    2
                      3YX  + Y  - 5Y

                           2    2
                        X  - Y

                          XY

        - equilibria
```

(continues on next page)

(continued from previous page)

```
                  qx

                  qy

                    2          2
            3.0qx     3.0qy
       -2 + ──────── + ────────
                 2          2
               LA         LA

                    2          2
              3.0qx     3.0qy
         - ──────── - ────────
                 2          2
               LA         LA

                  -qx
                  ────
                   LA

                  -qy
                  ────
                   LA

                  2          2
            1.0qx     1.0qy
            ──────── - ────────
                2          2
              LA         LA

                 1.0qxqy
                 ─────────
                      2
                   LA

    - relaxation parameters

                 0.0

                 0.0

                 0.0

        1.13122171945701

        1.13122171945701

        0.025706940874036

        0.025706940874036

        0.025706940874036

        0.025706940874036

  - moments matrices
```

(continues on next page)

(continued from previous page)

```
    1   1   1    1    1   1    1    1    1

    0   LA  0   -LA   0   LA  -LA  -LA  LA

    0   0   LA   0   -LA  LA  LA   -LA  -LA

    -4  -1  -1  -1   -1   2    2    2    2

    4   -2  -2  -2   -2   1    1    1    1

    0   -2  0    2    0   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
```

- inverse of moments matrices

$$
\begin{pmatrix}
1/9 & 0 & 0 & -1/9 & 1/9 & 0 & 0 & 0 & 0 \\
1/9 & \frac{1}{6LA} & 0 & -1/36 & -1/18 & -1/6 & 0 & 1/4 & 0 \\
1/9 & 0 & \frac{1}{6LA} & -1/36 & -1/18 & 0 & -1/6 & -1/4 & 0 \\
1/9 & \frac{-1}{6LA} & 0 & -1/36 & -1/18 & 1/6 & 0 & 1/4 & 0 \\
1/9 & 0 & \frac{-1}{6LA} & -1/36 & -1/18 & 0 & 1/6 & -1/4 & 0 \\
1/9 & \frac{1}{6LA} & \frac{1}{6LA} & 1/18 & 1/36 & 1/12 & 1/12 & 0 & 1/4 \\
1/9 & \frac{-1}{6LA} & \frac{1}{6LA} & 1/18 & 1/36 & -1/12 & 1/12 & 0 & -1/4 \\
1/9 & \frac{-1}{6LA} & \frac{-1}{6LA} & 1/18 & 1/36 & -1/12 & -1/12 & 0 & 1/4 \\
1/9 & \frac{1}{6LA} & \frac{-1}{6LA} & 1/18 & 1/36 & 1/12 & -1/12 & 0 & -1/4
\end{pmatrix}
$$

Run the simulation

For the simulation, we take

- The domain $x \in (0, L)$ and $y \in (-W/2, W/2)$, $L = 2$, $W = 1$,
- the viscosities $\mu = 10^{-2} = \eta = 10^{-2}$,
- the space step $\Delta x = 1/128$, the scheme velocity $\lambda = 1$,
- the mean density $\rho_0 = 1$.

Concerning the boundary conditions, we impose the velocity on all the edges by a bounce-back condition with a source term that reads

$$q_x(x, y) = \rho_0 v_{\max}\Big(1 - \frac{4y^2}{W^2}\Big), \qquad q_y(x, y) = 0,$$

with $v_{\max} = 0.1$.

We compute the solution for $t \in (0, 50)$ and we plot several slices of the solution during the simulation.

This problem has an exact solution given by

$$q_x = \rho_0 v_{\max}\Big(1 - \frac{4y^2}{W^2}\Big), \qquad q_y = 0, \qquad p = p_0 + Kx,$$

where the pressure gradient K reads

$$K = -\frac{8 v_{\max}\eta}{W^2}.$$

We compute the exact and the numerical gradients of the pressure.

[6]:
```
X, Y, LA = sp.symbols('X, Y, LA')
rho, qx, qy = sp.symbols('rho, qx, qy')

def bc(f, m, x, y):
    m[qx] = rhoo * vmax * (1.-4.*y**2/W**2)
    m[qy] = 0.

def plot_coupe(sol):
    fig, ax1 = plt.subplots()
    ax2 = ax1.twinx()
    ax1.cla()
    ax2.cla()
    mx = int(sol.domain.shape_in[0]/2)
    my = int(sol.domain.shape_in[1]/2)
    x = sol.domain.x
    y = sol.domain.y
    u = sol.m[qx] / rhoo
    for i in [0,mx,-1]:
        ax1.plot(y+x[i], u[i, :], 'b')
    for j in [0,my,-1]:
        ax1.plot(x+y[j], u[:,j], 'b')
    ax1.set_ylabel('velocity', color='b')
    for tl in ax1.get_yticklabels():
        tl.set_color('b')
    ax1.set_ylim(-.5*rhoo*vmax, 1.5*rhoo*vmax)
    p = sol.m[rho][:,my] * la**2 / 3.0
    p -= np.average(p)
```

(continues on next page)

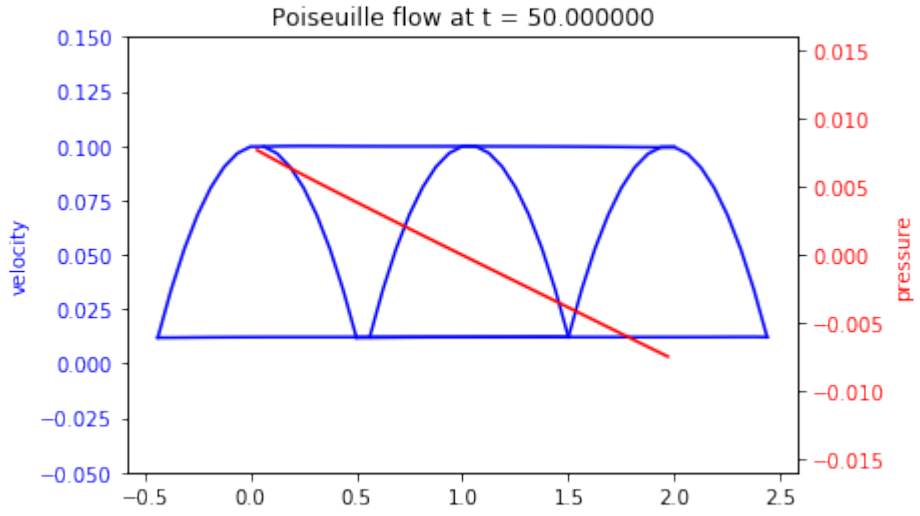
(continued from previous page)

```
    ax2.plot(x, p, 'r')
    ax2.set_ylabel('pressure', color='r')
    for tl in ax2.get_yticklabels():
        tl.set_color('r')
    ax2.set_ylim(pressure_gradient*L, -pressure_gradient*L)
    plt.title('Poiseuille flow at t = {0:f}'.format(sol.t))
    plt.draw()
    plt.pause(1.e-3)

# parameters
dx = 1./16  # spatial step
la = 1.      # velocity of the scheme
Tf = 50      # final time of the simulation
L = 2        # length of the domain
W = 1        # width of the domain
vmax = 0.1   # maximal velocity obtained in the middle of the channel
rhoo = 1.    # mean value of the density
mu = 1.e-2   # bulk viscosity
eta = 1.e-2  # shear viscosity
pressure_gradient = - vmax * 8.0 / W**2 * eta
# initialization
xmin, xmax, ymin, ymax = 0.0, L, -0.5*W, 0.5*W
dummy = 3.0/(la*rhoo*dx)
s_mu = 1.0/(0.5+mu*dummy)
s_eta = 1.0/(0.5+eta*dummy)
s_q = s_eta
s_es = s_mu
s  = [0.,0.,0.,s_mu,s_es,s_q,s_q,s_eta,s_eta]
dummy = 1./(LA**2*rhoo)
qx2 = dummy*qx**2
qy2 = dummy*qy**2
q2  = qx2+qy2
qxy = dummy*qx*qy

dico = {
    'box': {'x': [xmin, xmax],
            'y': [ymin, ymax],
            'label': 0
           },
    'space_step': dx,
    'scheme_velocity': la,
    'parameters': {LA: la},
    'schemes': [
        {
            'velocities': list(range(9)),
            'conserved_moments': [rho, qx, qy],
            'polynomials': [
                1, LA*X, LA*Y,
                3*(X**2+Y**2)-4,
                (9*(X**2+Y**2)**2-21*(X**2+Y**2)+8)/2,
                3*X*(X**2+Y**2)-5*X, 3*Y*(X**2+Y**2)-5*Y,
                X**2-Y**2, X*Y
            ],
            'relaxation_parameters': s,
            'equilibrium': [
                rho, qx, qy,
                -2*rho + 3*q2,
```

(continues on next page)

(continued from previous page)

```
                rho-3*q2,
                -qx/LA, -qy/LA,
                qx2-qy2, qxy
            ],
        },
    ],
    'init': {rho: rhoo,
             qx:0.,
             qy:0.
    },
    'boundary_conditions': {
        0: {'method': {0: pylbm.bc.BouzidiBounceBack}, 'value': bc}
    },
    'generator': 'cython',
}

sol = pylbm.Simulation(dico)
while (sol.t<Tf):
    sol.one_time_step()
plot_coupe(sol)
ny = int(sol.domain.shape_in[1]/2)
num_pressure_gradient = (sol.m[rho][-2,ny] - sol.m[rho][1,ny]) / (xmax-xmin) * la**2/␣
↪3.0
print("Exact pressure gradient    : {0:10.3e}".format(pressure_gradient))
print("Numerical pressure gradient: {0:10.3e}".format(num_pressure_gradient))
```




```
Exact pressure gradient    : -8.000e-03
Numerical pressure gradient: -7.074e-03
```

2.6.6 Lid driven cavity

In this tutorial, we consider the classical D_2Q_9 and D_3Q_{15} to simulate a lid driven acvity modeling by the Navier-Stokes equations. The D_2Q_9 is used in dimension 2 and the D_3Q_{15} in dimension 3.

[1]:
```
%matplotlib inline
```

The D_2Q_9 for Navier-Stokes

The D_2Q_9 is defined by:

- a space step Δx and a time step Δt related to the scheme velocity λ by the relation $\lambda = \Delta x/\Delta t$,
- nine velocities $\{(0,0),(\pm 1,0),(0,\pm 1),(\pm 1,\pm 1)\}$, identified in pylbm by the numbers 0 to 8,
- nine polynomials used to build the moments

$$\{1, \lambda X, \lambda Y, 3E-4, (9E^2-21E+8)/2, 3XE-5X, 3YE-5Y, X^2-Y^2, XY\},$$

where $E = X^2+Y^2$.

- three conserved moments ρ, q_x, and q_y,
- nine relaxation parameters (three are 0 corresponding to conserved moments): $\{0,0,0,s_\mu,s_\mu,s_\eta,s_\eta,s_\eta,s_\eta\}$, where s_μ and s_η are in $(0,2)$,
- equilibrium value of the non conserved moments

$$\begin{aligned}
m_3^e &= -2\rho + 3(q_x^2+q_y^2)/(\rho_0\lambda^2),\\
m_4^e &= \rho - 3(q_x^2+q_y^2)/(\rho_0\lambda^2),\\
m_5^e &= -q_x/\lambda,\\
m_6^e &= -q_y/\lambda,\\
m_7^e &= (q_x^2-q_y^2)/(\rho_0\lambda^2),\\
m_8^e &= q_xq_y/(\rho_0\lambda^2),
\end{aligned}$$

where ρ_0 is a given scalar.

This scheme is consistant at second order with the following equations (taken $\rho_0 = 1$)

$$\begin{aligned}
\partial_t\rho + \partial_x q_x + \partial_y q_y &= 0,\\
\partial_t q_x + \partial_x(q_x^2+p) + \partial_y(q_xq_y) &= \mu\partial_x(\partial_x q_x + \partial_y q_y) + \eta(\partial_{xx}+\partial_{yy})q_x,\\
\partial_t q_y + \partial_x(q_xq_y) + \partial_y(q_y^2+p) &= \mu\partial_y(\partial_x q_x + \partial_y q_y) + \eta(\partial_{xx}+\partial_{yy})q_y,
\end{aligned}$$

with $p = \rho\lambda^2/3$.

We write the dictionary for a simulation of the Navier-Stokes equations on $(0,1)^2$.

In order to impose the boundary conditions, we use the bounce-back conditions to fix $q_x = q_y = 0$ at south, east, and west and $q_x = \rho u$, $q_y = 0$ at north. The driven velocity u could be $u = \lambda/10$.

The solution is governed by the Reynolds number $Re = \rho_0 u/\eta$. We fix the relaxation parameters to have $Re = 1000$. The relaxation parameters related to the bulk viscosity μ should be large enough to ensure the stability (for instance $\mu = 10^{-3}$).

We compute the stationary solution of the problem obtained for large enough final time. We plot the solution with the function quiver of matplotlib.

[2]:
```
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
import pylbm
```

(continues on next page)

(continued from previous page)

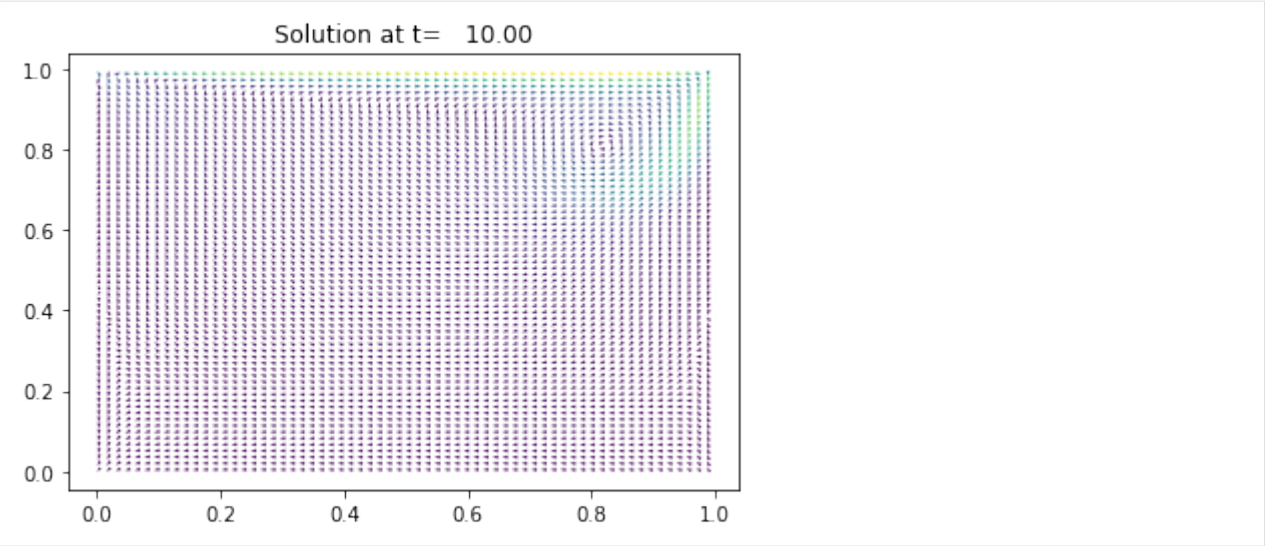
```
X, Y, LA = sp.symbols('X, Y, LA')
rho, qx, qy = sp.symbols('rho, qx, qy')

def bc(f, m, x, y):
    m[qx] = rhoo * vup

def plot(sol):
    pas = 2
    y, x = np.meshgrid(sol.domain.y[::pas], sol.domain.x[::pas])
    u = sol.m[qx][::pas,::pas] / sol.m[rho][::pas,::pas]
    v = sol.m[qy][::pas,::pas] / sol.m[rho][::pas,::pas]
    nv = np.sqrt(u**2+v**2)
    normu = nv.max()
    u = u / (nv+1e-5)
    v = v / (nv+1e-5)
    plt.quiver(x, y, u, v, nv, pivot='mid')
    plt.title('Solution at t={0:8.2f}'.format(sol.t))
    plt.show()

# parameters
Re = 1000
dx = 1./128  # spatial step
la = 1.      # velocity of the scheme
Tf = 10      # final time of the simulation
vup = la/5   # maximal velocity obtained in the middle of the channel
rhoo = 1.    # mean value of the density
mu = 1.e-4   # bulk viscosity
eta = rhoo*vup/Re  # shear viscosity
# initialization
xmin, xmax, ymin, ymax = 0., 1., 0., 1.
dummy = 3.0/(la*rhoo*dx)
s_mu = 1.0/(0.5+mu*dummy)
s_eta = 1.0/(0.5+eta*dummy)
s_q = s_eta
s_es = s_mu
s  = [0.,0.,0.,s_mu,s_es,s_q,s_q,s_eta,s_eta]
dummy = 1./(LA**2*rhoo)
qx2 = dummy*qx**2
qy2 = dummy*qy**2
q2  = qx2+qy2
qxy = dummy*qx*qy

print("Reynolds number: {0:10.3e}".format(Re))
print("Bulk viscosity : {0:10.3e}".format(mu))
print("Shear viscosity: {0:10.3e}".format(eta))
print("relaxation parameters: {0}".format(s))

dico = {
    'box': {'x': [xmin, xmax],
            'y': [ymin, ymax],
            'label': [0, 0, 0, 1]
           },
    'space_step': dx,
    'scheme_velocity': la,
    'parameters': {LA: la},
    'schemes': [
        {
```

(continues on next page)

(continued from previous page)

```
            'velocities': list(range(9)),
            'conserved_moments': [rho, qx, qy],
            'polynomials': [
                1, LA*X, LA*Y,
                3*(X**2+Y**2)-4,
                0.5*(9*(X**2+Y**2)**2-21*(X**2+Y**2)+8),
                3*X*(X**2+Y**2)-5*X, 3*Y*(X**2+Y**2)-5*Y,
                X**2-Y**2, X*Y
            ],
            'relaxation_parameters': s,
            'equilibrium': [
                rho, qx, qy,
                -2*rho + 3*q2,
                rho-3*q2,
                -qx/LA, -qy/LA,
                qx2-qy2, qxy
            ],
        },
    ],
    'init': {rho: rhoo,
             qx:0.,
             qy:0.
    },
    'boundary_conditions': {
        0: {'method': {0: pylbm.bc.BouzidiBounceBack}},
        1: {'method': {0: pylbm.bc.BouzidiBounceBack}, 'value': bc}
    },
    'generator': 'cython',
}

sol = pylbm.Simulation(dico)
while (sol.t<Tf):
    sol.one_time_step()
plot(sol)
```

```
Reynolds number:  1.000e+03
Bulk viscosity :  1.000e-04
Shear viscosity:  2.000e-04
relaxation parameters: [0.0, 0.0, 0.0, 1.8573551263001487, 1.8573551263001487, 1.
↪7337031900138697, 1.7337031900138697, 1.7337031900138697, 1.7337031900138697]
```



The $\mathrm{D_3Q_{15}}$ for Navier-Stokes

The $\mathrm{D_3Q_{15}}$ is defined by:

- a space step Δx and a time step Δt related to the scheme velocity λ by the relation $\lambda = \Delta x/\Delta t$,
- fifteen velocities $\{(0,0,0),(\pm 1,0,0),(0,\pm 1,0),(0,0,\pm 1),(\pm 1,\pm 1,\pm 1)\}$, identified in pylbm by the numbers $\{0,\dots,6,19,\dots,26\}$,
- fifteen polynomials used to build the moments

$$\{1, E-2, (15E^2-55E+32)/2, X, X(5E-13)/2, Y, Y(5E-13)/2, Z, Z(5E-13)/2, 3X^2-E, Y^2-Z^2, XY, YZ, ZX, XYZ$$

where $E = X^2+Y^2+Z^2$.

- four conserved moments ρ, q_x, q_y, and q_z,
- fifteen relaxation parameters (four are 0 corresponding to conserved moments): $\{0, s_1, s_2, 0, s_4, 0, s_4, 0, s_4, s_9, s_9, s_{11}, s_{11}, s_{11}, s_{14}\}$,
- equilibrium value of the non conserved moments

$$\begin{aligned}
m_1^e &= -\rho + q_x^2 + q_y^2 + q_z^2,\\
m_2^e &= -\rho,\\
m_4^e &= -7q_x/3,\\
m_6^e &= -7q_y/3,\\
m_8^e &= -7q_z/3,\\
m_9^e &= (2q_x^2 - (q_y^2+q_z^2))/3,\\
m_{10}^e &= q_y^2 - q_z^2,\\
m_{11}^e &= q_x q_y,\\
m_{12}^e &= q_y q_z,\\
m_{13}^e &= q_z q_x,\\
m_{14}^e &= 0.
\end{aligned}$$

This scheme is consistant at second order with the Navier-Stokes equations with the shear viscosity η and the relaxation parameter s_9 linked by the relation

$$s_9 = \frac{2}{1 + 6\eta/\Delta x}.$$

We write a dictionary for a simulation of the Navier-Stokes equations on $(0, 1)^3$.

In order to impose the boundary conditions, we use the bounce-back conditions to fix $q_x = q_y = q_z = 0$ at south, north, east, west, and bottom and $q_x = \rho u$, $q_y = q_z = 0$ at top. The driven velocity u could be $u = \lambda/10$.

We compute the stationary solution of the problem obtained for large enough final time. We plot the solution with the function quiver of matplotlib.

[3]:
```
X, Y, Z, LA = sp.symbols('X, Y, Z, LA')
rho, qx, qy, qz = sp.symbols('rho, qx, qy, qz')

def bc(f, m, x, y, z):
    m[qx] = rhoo * vup

def plot(sol):
    plt.clf()
    pas = 4
    nz = int(sol.domain.shape_in[1] / 2) + 1
    y, x = np.meshgrid(sol.domain.y[::pas], sol.domain.x[::pas])
    u = sol.m[qx][::pas,nz,::pas] / sol.m[rho][::pas,nz,::pas]
    v = sol.m[qz][::pas,nz,::pas] / sol.m[rho][::pas,nz,::pas]
    nv = np.sqrt(u**2+v**2)
    normu = nv.max()
    u = u / (nv+1e-5)
    v = v / (nv+1e-5)
    plt.quiver(x, y, u, v, nv, pivot='mid')
    plt.title('Solution at t={0:9.3f}'.format(sol.t))
    plt.show()

# parameters
Re = 2000
dx = 1./64   # spatial step
la = 1.      # velocity of the scheme
Tf = 3       # final time of the simulation
vup = la/10  # maximal velocity obtained in the middle of the channel
rhoo = 1.    # mean value of the density
eta = rhoo*vup/Re  # shear viscosity
# initialization
xmin, xmax, ymin, ymax, zmin, zmax = 0., 1., 0., 1., 0., 1.
dummy = 3.0/(la*rhoo*dx)

s1 = 1.6
s2 = 1.2
s4 = 1.6
s9 = 1./(.5+dummy*eta)
s11 = s9
s14 = 1.2
s  = [0, s1, s2, 0, s4, 0, s4, 0, s4, s9, s9, s11, s11, s11, s14]

r = X**2+Y**2+Z**2

print("Reynolds number: {0:10.3e}".format(Re))
print("Shear viscosity: {0:10.3e}".format(eta))
```

(continues on next page)

(continued from previous page)

```
dico = {
    'box':{
        'x': [xmin, xmax],
        'y': [ymin, ymax],
        'z': [zmin, zmax],
        'label': [0, 0, 0, 0, 0, 1]
    },
    'space_step': dx,
    'scheme_velocity': la,
    'parameters': {LA: la},
    'schemes': [
        {
            'velocities': list(range(7)) + list(range(19,27)),
            'conserved_moments': [rho, qx, qy, qz],
            'polynomials': [
                1,
                r - 2, .5*(15*r**2-55*r+32),
                X, .5*(5*r-13)*X,
                Y, .5*(5*r-13)*Y,
                Z, .5*(5*r-13)*Z,
                3*X**2-r, Y**2-Z**2,
                X*Y, Y*Z, Z*X,
                X*Y*Z
            ],
            'relaxation_parameters': s,
            'equilibrium': [
                rho,
                -rho + qx**2 + qy**2 + qz**2,
                -rho,
                qx,
                -7./3*qx,
                qy,
                -7./3*qy,
                qz,
                -7./3*qz,
                1./3*(2*qx**2-(qy**2+qz**2)),
                qy**2-qz**2,
                qx*qy,
                qy*qz,
                qz*qx,
                0
            ],
        },
    ],
    'init': {rho: rhoo,
             qx: 0.,
             qy: 0.,
             qz: 0.
    },
    'boundary_conditions':{
        0: {'method': {0: pylbm.bc.BouzidiBounceBack}},
        1: {'method': {0: pylbm.bc.BouzidiBounceBack}, 'value': bc}
    },
    'generator': 'cython',
}
```

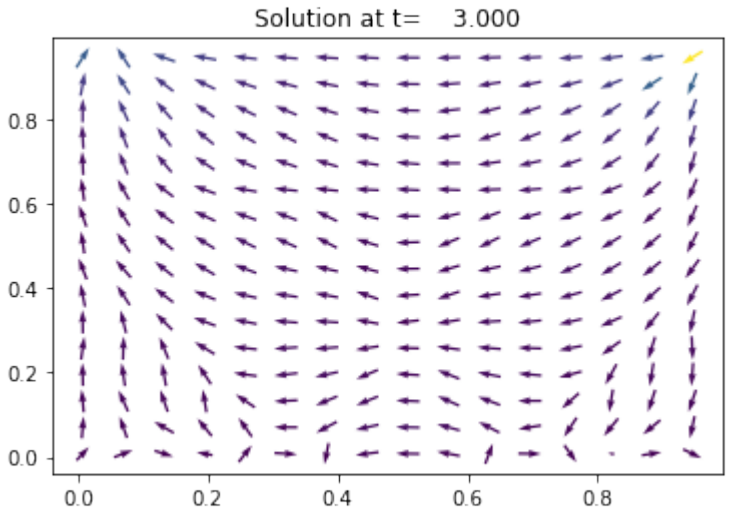
(continues on next page)

(continued from previous page)

```
sol = pylbm.Simulation(dico)
while (sol.t<Tf):
    sol.one_time_step()
plot(sol)
```

```
Reynolds number:  2.000e+03
Shear viscosity:  5.000e-05
```

```
[0] WARNING  pylbm.scheme in function _check_inverse line 384
Problem M * invM is not identity !!!
```



2.6.7 Von Karman vortex street

In this tutorial, we consider the classical $\mathrm{D_2Q_9}$ to simulate the Von Karman vortex street modeling by the Navier-Stokes equations.

In fluid dynamics, a Von Karman vortex street is a repeating pattern of swirling vortices caused by the unsteady separation of flow of a fluid around blunt bodies. It is named after the engineer and fluid dynamicist Theodore von Karman. For the simulation, we propose to simulate the Navier-Stokes equation into a rectangular domain with a circular hole of diameter d.

The $\mathrm{D_2Q_9}$ is defined by:

- a space step Δx and a time step Δt related to the scheme velocity λ by the relation $\lambda = \Delta x/\Delta t$,
- nine velocities $\{(0,0),(\pm 1,0),(0,\pm 1),(\pm 1,\pm 1)\}$, identified in pylbm by the numbers 0 to 8,
- nine polynomials used to build the moments

$$\{1, \lambda X, \lambda Y, 3E-4, (9E^2-21E+8)/2, 3XE-5X, 3YE-5Y, X^2-Y^2, XY\},$$

where $E = X^2+Y^2$.

- three conserved moments ρ, q_x, and q_y,
- nine relaxation parameters (three are 0 corresponding to conserved moments): $\{0,0,0,s_\mu,s_\mu,s_\eta,s_\eta,s_\eta,s_\eta\}$, where s_μ and s_η are in $(0,2)$,
- equilibrium value of the non conserved moments

$$
\begin{aligned}
m_3^e &= -2\rho + 3(q_x^2+q_y^2)/(\rho_0\lambda^2),\\
m_4^e &= \rho - 3(q_x^2+q_y^2)/(\rho_0\lambda^2),\\
m_5^e &= -q_x/\lambda,\\
m_6^e &= -q_y/\lambda,\\
m_7^e &= (q_x^2-q_y^2)/(\rho_0\lambda^2),\\
m_8^e &= q_xq_y/(\rho_0\lambda^2),
\end{aligned}
$$

where ρ_0 is a given scalar.

This scheme is consistant at second order with the following equations (taken $\rho_0 = 1$)

$$
\begin{aligned}
\partial_t\rho + \partial_x q_x + \partial_y q_y &= 0,\\
\partial_t q_x + \partial_x(q_x^2+p) + \partial_y(q_xq_y) &= \mu\partial_x(\partial_x q_x + \partial_y q_y) + \eta(\partial_{xx}+\partial_{yy})q_x,\\
\partial_t q_y + \partial_x(q_xq_y) + \partial_y(q_y^2+p) &= \mu\partial_y(\partial_x q_x + \partial_y q_y) + \eta(\partial_{xx}+\partial_{yy})q_y,
\end{aligned}
$$

with $p = \rho\lambda^2/3$.

We write a dictionary for a simulation of the Navier-Stokes equations on $(0,1)^2$.

In order to impose the boundary conditions, we use the bounce-back conditions to fix $q_x = q_y = \rho v_0$ at south, east, and north where the velocity v_0 could be $v_0 = \lambda/20$. At west, we impose the simple output condition of Neumann by repeating the second to last cells into the last cells.

The solution is governed by the Reynolds number $Re = \rho_0 v_0 d/\eta$, where d is the diameter of the circle. Fix the relaxation parameters to have $Re = 500$. The relaxation parameters related to the bulk viscosity μ should be large enough to ensure the stability (for instance $\mu = 10^{-3}$).

We compute the stationary solution of the problem obtained for large enough final time. We plot the vorticity of the solution with the function imshow of matplotlib.

[1]:
```
%matplotlib inline
```

[2]:
```
import numpy as np
import sympy as sp
import pylbm

X, Y, LA = sp.symbols('X, Y, LA')
rho, qx, qy = sp.symbols('rho, qx, qy')

def bc_in(f, m, x, y):
    m[qx] = rhoo * v0

def vorticity(sol):
    ux = sol.m[qx] / sol.m[rho]
    uy = sol.m[qy] / sol.m[rho]
    V = np.abs(uy[2:,1:-1] - uy[0:-2,1:-1] - ux[1:-1,2:] + ux[1:-1,0:-2])/(2*sol.
↪domain.dx)
    return -V

# parameters
rayon = 0.05
Re = 500
dx = 1./64   # spatial step
la = 1.      # velocity of the scheme
Tf = 75      # final time of the simulation
v0 = la/20   # maximal velocity obtained in the middle of the channel
rhoo = 1.    # mean value of the density
```

(continues on next page)

(continued from previous page)

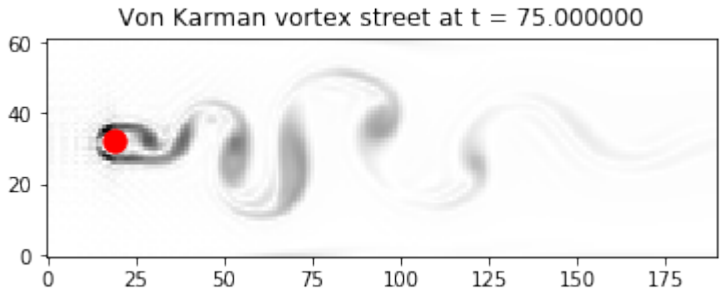
```
mu = 1.e-3   # bulk viscosity
eta = rhoo*v0*2*rayon/Re  # shear viscosity
# initialization
xmin, xmax, ymin, ymax = 0., 3., 0., 1.
dummy = 3.0/(la*rhoo*dx)
s_mu = 1.0/(0.5+mu*dummy)
s_eta = 1.0/(0.5+eta*dummy)
s_q = s_eta
s_es = s_mu
s  = [0.,0.,0.,s_mu,s_es,s_q,s_q,s_eta,s_eta]
dummy = 1./(LA**2*rhoo)
qx2 = dummy*qx**2
qy2 = dummy*qy**2
q2  = qx2+qy2
qxy = dummy*qx*qy

print("Reynolds number: {0:10.3e}".format(Re))
print("Bulk viscosity : {0:10.3e}".format(mu))
print("Shear viscosity: {0:10.3e}".format(eta))
print("relaxation parameters: {0}".format(s))

dico = {
    'box': {'x': [xmin, xmax],
            'y': [ymin, ymax],
            'label': [0, 2, 0, 0]
           },
    'elements': [pylbm.Circle([.3, 0.5*(ymin+ymax)+dx], rayon, label=1)],
    'space_step': dx,
    'scheme_velocity': la,
    'parameters': {LA: la},
    'schemes': [
        {
            'velocities': list(range(9)),
            'conserved_moments': [rho, qx, qy],
            'polynomials': [
                1, LA*X, LA*Y,
                3*(X**2+Y**2)-4,
                (9*(X**2+Y**2)**2-21*(X**2+Y**2)+8)/2,
                3*X*(X**2+Y**2)-5*X, 3*Y*(X**2+Y**2)-5*Y,
                X**2-Y**2, X*Y
            ],
            'relaxation_parameters': s,
            'equilibrium': [
                rho, qx, qy,
                -2*rho + 3*q2,
                rho-3*q2,
                -qx/LA, -qy/LA,
                qx2-qy2, qxy
            ],
        },
    ],
    'init': {rho:rhoo,
             qx:0.,
             qy:0.
    },
    'boundary_conditions': {
        0: {'method': {0: pylbm.bc.BouzidiBounceBack}, 'value': bc_in},
```

(continues on next page)

(continued from previous page)

```
        1: {'method': {0: pylbm.bc.BouzidiBounceBack}},
        2: {'method': {0: pylbm.bc.NeumannX}},
    },
    'generator': 'cython',
}

sol = pylbm.Simulation(dico)
while sol.t < Tf:
    sol.one_time_step()

viewer = pylbm.viewer.matplotlib_viewer
fig = viewer.Fig()
ax = fig[0]
im = ax.image(vorticity(sol).transpose(), clim = [-3., 0])
ax.ellipse([.3/dx, 0.5*(ymin+ymax)/dx], [rayon/dx,rayon/dx], 'r')
ax.title = 'Von Karman vortex street at t = {0:f}'.format(sol.t)
fig.show()
```

```
Reynolds number:  5.000e+02
Bulk viscosity :  1.000e-03
Shear viscosity:  1.000e-05
relaxation parameters: [0.0, 0.0, 0.0, 1.4450867052023122, 1.4450867052023122, 1.
↪9923493783869939, 1.9923493783869939, 1.9923493783869939, 1.9923493783869939]
```



2.6.8 Transport equation with source term

In this tutorial, we propose to add a source term in the advection equation. The problem reads

$$\partial_t u + c\partial_x u = S(t, x, u), \quad t > 0, , \quad x \in (0, 1),$$

where c is a constant scalar (typically $c = 1$). Additional boundary and initial conditions will be given in the following. S is the source term that can depend on the time t, the space x and the solution u.

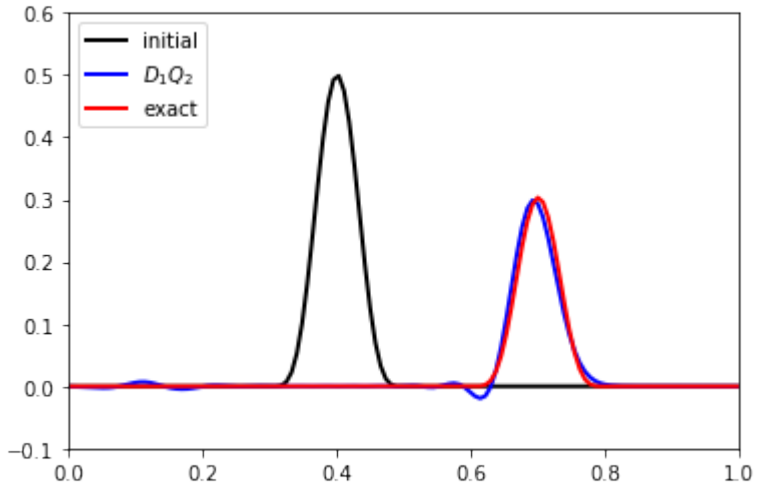
In order to simulate this problem, we use the D_1Q_2 scheme and we add an additional `key:value` in the dictionary for the source term. We deal with two examples.

A friction term

In this example, we takes $S(t, x, u) = -\alpha u$ where α is a positive constant. The dictionary of the simulation then reads:

[1]:
```
%matplotlib inline
import sympy as sp
import numpy as np
import pylbm
```

[2]:
```
C, ALPHA, X, u, LA = sp.symbols('C, ALPHA, X, u, LA')
c = 0.3
alpha = 0.5

def init(x):
    middle, width, height = 0.4, 0.1, 0.5
    return height/width**10 * (x%1-middle-width)**5 * \
                               (middle-x%1-width)**5 * (abs(x%1-middle)<=width)

def solution(t, x):
    return init(x - c*t)*np.exp(-alpha*t)

dico = {
    'box': {'x': [0., 1.], 'label': -1},
    'space_step': 1./128,
    'scheme_velocity': LA,
    'schemes': [
        {
            'velocities': [1,2],
            'conserved_moments': u,
            'polynomials': [1, LA*X],
            'relaxation_parameters': [0., 2.],
            'equilibrium': [u, C*u],
            'source_terms': {u: -ALPHA*u},
        },
    ],
    'init': {u: init},
    'parameters': {
        LA: 1.,
        C: c,
        ALPHA: alpha
    },
    'generator': 'numpy',
}

sol = pylbm.Simulation(dico) # build the simulation
viewer = pylbm.viewer.matplotlib_viewer
fig = viewer.Fig()
ax = fig[0]
ax.axis(0., 1., -.1, .6)
x = sol.domain.x
ax.plot(x, sol.m[u], width=2, color='k', label='initial')
while sol.t < 1:
    sol.one_time_step()
ax.plot(x, sol.m[u], width=2, color='b', label=r'$D_1Q_2$')
ax.plot(x, solution(sol.t, x), width=2, color='r', label='exact')
ax.legend()
```



A source term depending on time and space

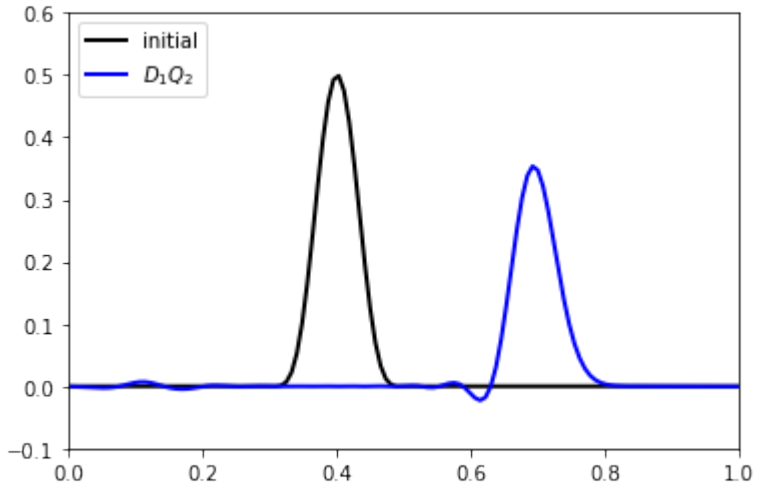
If the source term S depends explicitely on the time or on the space, we have to specify the corresponding variables in the dictionary through the key *parameters*. The time variable is prescribed by the key *'time'*. Moreover, sympy functions can be used to define the source term like in the following example. This example is just for testing the feature... no physical meaning in mind !

[3]:
```
t, C, X, u, LA = sp.symbols('t, C, X, u, LA')
c = 0.3

def init(x):
    middle, width, height = 0.4, 0.1, 0.5
    return height/width**10 * (x%1-middle-width)**5 * \
                               (middle-x%1-width)**5 * (abs(x%1-middle)<=width)

dico = {
    'box': {'x': [0., 1.], 'label': -1},
    'space_step': 1./128,
    'scheme_velocity': LA,
    'schemes': [
        {
            'velocities': [1, 2],
            'conserved_moments': u,
            'polynomials': [1, LA*X],
            'relaxation_parameters': [0., 2.],
            'equilibrium': [u, C*u],
            'source_terms': {u: -sp.Abs(X-t)**2*u},
        },
    ],
    'init': {u: init},
    'generator': 'cython',
    'parameters': {LA: 1., C: c},
}

sol = pylbm.Simulation(dico) # build the simulation
viewer = pylbm.viewer.matplotlib_viewer
fig = viewer.Fig()
```

(continues on next page)

(continued from previous page)

```
ax = fig[0]
ax.axis(0., 1., -.1, .6)
x = sol.domain.x
ax.plot(x, sol.m[u], width=2, color='k', label='initial')
while sol.t < 1:
    sol.one_time_step()
ax.plot(x, sol.m[u], width=2, color='b', label=r'$D_1Q_2$')
ax.legend()
```



get the notebook

Transport in 1D

In this tutorial, we will show how to implement from scratch a very basic lattice Boltzmann scheme: the D_1Q_2 for the advection equation and for Burger's equation.

get the notebook

The wave equation in 1D

In this tutorial, we will show how to implement from scratch a very basic lattice Boltzmann scheme: the D_1Q_3 for the waves equation.

get the notebook

The heat equation in 1D

In this tutorial, we present the D_1Q_3 to solve the heat equation in 1D by using pylbm.

get the notebook

The heat equation in 2D

In this tutorial, we present the D_2Q_5 to solve the heat equation in 2D by using pylbm.

get the notebook

Poiseuille flow

In this tutorial, we present the D_2Q_9 for Navier-Stokes equation to solve the Poiseuille flow in 2D by using pylbm.

get the notebook

Lid driven cavity

In this tutorial, we present the D_2Q_9 for Navier-Stokes equation to solve the lid driven cavity in 2D and the $D3Q15$ in 3D by using pylbm.

`get the notebook`

Von Karman vortex street

In this tutorial, we present the D_2Q_9 for Navier-Stokes equation to solve the Von Karman vortex street in 2D by using pylbm.

`get the notebook`

Transport equation with source term

In this tutorial, we will show how to implement with pylbm the D_1Q_2 for the advection equation with a source term.

You can also find other examples in the gallery.

CHAPTER

THREE

DOCUMENTATION OF THE CODE

The most important classes

Geometry(dico[, need_validation])	Create a geometry that defines the fluid part and the solid part.
Domain(dico[, need_validation])	Create a domain that defines the fluid part and the solid part and computes the distances between these two states.
Scheme(dico[, check_inverse, need_validation])	Create the class with all the needed informations for each elementary scheme.
Simulation(dico[, sorder, dtype, check_inverse])	create a class simulation

3.1 pylbm.Geometry

class pylbm.**Geometry**(*dico*, *need_validation=True*)

Create a geometry that defines the fluid part and the solid part.

Parameters

dico [dict]

dictionary that contains the following *key:value*

- box : a dictionary for the definition of the computed box
- elements : a list of elements (optional)

need_validation [bool] boolean to specify if the dictionary has to be validated (optional)

Notes

The dictionary that defines the box should contains the following *key:value*

- x : a list of the bounds in the first direction
- y : a list of the bounds in the second direction (optional)
- z : a list of the bounds in the third direction (optional)
- **label** [an integer or a list of integers] (length twice the number of dimensions) used to label each edge (optional)

Examples

see demo/examples/geometry/

Attributes

dim [int] number of spatial dimensions (1, 2, or 3)

bounds [ndarray] the bounds of the box in each spatial direction

box_label [list]

a list of the four labels for the left, right, bottom, top, front, and back edges

list_elem [list] a list that contains each element added or deleted in the box

Methods

add_elem(self, elem)	add a solid or a fluid part in the domain
list_of_elements_labels(self)	Get the list of all the labels used in the geometry.
list_of_labels(self)	Get the list of all the labels used in the geometry.
visualize(self[, viewer_app, figsize, ...])	plot a view of the geometry

3.1.1 pylbm.Geometry.add_elem

method

Geometry.**add_elem**(*self*, *elem*)

add a solid or a fluid part in the domain

Parameters

elem [Element] a geometric element to add (or to del)

3.1.2 pylbm.Geometry.list_of_elements_labels

method

Geometry.**list_of_elements_labels**(*self*)

Get the list of all the labels used in the geometry.

3.1.3 pylbm.Geometry.list_of_labels

method

Geometry.**list_of_labels**(*self*)

Get the list of all the labels used in the geometry.

3.1.4 pylbm.Geometry.visualize

method

Geometry.**visualize**(*self*, *viewer_app=<module 'pylbm.viewer.matplotlib_viewer' from '/home/docs/checkouts/readthedocs.org/user_builds/pylbm/conda/0.4.1/lib/python3.6/site-packages/pylbm-0.4.1-py3.6.egg/pylbm/viewer/matplotlib_viewer.py'>*, *figsize=(6, 4)*, *viewlabel=False*, *fluid_color='navy'*, *viewgrid=False*, *alpha=1.0*)

plot a view of the geometry

Parameters

viewer_app [Viewer] a viewer (default matplotlib_viewer)

viewlabel [boolean] activate the labels mark (default False)

fluid_color [color] color for the fluid part (default blue)

figsize [tuple] the size of the figure (default (6, 4))

viewgrid [bool] view the grid (default False)

alpha [double] transparency between 0 and 1 (default 1)

Returns

object views

3.2 pylbm.Domain

class pylbm.**Domain**(*dico*, *need_validation=True*)

Create a domain that defines the fluid part and the solid part and computes the distances between these two states.

Parameters

dico [dictionary] that contains the following *key:value*

- box : a dictionary that defines the computational box
- **elements** [the list of the elements] (available elements are given in the module `elements`)
- space_step : the spatial step
- schemes : a list of dictionaries,

each of them defining a elementary `Scheme` we only need the velocities to define a domain

need_validation [bool] boolean to specify if the dictionary has to be validated (optional)

Warning: the sizes of the box must be a multiple of the space step dx

Notes

The dictionary that defines the box should contains the following *key:value*

- x : a list of the bounds in the first direction
- y : a list of the bounds in the second direction (optional)
- z : a list of the bounds in the third direction (optional)

- label : an integer or a list of integers (length twice the number of dimensions) used to label each edge (optional)

See *Geometry* for more details.

In 1D, distance[q, i] is the distance between the point x[i] and the border in the direction of the qth velocity.

In 2D, distance[q, j, i] is the distance between the point (x[i], y[j]) and the border in the direction of qth velocity

In 3D, distance[q, k, j, i] is the distance between the point (x[i], y[j], z[k]) and the border in the direction of qth velocity

In 1D, flag[q, i] is the flag of the border reached by the point x[i] in the direction of the qth velocity

In 2D, flag[q, j, i] is the flag of the border reached by the point (x[i], y[j]) in the direction of qth velocity

In 2D, flag[q, k, j, i] is the flag of the border reached by the point (x[i], y[j], z[k]) in the direction of qth velocity

Examples

```
>>> dico = {'box': {'x': [0, 1], 'label': 0},
...         'space_step': 0.1,
...         'schemes': [{'velocities': list(range(3))}],
...        }
>>> dom = Domain(dico)
>>> dom
+--------------------+
| Domain information |
+--------------------+
    - spatial dimension: 1
    - space step: 0.1
    - with halo:
        bounds of the box: [-0.05] x [1.05]
        number of points: [12]
    - without halo:
        bounds of the box: [0.05] x [0.95]
        number of points: [10]
<BLANKLINE>
    +----------------------+
    | Geometry information |
    +----------------------+
        - spatial dimension: 1
        - bounds of the box: [0. 1.]
```

```
>>> dico = {'box': {'x': [0, 1], 'y': [0, 1], 'label': [0, 0, 1, 1]},
...         'space_step': 0.1,
...         'schemes': [{'velocities': list(range(9))},
...                     {'velocities': list(range(5))}
...                    ],
...        }
>>> dom = Domain(dico)
>>> dom
+--------------------+
| Domain information |
+--------------------+
    - spatial dimension: 2
    - space step: 0.1
    - with halo:
```

(continues on next page)

(continued from previous page)

```
        bounds of the box: [-0.05 -0.05] x [1.05 1.05]
        number of points: [12, 12]
    - without halo:
        bounds of the box: [0.05 0.05] x [0.95 0.95]
        number of points: [10, 10]
<BLANKLINE>
    +----------------------+
    | Geometry information |
    +----------------------+
        - spatial dimension: 2
        - bounds of the box: [0. 1.] x [0. 1.]
```

see demo/examples/domain/

Attributes

dim [int] number of spatial dimensions (example: 1, 2, or 3)

globalbounds [ndarray] the bounds of the box in each spatial direction

bounds [ndarray] the local bounds of the process in each spatial direction

dx [double] space step (example: 0.1, 1.e-3)

type [string] type of data (example: 'float64')

stencil [Stencil] the stencil of the velocities (object of the class *Stencil*)

global_size [list] number of points in each direction

extent [list] number of points to add on each side (max velocities)

coords [ndarray] coordinates of the domain

x [ndarray] x component of the coordinates in the interior domain.

y [ndarray] y component of the coordinates in the interior domain.

z [ndarray] z component of the coordinates in the interior domain.

in_or_out [ndarray] defines the fluid and the solid part (fluid: value=valin, solid: value=valout)

distance [ndarray] defines the distances to the borders. The distance is scaled by dx and is not equal to valin only for the points that reach the border with the specified velocity.

flag [ndarray] NumPy array that defines the flag of the border reached with the specified velocity

valin [int] value in the fluid domain

valout [int] value in the fluid domain

x_halo [ndarray] x component of the coordinates of the whole domain

y_halo [ndarray] y component of the coordinates of the whole domain

z_halo [ndarray] z component of the coordinates of the whole domain

shape_halo [list] shape of the whole domain with the halo points.

shape_in shape of the interior domain.

Methods

construct_mpi_topology(self, dico)	Create the mpi topology
create_coords(self)	Create the coordinates of the interior domain and the whole domain with halo points.
get_bounds(self)	Return the coordinates of the bottom right and upper left corner of the interior domain.
get_bounds_halo(self)	Return the coordinates of the bottom right and upper left corner of the whole domain with halo points.
list_of_labels(self)	Get the list of all the labels used in the geometry.
visualize(self[, viewer_app, view_distance, ...])	Visualize the domain by creating a plot.

3.2.1 pylbm.Domain.construct_mpi_topology

method

Domain.**construct_mpi_topology**(*self*, *dico*)
Create the mpi topology

3.2.2 pylbm.Domain.create_coords

method

Domain.**create_coords**(*self*)
Create the coordinates of the interior domain and the whole domain with halo points.

3.2.3 pylbm.Domain.get_bounds

method

Domain.**get_bounds**(*self*)
Return the coordinates of the bottom right and upper left corner of the interior domain.

3.2.4 pylbm.Domain.get_bounds_halo

method

Domain.**get_bounds_halo**(*self*)
Return the coordinates of the bottom right and upper left corner of the whole domain with halo points.

3.2.5 pylbm.Domain.list_of_labels

method

Domain.**list_of_labels**(*self*)
Get the list of all the labels used in the geometry.

3.2.6 pylbm.Domain.visualize

method

Domain.**visualize**(*self*, *viewer_app=<module 'pylbm.viewer.matplotlib_viewer' from '/home/docs/checkouts/readthedocs.org/user_builds/pylbm/conda/0.4.1/lib/python3.6/site-packages/pylbm-0.4.1-py3.6.egg/pylbm/viewer/matplotlib_viewer.py'>*, *view_distance=False*, *view_in=True*, *view_out=True*, *view_bound=False*, *label=None*, *scale=1*)

Visualize the domain by creating a plot.

Parameters

viewer_app [Viewer, optional] define the viewer to plot the domain default is viewer.matplotlib_viewer

view_distance [boolean or int or list, optional] view the distance between the interior points and the border default is False if True, then all velocities are considered can specify some specific velocities in a list

view_in [boolean, optional] view the inner points default is True

view_out [boolean, optional] view the outer points default is True

view_bound [boolean or int or list, optional] view the points on the bounds default is False

label [int or list, optional] view the distance only for the specified labels

scale [int or float, optional] scale used for the symbol (default 1)

Returns

object views

3.3 pylbm.Scheme

class pylbm.**Scheme**(*dico*, *check_inverse=False*, *need_validation=True*)

Create the class with all the needed informations for each elementary scheme.

Parameters

dico [a dictionary that contains the following *key:value*]

- dim : spatial dimension (optional if the *box* is given)
- scheme_velocity : the value of the ratio space step over time step (la = dx / dt)
- schemes : a list of dictionaries, one for each scheme

Notes

Each dictionary of the list *schemes* should contains the following *key:value*

- velocities : list of the velocities number
- conserved moments : list of the moments conserved by each scheme
- polynomials : list of the polynomial functions that define the moments
- equilibrium : list of the values that define the equilibrium
- relaxation_parameters : list of the value of the relaxation parameters
- source_terms : dictionary do define the source terms (optional, see examples)
- init : dictionary to define the initial conditions (see examples)

If the stencil has already been computed, it can be pass in argument.

Examples

see demo/examples/scheme/

Attributes

dim [int] spatial dimension

dx [double] space step

dt [double] time step

la [double] scheme velocity, ratio dx/dt

nschemes [int] number of elementary schemes

stencil [object of class *Stencil*] a stencil of velocities

P [list of sympy matrix] list of polynomials that define the moments

EQ [list of sympy matrix] list of the equilibrium functions

s [list of list of doubles] relaxation parameters (exemple: s[k][l] is the parameter associated to the lth moment in the kth scheme)

M [sympy matrix] the symbolic matrix of the moments

Mnum [numpy array] the numeric matrix of the moments (m = Mnum F)

invM [sympy matrix] the symbolic inverse matrix

invMnum [numpy array] the numeric inverse matrix (F = invMnum m)

Methods

set_source_terms(self, scheme)	set the source terms functions for the conserved moments.

3.3.1 pylbm.Scheme.set_source_terms

method

Scheme.**set_source_terms**(*self*, *scheme*)

set the source terms functions for the conserved moments.

Parameters

scheme [dictionnary] description of the LBM schemes

Returns

dictionnary the keys are the indices of the conserved moments and the values must be a sympy expression or None

3.4 pylbm.Simulation

class pylbm.**Simulation**(*dico*, *sorder=None*, *dtype='float64'*, *check_inverse=False*)

create a class simulation

Parameters

dico [dictionary]

domain [object of class *Domain*, optional]

scheme [object of class Scheme, optional]

type [optional argument (default value is 'float64')]

Notes

The methods transport, relaxation, equilibrium, f2m, m2f, boundary_condition, and one_time_step are just call of the methods of the class Scheme.

Examples

see demo/examples/

Attributes

dim [int] spatial dimension

type [float64] the type of the values

domain [*Domain*] the domain given in argument

scheme [Scheme] the scheme given in argument

m [numpy array] get the moment i in the interior domain.

F [numpy array] get the distribution function i in the interior domain.

m_halo [numpy array] get the moment i on the whole domain with halo points.

F_halo [numpy array] get the distribution function i on the whole domain with halo points.

Methods

boundary_condition(self)	perform the boundary conditions
equilibrium(self[, m_user])	set the moments to the equilibrium values (the array _m is modified)
f2m(self, **kwargs)	compute the moments from the distribution functions (the array _m is modified)
initialization(self, dico)	initialize all the numy array with the initial conditions set the initial values to the numpy arrays _F and _m
m2f(self[, m_user, f_user])	compute the distribution functions from the moments (the array _F is modified)
one_time_step(self, **kwargs)	compute one time step

Continued on next page

Table 5 – continued from previous page

relaxation(self, **kwargs)	compute the relaxation phase on moments (the array _m is modified)
source_term(self[, fraction_of_time_step])	compute the source term phase on moments (the array _m is modified)
transport(self, **kwargs)	compute the transport phase on distribution functions (the array _F is modified)

3.4.1 pylbm.Simulation.boundary_condition

method

Simulation.**boundary_condition**(*self*)
perform the boundary conditions

Notes

The array _F is modified in the phantom array (outer points) according to the specified boundary conditions.

3.4.2 pylbm.Simulation.equilibrium

method

Simulation.**equilibrium**(*self*, *m_user=None*, ***kwargs*)
set the moments to the equilibrium values (the array _m is modified)

Notes

Another moments vector can be set to equilibrium values: use directly the method of the class Scheme

3.4.3 pylbm.Simulation.f2m

method

Simulation.**f2m**(*self*, ***kwargs*)
compute the moments from the distribution functions (the array _m is modified)

3.4.4 pylbm.Simulation.initialization

method

Simulation.**initialization**(*self*, *dico*)
initialize all the numy array with the initial conditions set the initial values to the numpy arrays _F and _m

Parameters

dico [the dictionary with the *key:value* 'init']

Notes

The initial values are set to _m, the array _F is then initialized with the equilibrium values. If the initial values have to be set to _F, use the optional *key:value* 'inittype' with the value 'distributions' (default value is set to 'moments').

3.4.5 pylbm.Simulation.m2f

method

`Simulation.`**`m2f`**(*self*, *m_user=None*, *f_user=None*, ***kwargs*)
compute the distribution functions from the moments (the array _F is modified)

3.4.6 pylbm.Simulation.one_time_step

method

`Simulation.`**`one_time_step`**(*self*, ***kwargs*)
compute one time step

Notes

Modify the arrays _F and _m in order to go further of dt. This function is equivalent to successively use

- boundary_condition
- transport
- f2m
- relaxation
- m2f

3.4.7 pylbm.Simulation.relaxation

method

`Simulation.`**`relaxation`**(*self*, ***kwargs*)
compute the relaxation phase on moments (the array _m is modified)

3.4.8 pylbm.Simulation.source_term

method

`Simulation.`**`source_term`**(*self*, *fraction_of_time_step=1.0*, ***kwargs*)
compute the source term phase on moments (the array _m is modified)

3.4.9 pylbm.Simulation.transport

method

`Simulation.`**`transport`**(*self*, ***kwargs*)
compute the transport phase on distribution functions (the array _F is modified)

The modules

3.5 the module stencil

Stencil(dico[, need_validation])	Create the stencil of velocities used by the scheme(s).
OneStencil(v, nv)	Create a stencil of a LBM scheme.
Velocity([dim, num, vx, vy, vz])	Create a velocity.

3.5.1 pylbm.stencil.Stencil

class `pylbm.stencil.`**`Stencil`**(*dico*, *need_validation=True*)
Create the stencil of velocities used by the scheme(s).

The numbering of the velocities follows the convention for 1D and 2D.

Parameters

dico [a dictionary that contains the following *key:value*]

- dim : the value of the spatial dimension (1, 2 or 3)
- **schemes** [a list of dictionaries that contain] the key:value velocities

```
[
    {
        'velocities': [...]
    },
    {
        'velocities': [...]
    },
    {
        'velocities': [...]
    },
    ...
]
```

Notes

The velocities for each schemes are defined as a Python list.

Examples

```
>>> s = Stencil({'dim': 1,
...               'schemes':[{'velocities': list(range(9))},],
...              })
>>> s
```

(continues on next page)

(continued from previous page)

```
+---------------------+
| Stencil information |
+---------------------+
    - spatial dimension: 1
    - minimal velocity in each direction: [-4]
    - maximal velocity in each direction: [4]
    - information for each elementary stencil:
        stencil 0
            - number of velocities: 9
            - velocities
                (0: 0)
                (1: 1)
                (2: -1)
                (3: 2)
                (4: -2)
                (5: 3)
                (6: -3)
                (7: 4)
                (8: -4)
>>> s = Stencil({'dim': 2,
...               'schemes':[
...                             {'velocities': list(range(9))},
...                             {'velocities': list(range(49))},
...                         ],
...             })
>>> s
+---------------------+
| Stencil information |
+---------------------+
    - spatial dimension: 2
    - minimal velocity in each direction: [-3 -3]
    - maximal velocity in each direction: [3 3]
    - information for each elementary stencil:
        stencil 0
            - number of velocities: 9
            - velocities
                (0: 0, 0)
                (1: 1, 0)
                (2: 0, 1)
                (3: -1, 0)
                (4: 0, -1)
                (5: 1, 1)
                (6: -1, 1)
                (7: -1, -1)
                (8: 1, -1)
        stencil 1
            - number of velocities: 49
            - velocities
                (0: 0, 0)
                (1: 1, 0)
                (2: 0, 1)
                (3: -1, 0)
                (4: 0, -1)
                (5: 1, 1)
                (6: -1, 1)
                (7: -1, -1)
                (8: 1, -1)
```

(continues on next page)

(continued from previous page)

```
                (9: 2, 0)
                (10: 0, 2)
                (11: -2, 0)
                (12: 0, -2)
                (13: 2, 2)
                (14: -2, 2)
                (15: -2, -2)
                (16: 2, -2)
                (17: 2, 1)
                (18: 1, 2)
                (19: -1, 2)
                (20: -2, 1)
                (21: -2, -1)
                (22: -1, -2)
                (23: 1, -2)
                (24: 2, -1)
                (25: 3, 0)
                (26: 0, 3)
                (27: -3, 0)
                (28: 0, -3)
                (29: 3, 3)
                (30: -3, 3)
                (31: -3, -3)
                (32: 3, -3)
                (33: 3, 1)
                (34: 1, 3)
                (35: -1, 3)
                (36: -3, 1)
                (37: -3, -1)
                (38: -1, -3)
                (39: 1, -3)
                (40: 3, -1)
                (41: 3, 2)
                (42: 2, 3)
                (43: -2, 3)
                (44: -3, 2)
                (45: -3, -2)
                (46: -2, -3)
                (47: 2, -3)
                (48: 3, -2)
```

get the x component of the unique velocities

```
>>> s.uvx
array([ 0,  1,  0, -1,  0,  1, -1, -1,  1,  2,  0, -2,  0,  2, -2, -2,  2,
        2,  1, -1, -2, -2, -1,  1,  2,  3,  0, -3,  0,  3, -3, -3,  3,  3,
        1, -1, -3, -3, -1,  1,  3,  3,  2, -2, -3, -3, -2,  2,  3])
```

get the y component of the velocity for the second stencil

```
>>> s.vy[1]
array([ 0,  0,  1,  0, -1,  1,  1, -1, -1,  0,  2,  0, -2,  2,  2, -2, -2,
        1,  2,  2,  1, -1, -2, -2, -1,  0,  3,  0, -3,  3,  3, -3, -3,  1,
        3,  3,  1, -1, -3, -3, -1,  2,  3,  3,  2, -2, -3, -3, -2])
```

Attributes

dim [int] the spatial dimension (1, 2 or 3).

unique_velocities [NumPy array] array of all velocities involved in the stencils. Each unique velocity appeared only once.

uvx [NumPy array] the x component of the unique velocities.

uvy [NumPy array] the y component of the unique velocities.

uvz [NumPy array] the z component of the unique velocities.

unum [NumPy array] the numbering of the unique velocities.

vmax [int] the maximal velocity in norm for each spatial direction.

vmin [int] the minimal velocity in norm for each spatial direction.

vmax_full [int] the maximal velocity in norm for each spatial direction.

nstencils [int] the number of elementary stencils.

nv [list of integers] the number of velocities for each elementary stencil.

v [list of velocities] list of all the velocities for each elementary stencil.

vx [NumPy array] vx[k] the x component of the velocities for the stencil k.

vy [NumPy array] vy[k] the y component of the velocities for the stencil k.

vz [NumPy array] vz[k] the z component of the velocities for the stencil k.

num [NumPy array] num[k] the numbering of the velocities for the stencil k.

nv_ptr [list of integers] used to obtain the list of the velocities involved in a stencil. For instance, the list for the kth stencil is v[nv_ptr[k]:nv_ptr[k+1]]

unvtot [int] the number of unique velocities involved in the stencils.

Methods

append()	
clear()	
copy()	
count()	
extend()	
extract_dim(dico)	Extract the dimension from the dictionary
get_all_velocities(self[, scheme_id])	get all the velocities for all the stencils in one array
get_symmetric(self[, axis])	get the symmetric velocities.
index()	Raises ValueError if the value is not present.
insert()	L.insert(index, object) – insert object before index
is_symmetric(self)	check if all the velocities have their symmetric.
pop()	Raises IndexError if list is empty or index is out of range.
remove()	Raises ValueError if the value is not present.
reverse()	L.reverse() – reverse *IN PLACE*
sort()	
visualize(self[, viewer_mod, k, ...])	plot the velocities

pylbm.stencil.Stencil.append

method

Stencil.**append**()

pylbm.stencil.Stencil.clear

method

Stencil.**clear**()

pylbm.stencil.Stencil.copy

method

Stencil.**copy**()

pylbm.stencil.Stencil.count

method

Stencil.**count**()

pylbm.stencil.Stencil.extend

method

Stencil.**extend**()

pylbm.stencil.Stencil.extract_dim

method

static Stencil.**extract_dim**(*dico*)
Extract the dimension from the dictionary

pylbm.stencil.Stencil.get_all_velocities

method

Stencil.**get_all_velocities**(*self*, *scheme_id=None*)
get all the velocities for all the stencils in one array

Parameters

scheme_id: int specify for which scheme we want all velocities if None, return the velocities for all the schemes

Returns

ndarray all velocities of a scheme or of all the schemes

pylbm.stencil.Stencil.get_symmetric

method

Stencil.**get_symmetric**(*self*, *axis=None*)
get the symmetric velocities.

pylbm.stencil.Stencil.index

method

Stencil.**index**()
Raises ValueError if the value is not present.

pylbm.stencil.Stencil.insert

method

Stencil.**insert**()
L.insert(index, object) – insert object before index

pylbm.stencil.Stencil.is_symmetric

method

Stencil.**is_symmetric**(*self*)
check if all the velocities have their symmetric.

pylbm.stencil.Stencil.pop

method

Stencil.**pop**()
Raises IndexError if list is empty or index is out of range.

pylbm.stencil.Stencil.remove

method

Stencil.**remove**()
Raises ValueError if the value is not present.

pylbm.stencil.Stencil.reverse

method

Stencil.**reverse**()
L.reverse() – reverse *IN PLACE*

pylbm.stencil.Stencil.sort

method

Stencil.**sort**()

pylbm.stencil.Stencil.visualize

method

Stencil.**visualize**(*self*, *viewer_mod=<module 'pylbm.viewer.matplotlib_viewer' from '/home/docs/checkouts/readthedocs.org/user_builds/pylbm/conda/0.4.1/lib/python3.6/site-packages/pylbm-0.4.1-py3.6.egg/pylbm/viewer/matplotlib_viewer.py'>*, *k=None*, *unique_velocities=False*)

plot the velocities

Parameters

viewer [package used to plot the figure (could be matplotlib, ...)] see viewer for more information

k [list of stencil index to plot] if None plot all stencils

unique_velocities [if True plot the unique velocities]

Returns

fig the figure (fig if matplotlib is used)

3.5.2 pylbm.stencil.OneStencil

class pylbm.stencil.**OneStencil**(*v*, *nv*)

Create a stencil of a LBM scheme.

Parameters

v [list] the list of the velocities of that stencil

nv [int] size of the list

num2index [list of integers] link between the velocity number and its position in the unique velocities array

Attributes

v [list] the list of the velocities of that stencil

nv [int] size of the list v

num the numbering of the velocities.

vx the x component of the velocities.

vy the y component of the velocities.

vz the z component of the velocities.

3.5.3 pylbm.stencil.Velocity

class pylbm.stencil.**Velocity**(*dim=None*, *num=None*, *vx=None*, *vy=None*, *vz=None*)

Create a velocity.

Parameters

dim [int, optional] The dimension of the velocity.

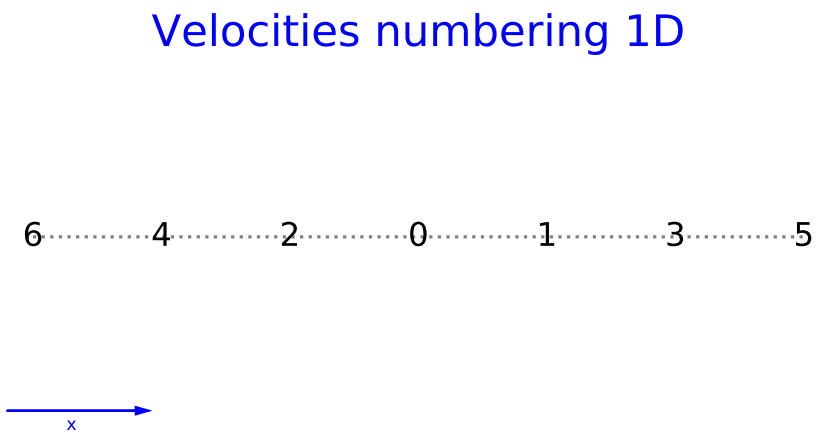
num [int, optional] The number of the velocity in the numbering convention of Lattice-Boltzmann scheme.

vx [int, optional] The x component of the velocity vector.

vy [int, optional] The y component of the velocity vector.

vz [int, optional] The z component of the velocity vector.

Notes



Examples

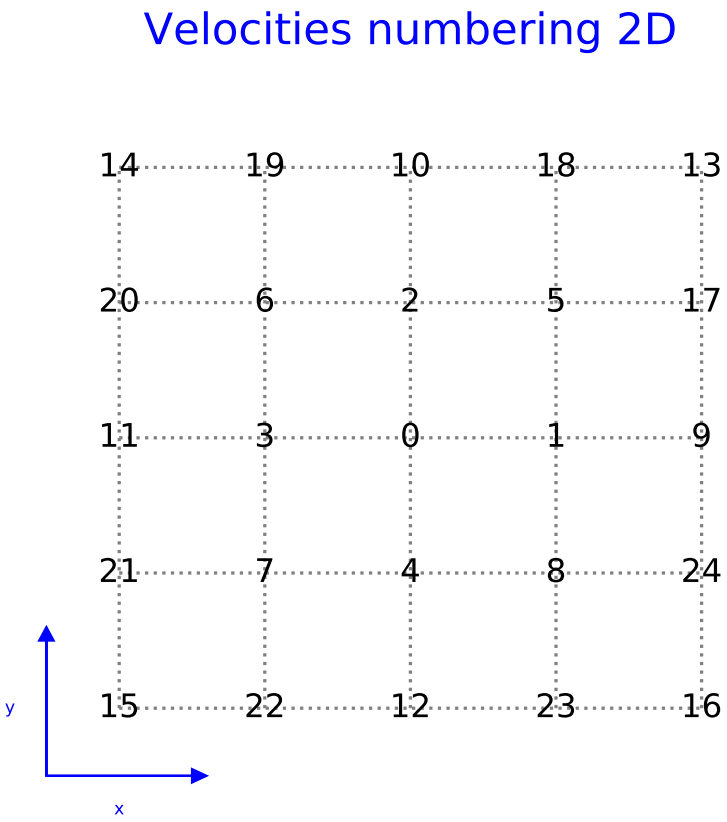
Create a velocity with the dimension and the number

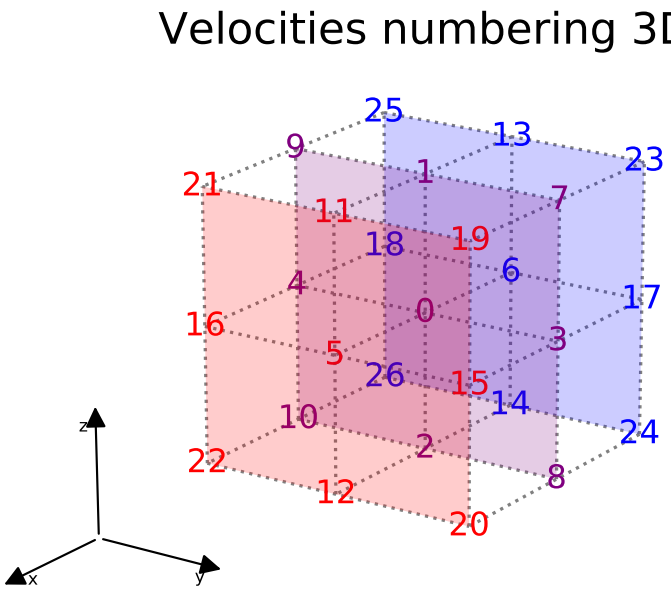
```
>>> v = Velocity(dim=1, num=2)
>>> v
(2: -1)
```

Create a velocity with a direction

```
>>> v = Velocity(vx=1, vy=1)
>>> v
(5: 1, 1)
```

Attributes


Velocities numbering 2D
14
19
10
18
13
20
6
2
5
17
11
3
0
1
9
21
7
4
8
24
15
22
12
23
16
y
x


Velocities numbering 3D
25
13
23
9
1
21
7
11
18
19
6
4
17
0
16
3
5
26
15
14
10
24
2
22
8
12
20
z
x
y

dim [int] The dimension of the velocity.

num The number of the velocity in the numbering convention of Lattice-Boltzmann scheme.

vx [int] The x component of the velocity vector.

vy [int] The y component of the velocity vector.

vz [int] The z component of the velocity vector.

v [list] velocity

Methods

get_symmetric(self[, axis])	return the symmetric velocity.

pylbm.stencil.Velocity.get_symmetric

method

Velocity.**get_symmetric**(*self*, *axis=None*)
return the symmetric velocity.

Parameters

axis [the axis of the symmetry, optional] (None involves the symmetric with the origin, 0 with the x axis, 1 with the y axis, and 2 with the z axis)

Returns

Velocity The symmetric of the velocity

Raises

ValueError if axis is not None and axis < 0 or axis >= dim

3.6 The module elements

New in version 0.2: the geometrical elements are yet implemented in 3D.

The module elements contains all the geometrical shapes that can be used to build the geometry.

The 2D elements are:

Circle(center, radius[, label, isfluid])	Class Circle
Ellipse(center, v1, v2[, label, isfluid])	Class Ellipse
Parallelogram(point, vecta, vectb[, label, ...])	Class Parallelogram
Triangle(point, vecta, vectb[, label, isfluid])	Class Triangle

3.6.1 pylbm.elements.Circle

class pylbm.elements.**Circle**(*center*, *radius*, *label=0*, *isfluid=False*)
Class Circle

Parameters

center [list] the two coordinates of the center

radius [float] the radius

label [list] default [0]

isfluid [boolean]

- True if the circle is added
- False if the circle is deleted

Examples

the circle centered in (0, 0) with radius 1

```
>>> center = [0., 0.]
>>> radius = 1.
>>> Circle(center, radius)
+--------+
| Circle |
+--------+
    - dimension: 2
    - center: [0. 0.]
    - radius: 1.0
    - label: [0]
    - type: solid
```

Attributes

number_of_bounds [int] 1

dimension: int 2

center [ndarray] the coordinates of the center of the circle

radius [double] positive float for the radius of the circle

label [list] the list of the label of the edge

isfluid [boolean] True if the circle is added and False if the circle is deleted

Methods

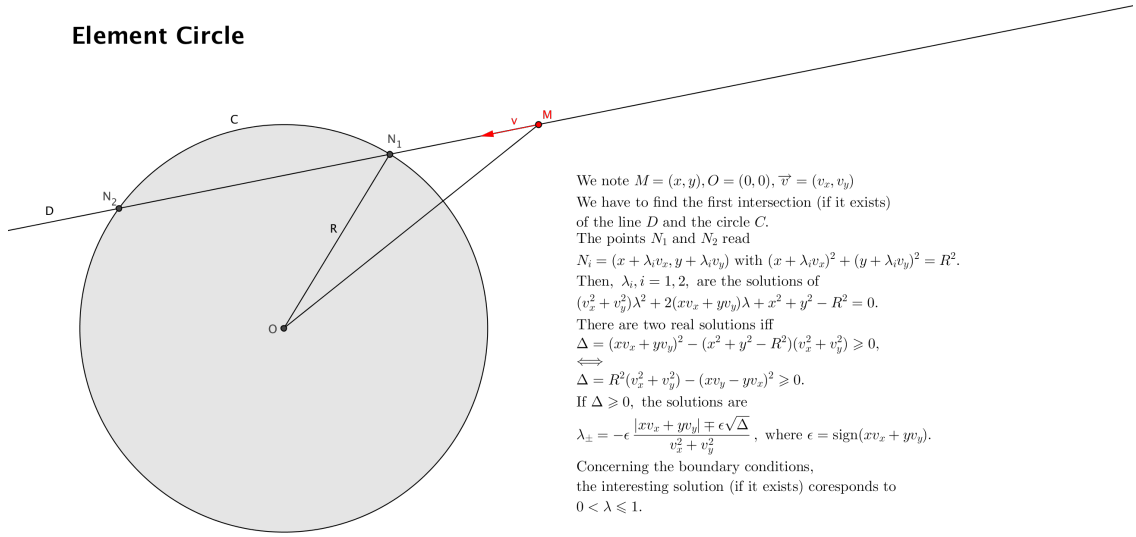
distance(self, grid, v[, dmax])	Compute the distance in the v direction between the circle and the points defined by (x, y).
get_bounds(self)	Get the bounds of the circle.
point_inside(self, grid)	return a boolean array which defines if a point is inside or outside of the circle.
test_label(self)	test if the number of labels is equal to the number of bounds.
visualize(self, viewer, color[, viewlabel, ...])	visualize the element

pylbm.elements.Circle.distance

method

Circle.**distance**(*self*, *grid*, *v*, *dmax=None*)

Compute the distance in the v direction between the circle and the points defined by (x, y).



Parameters

grid [ndarray] coordinates of the points

v [ndarray] direction of interest

dmax [float] distance max

Returns

ndarray array of distances

pylbm.elements.Circle.get_bounds

method

`Circle.`**`get_bounds`**(*self*)
Get the bounds of the circle.

pylbm.elements.Circle.point_inside

method

`Circle.`**`point_inside`**(*self*, *grid*)
return a boolean array which defines if a point is inside or outside of the circle.

Parameters

grid [ndarray] coordinates of the points

Returns

ndarray Array of boolean (True inside the circle, False otherwise)

Notes

the edge of the circle is considered as inside.

pylbm.elements.Circle.test_label

method

Circle.**test_label**(*self*)
test if the number of labels is equal to the number of bounds.

pylbm.elements.Circle.visualize

method

Circle.**visualize**(*self*, *viewer*, *color*, *viewlabel=False*, *scale=array([1., 1.])*, *alpha=1.0*)
visualize the element

Parameters

viewer [Viewer] a viewer (default matplotlib_viewer)

color [color] color of the element

viewlabel [bool] activate the labels mark (default False)

scale [ndarray] scale the distance of the labels (default ones)

alpha [double] transparency of the element (default 1)

3.6.2 pylbm.elements.Ellipse

class pylbm.elements.**Ellipse**(*center*, *v1*, *v2*, *label=0*, *isfluid=False*)
Class Ellipse

Parameters

center [list] the two coordinates of the center

v1 [list] a vector

v2 [list] a second vector (v1 and v2 have to be othogonal)

label [list] one integer (default [0])

isfluid [boolean]

- True if the ellipse is added
- False if the ellipse is deleted

Examples

the ellipse centered in (0, 0) with v1=[2,0], v2=[0,1]

```
>>> center = [0., 0.]
>>> v1 = [2., 0.]
>>> v2 = [0., 1.]
>>> Ellipse(center, v1, v2)
+---------+
| Ellipse |
+---------+
    - dimension: 2
    - center: [0. 0.]
    - v1: [2. 0.]
    - v2: [0. 1.]
    - label: [0]
    - type: solid
```

Attributes

number_of_bounds [int] 1

dim: int 2

center [ndarray] the coordinates of the center of the ellipse

v1 [ndarray] the coordinates of the first vector

v2 [ndarray] the coordinates of the second vector

label [list] the list of the label of the edge

isfluid [boolean] True if the ellipse is added and False if the ellipse is deleted

Methods

distance(self, grid, v[, dmax])	Compute the distance in the v direction between the ellipse and the points defined by (x, y).
get_bounds(self)	Get the bounds of the ellipse.
point_inside(self, grid)	return a boolean array which defines if a point is inside or outside of the ellipse.
test_label(self)	test if the number of labels is equal to the number of bounds.
visualize(self, viewer, color[, viewlabel, ...])	visualize the element

pylbm.elements.Ellipse.distance

method

Ellipse.**distance**(*self*, *grid*, *v*, *dmax=None*)

Compute the distance in the v direction between the ellipse and the points defined by (x, y).

Parameters

grid [ndarray] coordinates of the points

v [ndarray] direction of interest

dmax [float] distance max

Returns

ndarray array of distances

pylbm.elements.Ellipse.get_bounds

method

Ellipse.**get_bounds**(*self*)
Get the bounds of the ellipse.

pylbm.elements.Ellipse.point_inside

method

Ellipse.**point_inside**(*self*, *grid*)
return a boolean array which defines if a point is inside or outside of the ellipse.

Parameters

grid [ndarray] coordinates of the points

Returns

ndarray Array of boolean (True inside the ellipse, False otherwise)

Notes

the edge of the ellipse is considered as inside.

pylbm.elements.Ellipse.test_label

method

Ellipse.**test_label**(*self*)
test if the number of labels is equal to the number of bounds.

pylbm.elements.Ellipse.visualize

method

Ellipse.**visualize**(*self*, *viewer*, *color*, *viewlabel=False*, *scale=array([1., 1.])*, *alpha=1.0*)
visualize the element

Parameters

viewer [Viewer] a viewer (default matplotlib_viewer)

color [color] color of the element

viewlabel [bool] activate the labels mark (default False)

scale [ndarray] scale the distance of the labels (default ones)

alpha [double] transparency of the element (default 1)

3.6.3 pylbm.elements.Parallelogram

class pylbm.elements.**Parallelogram**(*point*, *vecta*, *vectb*, *label=0*, *isfluid=False*)
Class Parallelogram

Parameters

point [list] the coordinates of the first point of the parallelogram

vecta [list] the coordinates of the first vector

vectb [list] the coordinates of the second vector

label [list] four integers (default [0, 0, 0, 0])

isfluid [boolean]

- True if the parallelogram is added
- False if the parallelogram is deleted

Examples

the square [0,1]x[0,1]

```
>>> point = [0., 0.]
>>> vecta = [1., 0.]
>>> vectb = [0., 1.]
>>> Parallelogram(point, vecta, vectb)
+---------------+
| Parallelogram |
+---------------+
    - dimension: 2
    - start point: [0. 0.]
    - v1: [1. 0.]
    - v2: [0. 1.]
    - label: [0, 0, 0, 0]
    - type: solid
```

Attributes

number_of_bounds [int] 4

dim: int 2

point [ndarray] the coordinates of the first point of the parallelogram

v1 [ndarray] the coordinates of the first vector

v2 [ndarray] the coordinates of the second vector

label [list] the list of the label of the edge

isfluid [boolean] True if the parallelogram is added and False if the parallelogram is deleted

Methods

distance(self, grid, v[, dmax])	Compute the distance in the v direction between the parallelogram and the points defined by (x, y).

Continued on next page

Table 12 – continued from previous page

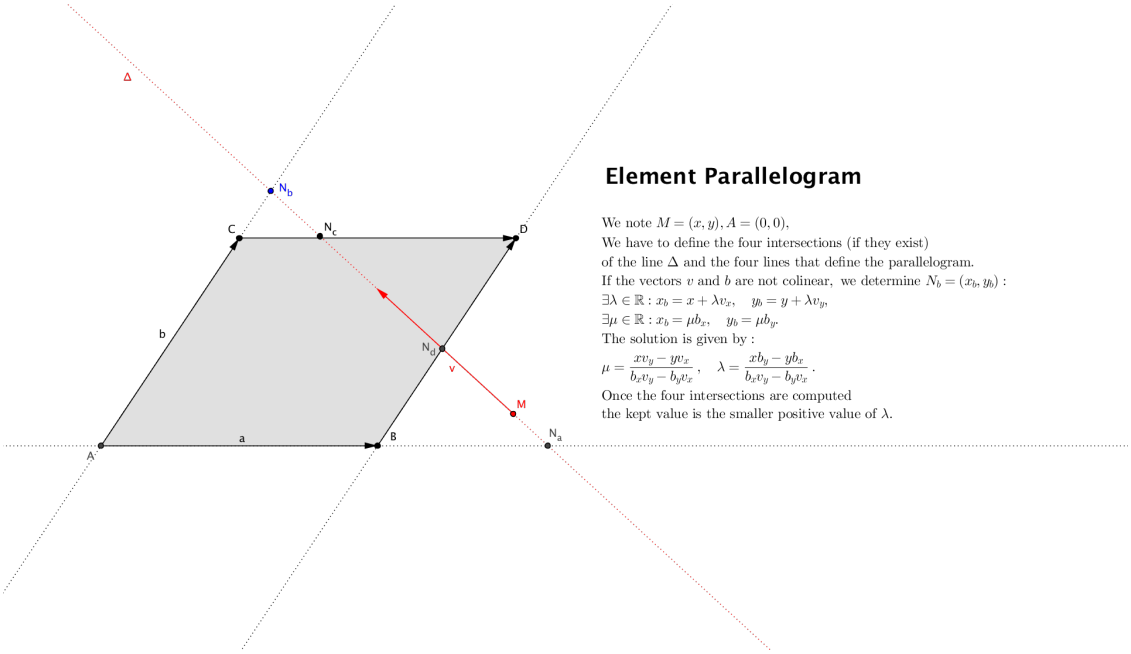
get_bounds(self)	return the bounds of the parallelogram.
point_inside(self, grid)	return a boolean array which defines if a point is inside or outside of the parallelogram.
test_label(self)	test if the number of labels is equal to the number of bounds.
visualize(self, viewer, color[, viewlabel, ...])	visualize the element

pylbm.elements.Parallelogram.distance

method

`Parallelogram.`**`distance`**(*self*, *grid*, *v*, *dmax=None*)

Compute the distance in the v direction between the parallelogram and the points defined by (x, y).



Parameters

grid [ndarray] coordinates of the points

v [ndarray] direction of interest

dmax [float] distance max

Returns

ndarray array of distances

pylbm.elements.Parallelogram.get_bounds

method

`Parallelogram.`**`get_bounds`**(*self*)

return the bounds of the parallelogram.

pylbm.elements.Parallelogram.point_inside

method

Parallelogram.**point_inside**(*self*, *grid*)
return a boolean array which defines if a point is inside or outside of the parallelogram.

Parameters

grid [ndarray] coordinates of the points

Returns

ndarray Array of boolean (True inside the parallelogram, False otherwise)

Notes

the edges of the parallelogram are considered as inside.

pylbm.elements.Parallelogram.test_label

method

Parallelogram.**test_label**(*self*)
test if the number of labels is equal to the number of bounds.

pylbm.elements.Parallelogram.visualize

method

Parallelogram.**visualize**(*self*, *viewer*, *color*, *viewlabel=False*, *scale=array([1., 1.])*, *alpha=1.0*)
visualize the element

Parameters

viewer [Viewer] a viewer (default matplotlib_viewer)

color [color] color of the element

viewlabel [bool] activate the labels mark (default False)

scale [ndarray] scale the distance of the labels (default ones)

alpha [double] transparency of the element (default 1)

3.6.4 pylbm.elements.Triangle

class pylbm.elements.**Triangle**(*point*, *vecta*, *vectb*, *label=0*, *isfluid=False*)
Class Triangle

Parameters

point [list] the coordinates of the first point of the triangle

vecta [list] the coordinates of the first vector

vectb [list] the coordinates of the second vector

label [list] three integers (default [0, 0, 0])

isfluid [boolean]

- True if the triangle is added
- False if the triangle is deleted

Examples

the bottom half square of [0,1]x[0,1]

```
>>> point = [0., 0.]
>>> vecta = [1., 0.]
>>> vectb = [0., 1.]
>>> Triangle(point, vecta, vectb)
+----------+
| Triangle |
+----------+
    - dimension: 2
    - start point: [0. 0.]
    - v1: [1. 0.]
    - v2: [0. 1.]
    - label: [0, 0, 0]
    - type: solid
```

Attributes

point [ndarray] the coordinates of the first point of the triangle

v1 [ndarray] the coordinates of the first vector

v2 [ndarray] the coordinates of the second vector

label [list] the list of the label of the edge

isfluid [boolean] True if the triangle is added and False if the triangle is deleted

number_of_bounds [int] number of edges: 3

dim: int 2

Methods

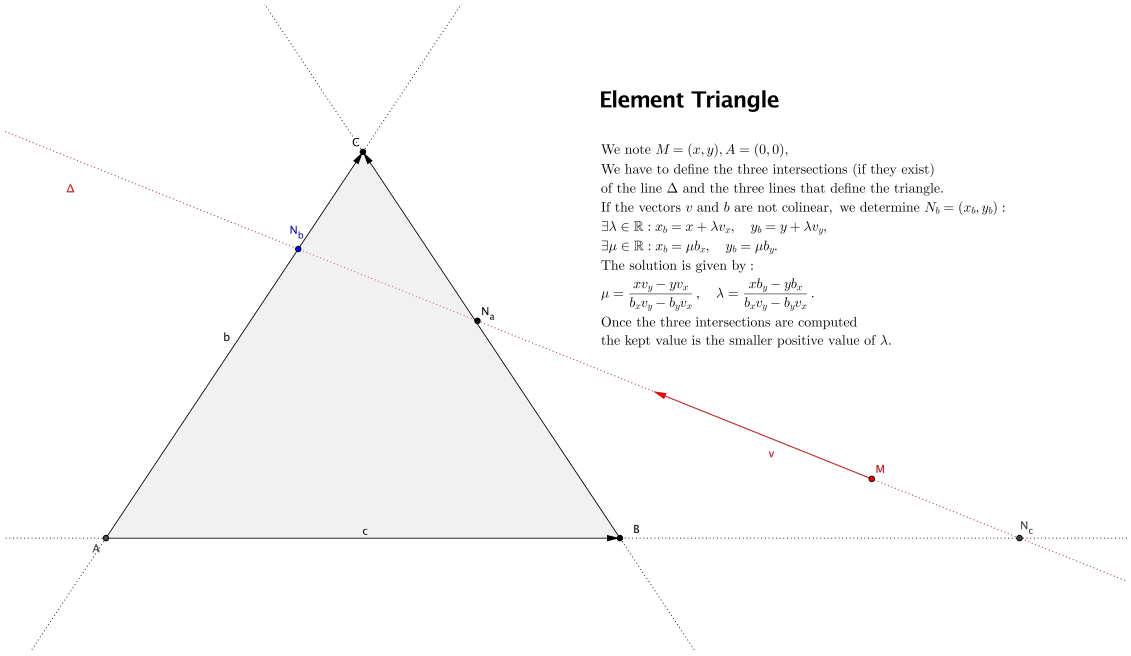
distance(self, grid, v[, dmax])	Compute the distance in the v direction between the triangle and the points defined by (x, y).
get_bounds(self)	return the smallest box where the triangle is.
point_inside(self, grid)	return a boolean array which defines if a point is inside or outside of the triangle.
test_label(self)	test if the number of labels is equal to the number of bounds.
visualize(self, viewer, color[, viewlabel, ...])	visualize the element

pylbm.elements.Triangle.distance

method

Triangle.**distance**(*self*, *grid*, *v*, *dmax=None*)

Compute the distance in the v direction between the triangle and the points defined by (x, y).



Parameters

grid [ndarray] coordinates of the points

v [ndarray] direction of interest

dmax [float] distance max

Returns

ndarray array of distances

pylbm.elements.Triangle.get_bounds

method

`Triangle.`**`get_bounds`**(*self*)
return the smallest box where the triangle is.

pylbm.elements.Triangle.point_inside

method

`Triangle.`**`point_inside`**(*self*, *grid*)
return a boolean array which defines if a point is inside or outside of the triangle.

Parameters

grid [ndarray] coordinates of the points

Returns

ndarray Array of boolean (True inside the triangle, False otherwise)

Notes

the edges of the triangle are considered as inside.

pylbm.elements.Triangle.test_label

method

Triangle.**test_label**(*self*)
test if the number of labels is equal to the number of bounds.

pylbm.elements.Triangle.visualize

method

Triangle.**visualize**(*self*, *viewer*, *color*, *viewlabel=False*, *scale=array([1., 1., 1.])*, *alpha=1.0*)
visualize the element

Parameters

viewer [Viewer] a viewer (default matplotlib_viewer)

color [color] color of the element

viewlabel [bool] activate the labels mark (default False)

scale [ndarray] scale the distance of the labels (default ones)

alpha [double] transparency of the element (default 1)

The 3D elements are:

Sphere(center, radius[, label, isfluid])	Class Sphere
Ellipsoid(center, v1, v2, v3[, label, isfluid])	Class Ellipsoid
Parallelepiped(point, v0, v1, v2[, label, ...])	Class Parallelepiped
CylinderCircle(center, v1, v2, w[, label, ...])	Class CylinderCircle
CylinderEllipse(center, v1, v2, w[, label, ...])	Class CylinderEllipse
CylinderTriangle(center, v1, v2, w[, label, ...])	Class CylinderTriangle

3.6.5 pylbm.elements.Sphere

class pylbm.elements.**Sphere**(*center*, *radius*, *label=0*, *isfluid=False*)
Class Sphere

Parameters

center [list] the three coordinates of the center

radius [real] a positive real number for the radius

label [list] one integer (default [0])

isfluid [boolean]

- True if the sphere is added
- False if the sphere is deleted

Examples

the sphere centered in (0, 0, 0) with radius 1

```
>>> center = [0., 0., 0.]
>>> radius = 1.
>>> Sphere(center, radius)
+--------+
| Sphere |
+--------+
    - dimension: 3
    - center: [0. 0. 0.]
    - radius: 1.0
    - label: [0]
    - type: solid
```

Attributes

number_of_bounds [int] 1

dim: int 3

center [ndarray] the coordinates of the center of the sphere

radius [real] a positive real number for the radius of the sphere

label [list] the list of the label of the edge

isfluid [boolean] True if the sphere is added and False if the sphere is deleted

Methods

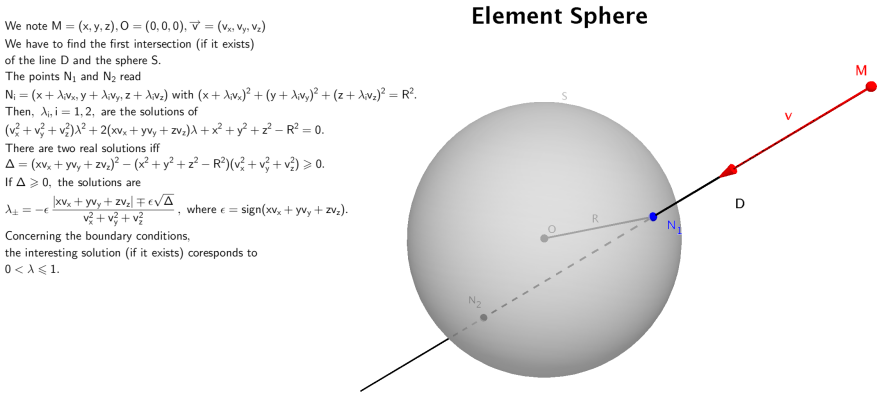
distance(self, grid, v[, dmax])	Compute the distance in the v direction between the sphere and the points defined by (x, y, z).
get_bounds(self)	Get the bounds of the sphere.
point_inside(self, grid)	return a boolean array which defines if a point is inside or outside of the sphere.
test_label(self)	test if the number of labels is equal to the number of bounds.
visualize(self, viewer, color[, viewlabel, ...])	visualize the element

pylbm.elements.Sphere.distance

method

Sphere.**distance**(*self*, *grid*, *v*, *dmax=None*)

Compute the distance in the v direction between the sphere and the points defined by (x, y, z).



Parameters

grid [ndarray] coordinates of the points

v [ndarray] direction of interest

dmax [float] distance max

Returns

ndarray array of distances

pylbm.elements.Sphere.get_bounds

method

`Sphere.`**`get_bounds`**(*self*)

Get the bounds of the sphere.

pylbm.elements.Sphere.point_inside

method

`Sphere.`**`point_inside`**(*self*, *grid*)

return a boolean array which defines if a point is inside or outside of the sphere.

Parameters

grid [ndarray] coordinates of the points

Returns

ndarray Array of boolean (True inside the sphere, False otherwise)

Notes

the edge of the sphere is considered as inside.

pylbm.elements.Sphere.test_label

method

Sphere.**test_label**(*self*)
test if the number of labels is equal to the number of bounds.

pylbm.elements.Sphere.visualize

method

Sphere.**visualize**(*self*, *viewer*, *color*, *viewlabel=False*, *scale=array([1., 1., 1.])*, *alpha=1.0*)
visualize the element

Parameters

viewer [Viewer] a viewer (default matplotlib_viewer)

color [color] color of the element

viewlabel [bool] activate the labels mark (default False)

scale [ndarray] scale the distance of the labels (default ones)

alpha [double] transparency of the element (default 1)

3.6.6 pylbm.elements.Ellipsoid

class pylbm.elements.**Ellipsoid**(*center*, *v1*, *v2*, *v3*, *label=0*, *isfluid=False*)
Class Ellipsoid

Parameters

center [list] the three coordinates of the center

v1 [list] a vector

v2 [list] a vector

v3 [list] a vector (v1, v2, and v3 have to be orthogonal)

label [list] one integer (default [0])

isfluid [boolean]

- True if the ellipsoid is added
- False if the ellipsoid is deleted

Examples

the ellipsoid centered in (0, 0, 0) with v1=[3,0,0], v2=[0,2,0], and v3=[0,0,1]

```
>>> center = [0., 0., 0.]
>>> v1, v2, v3 = [3,0,0], [0,2,0], [0,0,1]
>>> Ellipsoid(center, v1, v2, v3)
+-----------+
| Ellipsoid |
+-----------+
    - dimension: 3
```

(continues on next page)

(continued from previous page)

```
- center: [0. 0. 0.]
- v1: [3 0 0]
- v2: [0 2 0]
- v3: [0 0 1]
- label: [0]
- type: solid
```

Attributes

number_of_bounds [int] 1

dim: int 3

center [ndarray] the coordinates of the center of the sphere

v1 [ndarray] the coordinates of the first vector

v2 [ndarray] the coordinates of the second vector

v3 [ndarray] the coordinates of the third vector

label [list] the list of the label of the edge

isfluid [boolean] True if the ellipsoid is added and False if the ellipsoid is deleted

Methods

distance(self, grid, v[, dmax])	Compute the distance in the v direction between the ellipsoid and the points defined by (x, y, z).
get_bounds(self)	Get the bounds of the ellipsoid.
point_inside(self, grid)	return a boolean array which defines if a point is inside or outside of the ellipsoid.
test_label(self)	test if the number of labels is equal to the number of bounds.
visualize(self, viewer, color[, viewlabel, ...])	visualize the element

pylbm.elements.Ellipsoid.distance

method

Ellipsoid.**distance**(*self*, *grid*, *v*, *dmax=None*)

Compute the distance in the v direction between the ellipsoid and the points defined by (x, y, z).

Parameters

grid [ndarray] coordinates of the points

v [ndarray] direction of interest

dmax [float] distance max

Returns

ndarray array of distances

pylbm.elements.Ellipsoid.get_bounds

method

Ellipsoid.**get_bounds**(*self*)
Get the bounds of the ellipsoid.

pylbm.elements.Ellipsoid.point_inside

method

Ellipsoid.**point_inside**(*self*, *grid*)
return a boolean array which defines if a point is inside or outside of the ellipsoid.

Parameters

grid [ndarray] coordinates of the points

Returns

ndarray Array of boolean (True inside the ellipsoid, False otherwise)

Notes

the edge of the ellipsoid is considered as inside.

pylbm.elements.Ellipsoid.test_label

method

Ellipsoid.**test_label**(*self*)
test if the number of labels is equal to the number of bounds.

pylbm.elements.Ellipsoid.visualize

method

Ellipsoid.**visualize**(*self*, *viewer*, *color*, *viewlabel=False*, *scale=array([1., 1., 1.])*, *alpha=1.0*)
visualize the element

Parameters

viewer [Viewer] a viewer (default matplotlib_viewer)

color [color] color of the element

viewlabel [bool] activate the labels mark (default False)

scale [ndarray] scale the distance of the labels (default ones)

alpha [double] transparency of the element (default 1)

3.6.7 pylbm.elements.Parallelepiped

class pylbm.elements.**Parallelepiped**(*point*, *v0*, *v1*, *v2*, *label=0*, *isfluid=False*)
Class Parallelepiped

Parameters

point [list] the three coordinates of the first point

v0 [list] the three coordinates of the first vector that defines the edge

v1 [list] the three coordinates of the second vector that defines the edge

v2 [list] the three coordinates of the third vector that defines the edge

label [list] three integers (default [0,0,0] for the bottom, the top and the side)

isfluid [boolean]

- True if the cylinder is added
- False if the cylinder is deleted

Examples

the vertical canonical cube centered in (0, 0, 0)

```
>>> center = [0., 0., 0.5]
>>> v0, v1, v2 = [1., 0., 0.], [0., 1., 0.], [0., 0., 1.]
>>> Parallelepiped(center, v0, v1, v2)
+----------------+
| Parallelepiped |
+----------------+
    - dimension: 3
    - center: [0. 0. 1.]
    - v1: [1. 0. 0.]
    - v2: [0. 1. 0.]
    - w: [0.  0.  0.5]
    - label: [0, 0, 0, 0, 0, 0]
    - type: solid
```

Attributes

number_of_bounds [int] 6

dim: int 3

point [ndarray] the coordinates of the first point of the parallelepiped

v0 [list] the three coordinates of the first vector

v1 [list] the three coordinates of the second vector

v2 [list] the three coordinates of the third vector

label [list] the list of the label of the edge

isfluid [boolean] True if the parallelepiped is added and False if the parallelepiped is deleted

Methods

change_of_variables(self)	matrix for the change of variables used to write the coordinates in the basis of the cylinder
distance(self, grid, v[, dmax])	Compute the distance in the v direction between the cylinder and the points defined by (x, y, z).
get_bounds(self)	Get the bounds of the cylinder.
point_inside(self, grid)	return a boolean array which defines if a point is inside or outside of the cylinder.

Continued on next page

Table 17 – continued from previous page

test_label(self)	test if the number of labels is equal to the number of bounds.
visualize(self, viewer, color[, viewlabel, ...])	visualize the element

pylbm.elements.Parallelepiped.change_of_variables

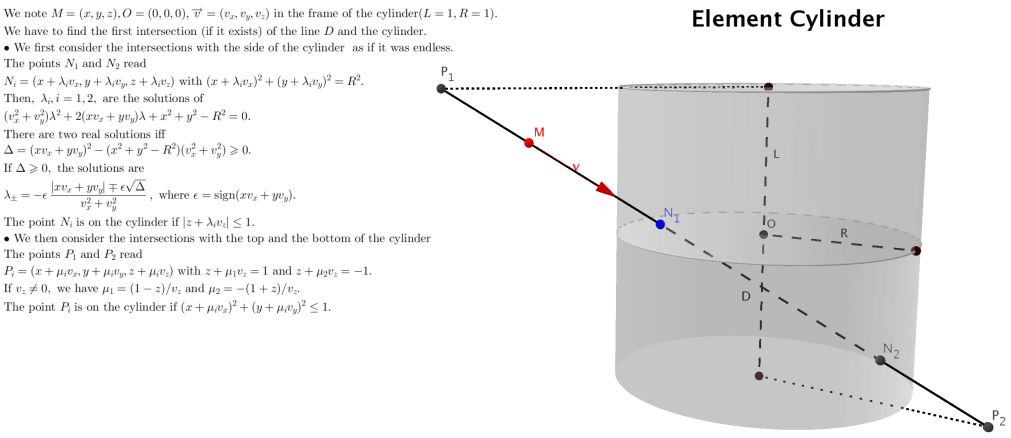
method

Parallelepiped.**change_of_variables**(*self*)
matrix for the change of variables used to write the coordinates in the basis of the cylinder

pylbm.elements.Parallelepiped.distance

method

Parallelepiped.**distance**(*self*, *grid*, *v*, *dmax=None*)
Compute the distance in the v direction between the cylinder and the points defined by (x, y, z).



Parameters

grid [ndarray] coordinates of the points

v [ndarray] direction of interest

dmax [float] distance max

Returns

ndarray array of distances

pylbm.elements.Parallelepiped.get_bounds

method

Parallelepiped.**get_bounds**(*self*)
Get the bounds of the cylinder.

Returns

ndarray minimal box where the cylinder is included

pylbm.elements.Parallelepiped.point_inside

method

`Parallelepiped.`**`point_inside`**(*self*, *grid*)
return a boolean array which defines if a point is inside or outside of the cylinder.

Parameters

grid [ndarray] coordinates of the points

Returns

ndarray Array of boolean (True inside the cylinder, False otherwise)

Notes

the edge of the cylinder is considered as inside.

pylbm.elements.Parallelepiped.test_label

method

`Parallelepiped.`**`test_label`**(*self*)
test if the number of labels is equal to the number of bounds.

pylbm.elements.Parallelepiped.visualize

method

`Parallelepiped.`**`visualize`**(*self*, *viewer*, *color*, *viewlabel=False*, *scale=array([1., 1., 1.])*, *alpha=1.0*)
visualize the element

Parameters

viewer [Viewer] a viewer (default matplotlib_viewer)

color [color] color of the element

viewlabel [bool] activate the labels mark (default False)

scale [ndarray] scale the distance of the labels (default ones)

alpha [double] transparency of the element (default 1)

3.6.8 pylbm.elements.CylinderCircle

class `pylbm.elements.`**`CylinderCircle`**(*center*, *v1*, *v2*, *w*, *label=0*, *isfluid=False*)
Class CylinderCircle

Parameters

center [list] the three coordinates of the center

v1 [list] the first vector that defines the circular section

v2 [list] the second vector that defines the circular section

w [list] the vector that defines the direction of the side

label [list] three integers (default [0,0,0] for the bottom, the top and the side)

isfluid [boolean]

- True if the cylinder is added
- False if the cylinder is deleted

Examples

the vertical canonical cylinder centered in (0, 0, 1/2) with radius 1

```
>>> center = [0., 0., 0.5]
>>> v1, v2 = [1., 0., 0.], [0., 1., 0.]
>>> w = [0., 0., 1.]
>>> CylinderCircle(center, v1, v2, w)
+----------------+
| CylinderCircle |
+----------------+
    - dimension: 3
    - center: [0.  0.  0.5]
    - v1: [1. 0. 0.]
    - v2: [0. 1. 0.]
    - w: [0. 0. 1.]
    - label: [0, 0, 0]
    - type: solid
```

Attributes

number_of_bounds [int] 3

dim: int 3

center [ndarray] the coordinates of the center of the cylinder

v1 [list] the three coordinates of the first vector defining the base section

v2 [list] the three coordinates of the second vector defining the base section

w [list] the three coordinates of the vector defining the direction of the side

label [list] the list of the label of the edge

isfluid [boolean] True if the cylinder is added and False if the cylinder is deleted

Methods

change_of_variables(self)	matrix for the change of variables used to write the coordinates in the basis of the cylinder
distance(self, grid, v[, dmax])	Compute the distance in the v direction between the cylinder and the points defined by (x, y, z).
get_bounds(self)	Get the bounds of the cylinder.

Continued on next page

Table 18 – continued from previous page

point_inside(self, grid)	return a boolean array which defines if a point is inside or outside of the cylinder.
test_label(self)	test if the number of labels is equal to the number of bounds.
visualize(self, viewer, color[, viewlabel, ...])	visualize the element

pylbm.elements.CylinderCircle.change_of_variables

method

CylinderCircle.**change_of_variables**(*self*)

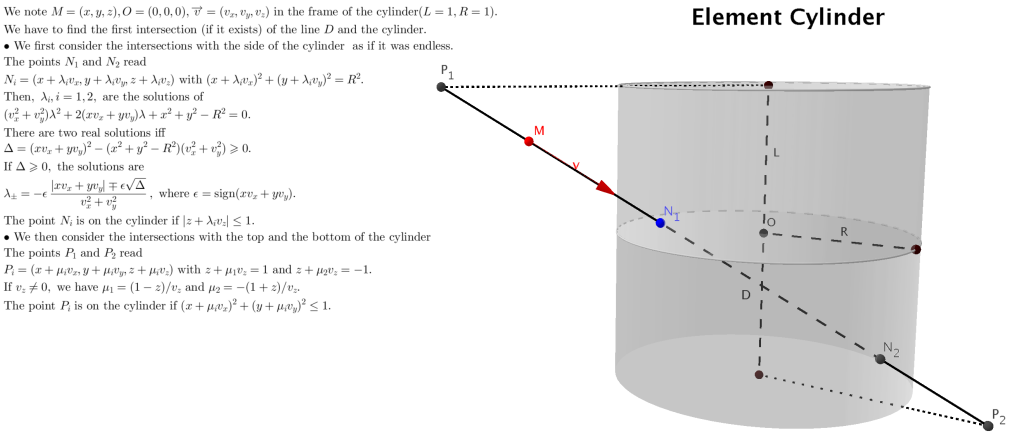
matrix for the change of variables used to write the coordinates in the basis of the cylinder

pylbm.elements.CylinderCircle.distance

method

CylinderCircle.**distance**(*self*, *grid*, *v*, *dmax=None*)

Compute the distance in the v direction between the cylinder and the points defined by (x, y, z).



Parameters

grid [ndarray] coordinates of the points

v [ndarray] direction of interest

dmax [float] distance max

Returns

ndarray array of distances

pylbm.elements.CylinderCircle.get_bounds

method

CylinderCircle.**get_bounds**(*self*)
Get the bounds of the cylinder.

Returns

ndarray minimal box where the cylinder is included

pylbm.elements.CylinderCircle.point_inside

method

CylinderCircle.**point_inside**(*self*, *grid*)
return a boolean array which defines if a point is inside or outside of the cylinder.

Parameters

grid [ndarray] coordinates of the points

Returns

ndarray Array of boolean (True inside the cylinder, False otherwise)

Notes

the edge of the cylinder is considered as inside.

pylbm.elements.CylinderCircle.test_label

method

CylinderCircle.**test_label**(*self*)
test if the number of labels is equal to the number of bounds.

pylbm.elements.CylinderCircle.visualize

method

CylinderCircle.**visualize**(*self*, *viewer*, *color*, *viewlabel=False*, *scale=array([1., 1., 1.])*, *alpha=1.0*)
visualize the element

Parameters

viewer [Viewer] a viewer (default matplotlib_viewer)

color [color] color of the element

viewlabel [bool] activate the labels mark (default False)

scale [ndarray] scale the distance of the labels (default ones)

alpha [double] transparency of the element (default 1)

3.6.9 pylbm.elements.CylinderEllipse

class pylbm.elements.**CylinderEllipse**(*center*, *v1*, *v2*, *w*, *label=0*, *isfluid=False*)
Class CylinderEllipse

Parameters

center [list] the three coordinates of the center

v1 [list] the first vector that defines the circular section

v2 [list] the second vector that defines the circular section

w [list] the vector that defines the direction of the side

label [list] three integers (default [0,0,0] for the bottom, the top and the side)

isfluid [boolean]

- True if the cylinder is added
- False if the cylinder is deleted

Warning: The vectors v1 and v2 have to be orthogonal.

Examples

the vertical canonical cylinder centered in (0, 0, 1/2) with radius 1

```
>>> center = [0., 0., 0.5]
>>> v1, v2 = [1., 0., 0.], [0., 1., 0.]
>>> w = [0., 0., 1.]
>>> CylinderEllipse(center, v1, v2, w)
+-----------------+
| CylinderEllipse |
+-----------------+
    - dimension: 3
    - center: [0.  0.  0.5]
    - v1: [1. 0. 0.]
    - v2: [0. 1. 0.]
    - w: [0. 0. 1.]
    - label: [0, 0, 0]
    - type: solid
```

Attributes

number_of_bounds [int] 3

dim: int 3

center [ndarray] the coordinates of the center of the cylinder

v1 [list] the three coordinates of the first vector defining the base section

v2 [list] the three coordinates of the second vector defining the base section

w [list] the three coordinates of the vector defining the direction of the side

label [list] the list of the label of the edge

isfluid [boolean] True if the cylinder is added and False if the cylinder is deleted

Methods

change_of_variables(self)	matrix for the change of variables used to write the coordinates in the basis of the cylinder
distance(self, grid, v[, dmax])	Compute the distance in the v direction between the cylinder and the points defined by (x, y, z).
get_bounds(self)	Get the bounds of the cylinder.
point_inside(self, grid)	return a boolean array which defines if a point is inside or outside of the cylinder.
test_label(self)	test if the number of labels is equal to the number of bounds.
visualize(self, viewer, color[, viewlabel, ...])	visualize the element

pylbm.elements.CylinderEllipse.change_of_variables

method

CylinderEllipse.**change_of_variables**(*self*)

matrix for the change of variables used to write the coordinates in the basis of the cylinder

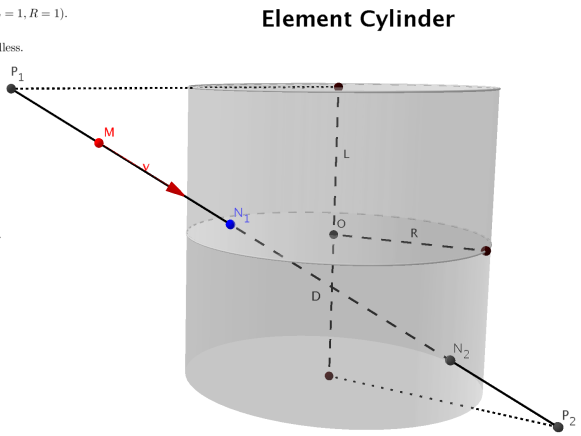
pylbm.elements.CylinderEllipse.distance

method

CylinderEllipse.**distance**(*self*, *grid*, *v*, *dmax=None*)

Compute the distance in the v direction between the cylinder and the points defined by (x, y, z).

We note $M = (x, y, z), O = (0, 0, 0), \vec{v} = (v_x, v_y, v_z)$ in the frame of the cylinder$(L = 1, R = 1)$.
We have to find the first intersection (if it exists) of the line D and the cylinder.
• We first consider the intersections with the side of the cylinder as if it was endless.
The points N_1 and N_2 read
$N_i = (x + \lambda_i v_x, y + \lambda_i v_y, z + \lambda_i v_z)$ with $(x + \lambda_i v_x)^2 + (y + \lambda_i v_y)^2 = R^2$.
Then, $\lambda_i, i = 1, 2,$ are the solutions of
$(v_x^2 + v_y^2)\lambda^2 + 2(xv_x + yv_y)\lambda + x^2 + y^2 - R^2 = 0$.
There are two real solutions iff
$\Delta = (xv_x + yv_y)^2 - (x^2 + y^2 - R^2)(v_x^2 + v_y^2) \geqslant 0$.
If $\Delta \geqslant 0,$ the solutions are
$\lambda_\pm = -\epsilon \dfrac{|xv_x + yv_y| \mp \epsilon\sqrt{\Delta}}{v_x^2 + v_y^2}$, where $\epsilon = \text{sign}(xv_x + yv_y)$.
The point N_i is on the cylinder if $|z + \lambda_i v_z| \leq 1$.
• We then consider the intersections with the top and the bottom of the cylinder
The points P_1 and P_2 read
$P_i = (x + \mu_i v_x, y + \mu_i v_y, z + \mu_i v_z)$ with $z + \mu_1 v_z = 1$ and $z + \mu_2 v_z = -1$.
If $v_z \neq 0,$ we have $\mu_1 = (1 - z)/v_z$ and $\mu_2 = -(1 + z)/v_z$.
The point P_i is on the cylinder if $(x + \mu_i v_x)^2 + (y + \mu_i v_y)^2 \leq 1$.



Parameters

grid [ndarray] coordinates of the points

v [ndarray] direction of interest

dmax [float] distance max

Returns

ndarray array of distances

pylbm.elements.CylinderEllipse.get_bounds

method

CylinderEllipse.**get_bounds**(*self*)
Get the bounds of the cylinder.

Returns

ndarray minimal box where the cylinder is included

pylbm.elements.CylinderEllipse.point_inside

method

CylinderEllipse.**point_inside**(*self*, *grid*)
return a boolean array which defines if a point is inside or outside of the cylinder.

Parameters

grid [ndarray] coordinates of the points

Returns

ndarray Array of boolean (True inside the cylinder, False otherwise)

Notes

the edge of the cylinder is considered as inside.

pylbm.elements.CylinderEllipse.test_label

method

CylinderEllipse.**test_label**(*self*)
test if the number of labels is equal to the number of bounds.

pylbm.elements.CylinderEllipse.visualize

method

CylinderEllipse.**visualize**(*self*, *viewer*, *color*, *viewlabel=False*, *scale=array([1., 1., 1.])*, *alpha=1.0*)
visualize the element

Parameters

viewer [Viewer] a viewer (default matplotlib_viewer)

color [color] color of the element

viewlabel [bool] activate the labels mark (default False)

scale [ndarray] scale the distance of the labels (default ones)

alpha [double] transparency of the element (default 1)

3.6.10 pylbm.elements.CylinderTriangle

class pylbm.elements.**CylinderTriangle**(*center*, *v1*, *v2*, *w*, *label=0*, *isfluid=False*)

Class CylinderTriangle

Parameters

center [list] the three coordinates of the center

v1 [list] the first vector that defines the triangular section

v2 [list] the second vector that defines the triangular section

w [list] the vector that defines the direction of the side

label [list] three integers (default [0,0,0] for the bottom, the top and the side)

isfluid [boolean]

- True if the cylinder is added
- False if the cylinder is deleted

Examples

the vertical canonical cylinder centered in (0, 0, 1/2)

```
>>> center = [0., 0., 0.5]
>>> v1, v2 = [1., 0., 0.], [0., 1., 0.]
>>> w = [0., 0., 1.]
>>> CylinderTriangle(center, v1, v2, w)
+------------------+
| CylinderTriangle |
+------------------+
    - dimension: 3
    - center: [0.  0.  0.5]
    - v1: [1. 0. 0.]
    - v2: [0. 1. 0.]
    - w: [0. 0. 1.]
    - label: [0, 0, 0, 0, 0]
    - type: solid
```

Attributes

number_of_bounds [int] 5

dim: int 3

center [numpy array] the coordinates of the center of the cylinder

v1 [list of doubles] the three coordinates of the first vector defining the base section

v2 [list of doubles] the three coordinates of the second vector defining the base section

w [list of doubles] the three coordinates of the vector defining the direction of the side

label [list of integers] the list of the label of the edge

isfluid [boolean] True if the cylinder is added and False if the cylinder is deleted

Methods

change_of_variables(self)	matrix for the change of variables used to write the coordinates in the basis of the cylinder
distance(self, grid, v[, dmax])	Compute the distance in the v direction between the cylinder and the points defined by (x, y, z).
get_bounds(self)	Get the bounds of the cylinder.
point_inside(self, grid)	return a boolean array which defines if a point is inside or outside of the cylinder.
test_label(self)	test if the number of labels is equal to the number of bounds.
visualize(self, viewer, color[, viewlabel, ...])	visualize the element

pylbm.elements.CylinderTriangle.change_of_variables

method

CylinderTriangle.**change_of_variables**(*self*)

matrix for the change of variables used to write the coordinates in the basis of the cylinder

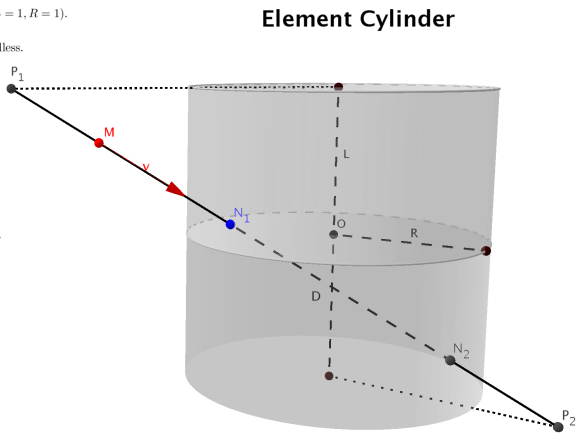
pylbm.elements.CylinderTriangle.distance

method

CylinderTriangle.**distance**(*self*, *grid*, *v*, *dmax=None*)

Compute the distance in the v direction between the cylinder and the points defined by (x, y, z).

We note $M = (x, y, z), O = (0, 0, 0), \vec{v} = (v_x, v_y, v_z)$ in the frame of the cylinder$(L = 1, R = 1)$.
We have to find the first intersection (if it exists) of the line D and the cylinder.
• We first consider the intersections with the side of the cylinder as if it was endless.
The points N_1 and N_2 read
$N_i = (x + \lambda_i v_x, y + \lambda_i v_y, z + \lambda_i v_z)$ with $(x + \lambda_i v_x)^2 + (y + \lambda_i v_y)^2 = R^2$.
Then, $\lambda_i, i = 1, 2,$ are the solutions of
$(v_x^2 + v_y^2)\lambda^2 + 2(xv_x + yv_y)\lambda + x^2 + y^2 - R^2 = 0$.
There are two real solutions iff
$\Delta = (xv_x + yv_y)^2 - (x^2 + y^2 - R^2)(v_x^2 + v_y^2) \geqslant 0$.
If $\Delta \geqslant 0,$ the solutions are
$\lambda_\pm = -\epsilon \dfrac{|xv_x + yv_y| \mp \epsilon\sqrt{\Delta}}{v_x^2 + v_y^2}$, where $\epsilon = \text{sign}(xv_x + yv_y)$.
The point N_i is on the cylinder if $|z + \lambda_i v_z| \leq 1$.
• We then consider the intersections with the top and the bottom of the cylinder
The points P_1 and P_2 read
$P_i = (x + \mu_i v_x, y + \mu_i v_y, z + \mu_i v_z)$ with $z + \mu_1 v_z = 1$ and $z + \mu_2 v_z = -1$.
If $v_z \neq 0,$ we have $\mu_1 = (1 - z)/v_z$ and $\mu_2 = -(1 + z)/v_z$.
The point P_i is on the cylinder if $(x + \mu_i v_x)^2 + (y + \mu_i v_y)^2 \leq 1$.



Parameters

grid [ndarray] coordinates of the points

v [ndarray] direction of interest

dmax [float] distance max

Returns

ndarray array of distances

pylbm.elements.CylinderTriangle.get_bounds

method

`CylinderTriangle.`**`get_bounds`**(*self*)
Get the bounds of the cylinder.

Returns

ndarray minimal box where the cylinder is included

pylbm.elements.CylinderTriangle.point_inside

method

`CylinderTriangle.`**`point_inside`**(*self*, *grid*)
return a boolean array which defines if a point is inside or outside of the cylinder.

Parameters

grid [ndarray] coordinates of the points

Returns

ndarray Array of boolean (True inside the cylinder, False otherwise)

Notes

the edge of the cylinder is considered as inside.

pylbm.elements.CylinderTriangle.test_label

method

`CylinderTriangle.`**`test_label`**(*self*)
test if the number of labels is equal to the number of bounds.

pylbm.elements.CylinderTriangle.visualize

method

`CylinderTriangle.`**`visualize`**(*self*, *viewer*, *color*, *viewlabel=False*, *scale=array([1., 1., 1.])*, *alpha=1.0*)
visualize the element

Parameters

viewer [Viewer] a viewer (default matplotlib_viewer)

color [color] color of the element

viewlabel [bool] activate the labels mark (default False)

scale [ndarray] scale the distance of the labels (default ones)

alpha [double] transparency of the element (default 1)

3.7 the module geometry

Geometry(dico[, need_validation])	Create a geometry that defines the fluid part and the solid part.

3.7.1 pylbm.geometry.Geometry

class pylbm.geometry.**Geometry**(*dico*, *need_validation=True*)

Create a geometry that defines the fluid part and the solid part.

Parameters

dico [dict]

dictionary that contains the following *key:value*

- box : a dictionary for the definition of the computed box
- elements : a list of elements (optional)

need_validation [bool] boolean to specify if the dictionary has to be validated (optional)

Notes

The dictionary that defines the box should contains the following *key:value*

- x : a list of the bounds in the first direction
- y : a list of the bounds in the second direction (optional)
- z : a list of the bounds in the third direction (optional)
- **label** [an integer or a list of integers] (length twice the number of dimensions) used to label each edge (optional)

Examples

see demo/examples/geometry/

Attributes

dim [int] number of spatial dimensions (1, 2, or 3)

bounds [ndarray] the bounds of the box in each spatial direction

box_label [list]

a list of the four labels for the left, right, bottom, top, front, and back edges

list_elem [list] a list that contains each element added or deleted in the box

Methods

add_elem(self, elem)	add a solid or a fluid part in the domain
list_of_elements_labels(self)	Get the list of all the labels used in the geometry.
list_of_labels(self)	Get the list of all the labels used in the geometry.
visualize(self[, viewer_app, figsize, ...])	plot a view of the geometry

pylbm.geometry.Geometry.add_elem

method

Geometry.**add_elem**(*self*, *elem*)

add a solid or a fluid part in the domain

Parameters

elem [Element] a geometric element to add (or to del)

pylbm.geometry.Geometry.list_of_elements_labels

method

Geometry.**list_of_elements_labels**(*self*)

Get the list of all the labels used in the geometry.

pylbm.geometry.Geometry.list_of_labels

method

Geometry.**list_of_labels**(*self*)

Get the list of all the labels used in the geometry.

pylbm.geometry.Geometry.visualize

method

Geometry.**visualize**(*self*, *viewer_app=<module 'pylbm.viewer.matplotlib_viewer' from '/home/docs/checkouts/readthedocs.org/user_builds/pylbm/conda/0.4.1/lib/python3.6/site-packages/pylbm-0.4.1-py3.6.egg/pylbm/viewer/matplotlib_viewer.py'>*, *figsize=(6, 4)*, *viewlabel=False*, *fluid_color='navy'*, *viewgrid=False*, *alpha=1.0*)

plot a view of the geometry

Parameters

viewer_app [Viewer] a viewer (default matplotlib_viewer)

viewlabel [boolean] activate the labels mark (default False)

fluid_color [color] color for the fluid part (default blue)

figsize [tuple] the size of the figure (default (6, 4))

viewgrid [bool] view the grid (default False)

alpha [double] transparency between 0 and 1 (default 1)

Returns

object views

3.8 the module domain

Domain(dico[, need_validation])	Create a domain that defines the fluid part and the solid part and computes the distances between these two states.

3.8.1 pylbm.domain.Domain

class pylbm.domain.**Domain**(*dico*, *need_validation=True*)

Create a domain that defines the fluid part and the solid part and computes the distances between these two states.

Parameters

dico [dictionary] that contains the following *key:value*

- box : a dictionary that defines the computational box
- **elements** [the list of the elements] (available elements are given in the module `elements`)
- space_step : the spatial step
- schemes : a list of dictionaries,

each of them defining a elementary `Scheme` we only need the velocities to define a domain

need_validation [bool] boolean to specify if the dictionary has to be validated (optional)

Warning: the sizes of the box must be a multiple of the space step dx

Notes

The dictionary that defines the box should contains the following *key:value*

- x : a list of the bounds in the first direction
- y : a list of the bounds in the second direction (optional)
- z : a list of the bounds in the third direction (optional)
- label : an integer or a list of integers (length twice the number of dimensions) used to label each edge (optional)

See *Geometry* for more details.

In 1D, distance[q, i] is the distance between the point x[i] and the border in the direction of the qth velocity.

In 2D, distance[q, j, i] is the distance between the point (x[i], y[j]) and the border in the direction of qth velocity

In 3D, distance[q, k, j, i] is the distance between the point (x[i], y[j], z[k]) and the border in the direction of qth velocity

In 1D, flag[q, i] is the flag of the border reached by the point x[i] in the direction of the qth velocity

In 2D, flag[q, j, i] is the flag of the border reached by the point (x[i], y[j]) in the direction of qth velocity

In 2D, flag[q, k, j, i] is the flag of the border reached by the point (x[i], y[j], z[k]) in the direction of qth velocity

Examples

```
>>> dico = {'box': {'x': [0, 1], 'label': 0},
...         'space_step': 0.1,
...         'schemes': [{'velocities': list(range(3))}],
...        }
>>> dom = Domain(dico)
>>> dom
+--------------------+
| Domain information |
+--------------------+
    - spatial dimension: 1
    - space step: 0.1
    - with halo:
        bounds of the box: [-0.05] x [1.05]
        number of points: [12]
    - without halo:
        bounds of the box: [0.05] x [0.95]
        number of points: [10]
<BLANKLINE>
    +----------------------+
    | Geometry information |
    +----------------------+
        - spatial dimension: 1
        - bounds of the box: [0. 1.]
```

```
>>> dico = {'box': {'x': [0, 1], 'y': [0, 1], 'label': [0, 0, 1, 1]},
...         'space_step': 0.1,
...         'schemes': [{'velocities': list(range(9))},
...                     {'velocities': list(range(5))}
...                    ],
...        }
>>> dom = Domain(dico)
>>> dom
+--------------------+
| Domain information |
+--------------------+
    - spatial dimension: 2
    - space step: 0.1
    - with halo:
        bounds of the box: [-0.05 -0.05] x [1.05 1.05]
        number of points: [12, 12]
    - without halo:
        bounds of the box: [0.05 0.05] x [0.95 0.95]
        number of points: [10, 10]
<BLANKLINE>
    +----------------------+
    | Geometry information |
    +----------------------+
        - spatial dimension: 2
        - bounds of the box: [0. 1.] x [0. 1.]
```

see demo/examples/domain/

Attributes

dim [int] number of spatial dimensions (example: 1, 2, or 3)

globalbounds [ndarray] the bounds of the box in each spatial direction

bounds [ndarray] the local bounds of the process in each spatial direction

dx [double] space step (example: 0.1, 1.e-3)

type [string] type of data (example: 'float64')

stencil [Stencil] the stencil of the velocities (object of the class *Stencil*)

global_size [list] number of points in each direction

extent [list] number of points to add on each side (max velocities)

coords [ndarray] coordinates of the domain

x [ndarray] x component of the coordinates in the interior domain.

y [ndarray] y component of the coordinates in the interior domain.

z [ndarray] z component of the coordinates in the interior domain.

in_or_out [ndarray] defines the fluid and the solid part (fluid: value=valin, solid: value=valout)

distance [ndarray] defines the distances to the borders. The distance is scaled by dx and is not equal to valin only for the points that reach the border with the specified velocity.

flag [ndarray] NumPy array that defines the flag of the border reached with the specified velocity

valin [int] value in the fluid domain

valout [int] value in the fluid domain

x_halo [ndarray] x component of the coordinates of the whole domain

y_halo [ndarray] y component of the coordinates of the whole domain

z_halo [ndarray] z component of the coordinates of the whole domain

shape_halo [list] shape of the whole domain with the halo points.

shape_in shape of the interior domain.

Methods

construct_mpi_topology(self, dico)	Create the mpi topology
create_coords(self)	Create the coordinates of the interior domain and the whole domain with halo points.
get_bounds(self)	Return the coordinates of the bottom right and upper left corner of the interior domain.
get_bounds_halo(self)	Return the coordinates of the bottom right and upper left corner of the whole domain with halo points.
list_of_labels(self)	Get the list of all the labels used in the geometry.
visualize(self[, viewer_app, view_distance, ...])	Visualize the domain by creating a plot.

pylbm.domain.Domain.construct_mpi_topology

method

Domain.**construct_mpi_topology**(*self*, *dico*)

Create the mpi topology

pylbm.domain.Domain.create_coords

method

Domain.**create_coords**(*self*)
Create the coordinates of the interior domain and the whole domain with halo points.

pylbm.domain.Domain.get_bounds

method

Domain.**get_bounds**(*self*)
Return the coordinates of the bottom right and upper left corner of the interior domain.

pylbm.domain.Domain.get_bounds_halo

method

Domain.**get_bounds_halo**(*self*)
Return the coordinates of the bottom right and upper left corner of the whole domain with halo points.

pylbm.domain.Domain.list_of_labels

method

Domain.**list_of_labels**(*self*)
Get the list of all the labels used in the geometry.

pylbm.domain.Domain.visualize

method

Domain.**visualize**(*self*, *viewer_app=<module 'pylbm.viewer.matplotlib_viewer' from '/home/docs/checkouts/readthedocs.org/user_builds/pylbm/conda/0.4.1/lib/python3.6/site-packages/pylbm-0.4.1-py3.6.egg/pylbm/viewer/matplotlib_viewer.py'>*, *view_distance=False*, *view_in=True*, *view_out=True*, *view_bound=False*, *label=None*, *scale=1*)
Visualize the domain by creating a plot.

Parameters

viewer_app [Viewer, optional] define the viewer to plot the domain default is viewer.matplotlib_viewer

view_distance [boolean or int or list, optional] view the distance between the interior points and the border default is False if True, then all velocities are considered can specify some specific velocities in a list

view_in [boolean, optional] view the inner points default is True

view_out [boolean, optional] view the outer points default is True

view_bound [boolean or int or list, optional] view the points on the bounds default is False

label [int or list, optional] view the distance only for the specified labels

scale [int or float, optional] scale used for the symbol (default 1)

Returns

object views

3.9 the module bounds

The module bounds contains the classes needed to treat the boundary conditions with the LBM formalism

The classes are

Boundary(domain, generator, dico)	Construct the boundary problem by defining the list of indices on the border and the methods used on each label.
BoundaryMethod(istore, ilabel, distance, ...)	Set boundary method.
BounceBack(istore, ilabel, distance, ...)	Boundary condition of type bounce-back
AntiBounceBack(istore, ilabel, distance, ...)	Boundary condition of type anti bounce-back
BouzidiBounceBack(istore, ilabel, distance, ...)	Boundary condition of type Bouzidi bounce-back [BFL01]
BouzidiAntiBounceBack(istore, ilabel, ...)	Boundary condition of type Bouzidi anti bounce-back
Neumann(istore, ilabel, distance, stencil, ...)	Boundary condition of type Neumann
NeumannX(istore, ilabel, distance, stencil, ...)	Boundary condition of type Neumann along the x direction
NeumannY(istore, ilabel, distance, stencil, ...)	Boundary condition of type Neumann along the y direction
NeumannZ(istore, ilabel, distance, stencil, ...)	Boundary condition of type Neumann along the z direction

3.9.1 pylbm.boundary.Boundary

class pylbm.boundary.**Boundary**(*domain*, *generator*, *dico*)

Construct the boundary problem by defining the list of indices on the border and the methods used on each label.

Parameters

domain [pylbm.Domain] the simulation domain

dico [dictionary]

describes the boundaries

- key is a label
- **value are again a dictionnary with**
  - "method" key that gives the boundary method class used (Bounce_back, Anti_bounce_back, ...)
  - "value_bc" key that gives the value on the boundary

Attributes

bv_per_label [dictionnary] for each label key, a list of spatial indices and distance define for each velocity the points on the domain that are on the boundary.

methods [list] list of boundary methods used in the LBM scheme The list contains Boundary_method instance.

3.9.2 pylbm.boundary.BoundaryMethod

class pylbm.boundary.**BoundaryMethod**(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *generator*)

Set boundary method.

Parameters

FIXME [add parameters documentation]

Attributes

feq [ndarray] the equilibrium values of the distribution function on the border

rhs [ndarray] the additional terms to fix the boundary values

distance [ndarray] distance to the border (needed for Bouzidi type conditions)

istore [ndarray] indices of points where we store the boundary condition

ilabel [ndarray] label of the boundary

iload [list] indices of points needed to compute the boundary condition

value_bc [dictionnary] the prescribed values on the border

Methods

fix_iload(self)	Transpose iload and istore.
move2gpu(self)	Move arrays needed to compute the boundary on the GPU memory.
prepare_rhs(self, simulation)	Compute the distribution function at the equilibrium with the value on the border.
update(self, ff, **kwargs)	Update distribution functions with this boundary condition.

pylbm.boundary.BoundaryMethod.fix_iload

method

BoundaryMethod.**fix_iload**(*self*)

Transpose iload and istore.

Must be fix in a future version.

pylbm.boundary.BoundaryMethod.move2gpu

method

BoundaryMethod.**move2gpu**(*self*)

Move arrays needed to compute the boundary on the GPU memory.

pylbm.boundary.BoundaryMethod.prepare_rhs

method

BoundaryMethod.**prepare_rhs**(*self*, *simulation*)

Compute the distribution function at the equilibrium with the value on the border.

Parameters

simulation [Simulation] simulation class

pylbm.boundary.BoundaryMethod.update

method

BoundaryMethod.**update**(*self*, *ff*, ***kwargs*)

Update distribution functions with this boundary condition.

Parameters

ff [array] The distribution functions

3.9.3 pylbm.boundary.BounceBack

class pylbm.boundary.**BounceBack**(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *generator*)

Boundary condition of type bounce-back

Notes

Attributes

function Return the generated function

Methods

fix_iload(self)	Transpose iload and istore.
generate(self, sorder)	Generate the numerical code.
move2gpu(self)	Move arrays needed to compute the boundary on the GPU memory.
prepare_rhs(self, simulation)	Compute the distribution function at the equilibrium with the value on the border.
set_iload(self)	Compute the indices that are needed (symmertic velocities and space indices).
set_rhs(self)	Compute and set the additional terms to fix the boundary values.
update(self, ff, **kwargs)	Update distribution functions with this boundary condition.

pylbm.boundary.BounceBack.fix_iload

method

BounceBack.**fix_iload**(*self*)

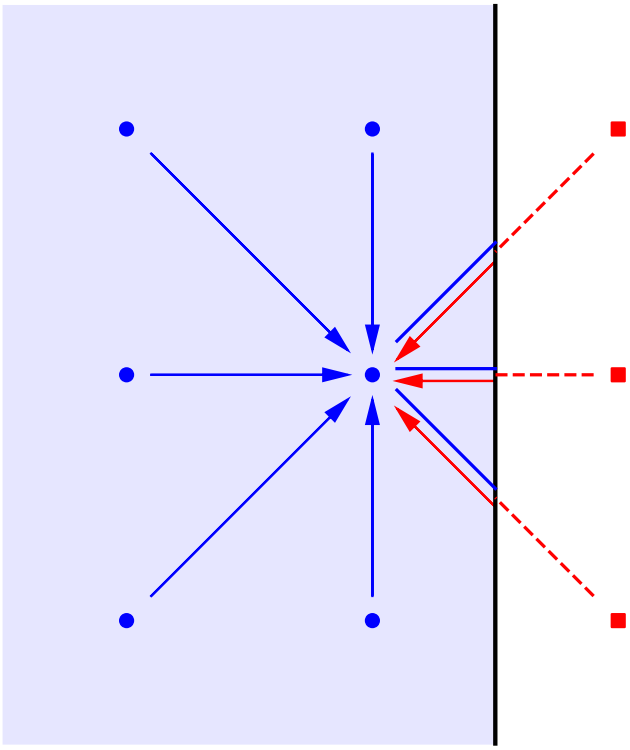
Transpose iload and istore.

Must be fix in a future version.

pylbm.boundary.BounceBack.generate

method

bounce back: the exiting particles bounce back without sign modification



BounceBack.**generate**(*self*, *sorder*)
Generate the numerical code.

Parameters

sorder [list] the order of nv, nx, ny and nz

pylbm.boundary.BounceBack.move2gpu

method

BounceBack.**move2gpu**(*self*)
Move arrays needed to compute the boundary on the GPU memory.

pylbm.boundary.BounceBack.prepare_rhs

method

BounceBack.**prepare_rhs**(*self*, *simulation*)
Compute the distribution function at the equilibrium with the value on the border.

Parameters

simulation [Simulation] simulation class

pylbm.boundary.BounceBack.set_iload

method

BounceBack.**set_iload**(*self*)
Compute the indices that are needed (symmertic velocities and space indices).

pylbm.boundary.BounceBack.set_rhs

method

BounceBack.**set_rhs**(*self*)
Compute and set the additional terms to fix the boundary values.

pylbm.boundary.BounceBack.update

method

BounceBack.**update**(*self*, *ff*, ***kwargs*)
Update distribution functions with this boundary condition.

Parameters

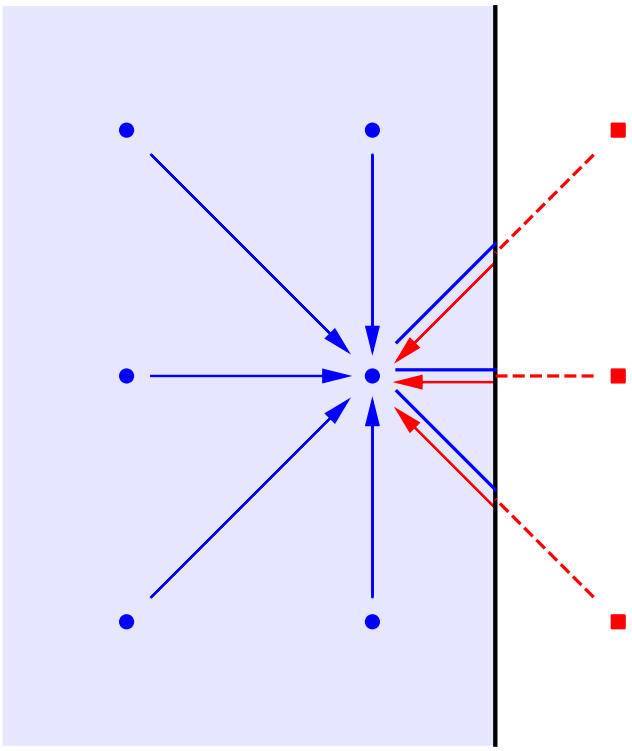
ff [array] The distribution functions

3.9.4 pylbm.boundary.AntiBounceBack

class pylbm.boundary.**AntiBounceBack**(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *generator*)
Boundary condition of type anti bounce-back

Notes

anti bounce back: the exiting particles bounce back with sign modification



Attributes

function Return the generated function

Methods

fix_iload(self)	Transpose iload and istore.
generate(self, sorder)	Generate the numerical code.
move2gpu(self)	Move arrays needed to compute the boundary on the GPU memory.

Continued on next page

Table 28 – continued from previous page

prepare_rhs(self, simulation)	Compute the distribution function at the equilibrium with the value on the border.
set_iload(self)	Compute the indices that are needed (symmertic velocities and space indices).
set_rhs(self)	Compute and set the additional terms to fix the boundary values.
update(self, ff, **kwargs)	Update distribution functions with this boundary condition.

pylbm.boundary.AntiBounceBack.fix_iload

method

AntiBounceBack.**fix_iload**(*self*)

Transpose iload and istore.

Must be fix in a future version.

pylbm.boundary.AntiBounceBack.generate

method

AntiBounceBack.**generate**(*self*, *sorder*)

Generate the numerical code.

Parameters

sorder [list] the order of nv, nx, ny and nz

pylbm.boundary.AntiBounceBack.move2gpu

method

AntiBounceBack.**move2gpu**(*self*)

Move arrays needed to compute the boundary on the GPU memory.

pylbm.boundary.AntiBounceBack.prepare_rhs

method

AntiBounceBack.**prepare_rhs**(*self*, *simulation*)

Compute the distribution function at the equilibrium with the value on the border.

Parameters

simulation [Simulation] simulation class

pylbm.boundary.AntiBounceBack.set_iload

method

AntiBounceBack.**set_iload**(*self*)

Compute the indices that are needed (symmertic velocities and space indices).

pylbm.boundary.AntiBounceBack.set_rhs

method

AntiBounceBack.**set_rhs**(*self*)
Compute and set the additional terms to fix the boundary values.

pylbm.boundary.AntiBounceBack.update

method

AntiBounceBack.**update**(*self*, *ff*, ***kwargs*)
Update distribution functions with this boundary condition.

Parameters

ff [array] The distribution functions

3.9.5 pylbm.boundary.BouzidiBounceBack

class pylbm.boundary.**BouzidiBounceBack**(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *generator*)
Boundary condition of type Bouzidi bounce-back [BFL01]

Notes

Attributes

function Return the generated function

Methods

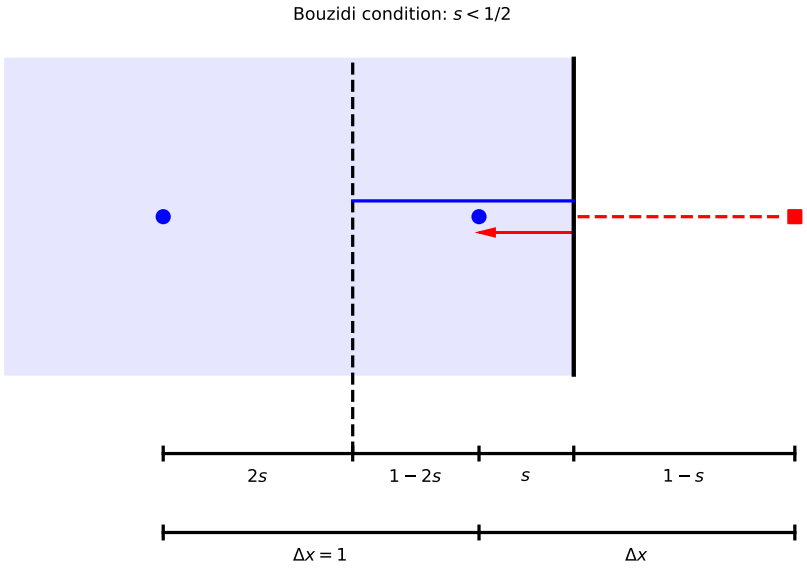
fix_iload(self)	Transpose iload and istore.
generate(self, sorder)	Generate the numerical code.
move2gpu(self)	Move arrays needed to compute the boundary on the GPU memory.
prepare_rhs(self, simulation)	Compute the distribution function at the equilibrium with the value on the border.
set_iload(self)	Compute the indices that are needed (symmertic velocities and space indices).
set_rhs(self)	Compute and set the additional terms to fix the boundary values.
update(self, ff, **kwargs)	Update distribution functions with this boundary condition.

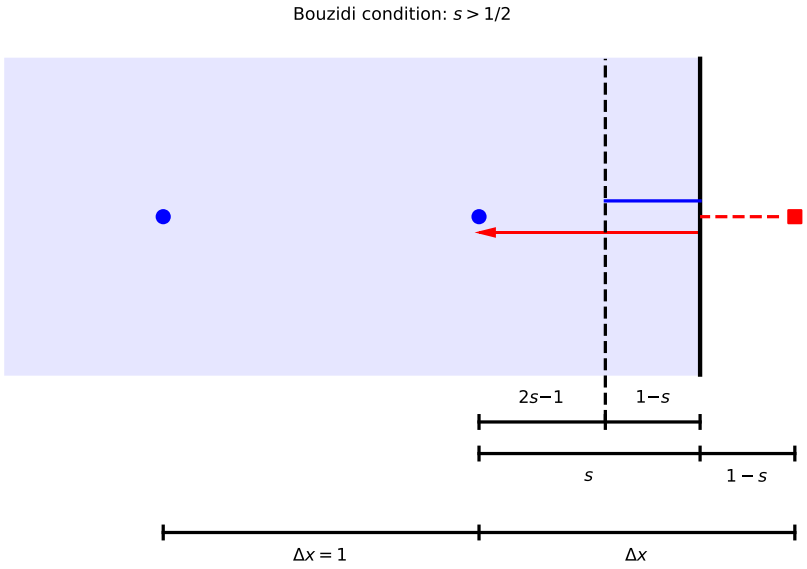
pylbm.boundary.BouzidiBounceBack.fix_iload

method

BouzidiBounceBack.**fix_iload**(*self*)
Transpose iload and istore.

Must be fix in a future version.


Bouzidi condition: $s < 1/2$
$2s$
$1-2s$
s
$1-s$
$\Delta x = 1$
Δx


Bouzidi condition: $s > 1/2$
$2s-1$
$1-s$
s
$1-s$
$\Delta x = 1$
Δx

pylbm.boundary.BouzidiBounceBack.generate

method

BouzidiBounceBack.**generate**(*self*, *sorder*)
Generate the numerical code.

Parameters

sorder [list] the order of nv, nx, ny and nz

pylbm.boundary.BouzidiBounceBack.move2gpu

method

BouzidiBounceBack.**move2gpu**(*self*)
Move arrays needed to compute the boundary on the GPU memory.

pylbm.boundary.BouzidiBounceBack.prepare_rhs

method

BouzidiBounceBack.**prepare_rhs**(*self*, *simulation*)
Compute the distribution function at the equilibrium with the value on the border.

Parameters

simulation [Simulation] simulation class

pylbm.boundary.BouzidiBounceBack.set_iload

method

BouzidiBounceBack.**set_iload**(*self*)
Compute the indices that are needed (symmertic velocities and space indices).

pylbm.boundary.BouzidiBounceBack.set_rhs

method

BouzidiBounceBack.**set_rhs**(*self*)
Compute and set the additional terms to fix the boundary values.

pylbm.boundary.BouzidiBounceBack.update

method

BouzidiBounceBack.**update**(*self*, *ff*, ***kwargs*)
Update distribution functions with this boundary condition.

Parameters

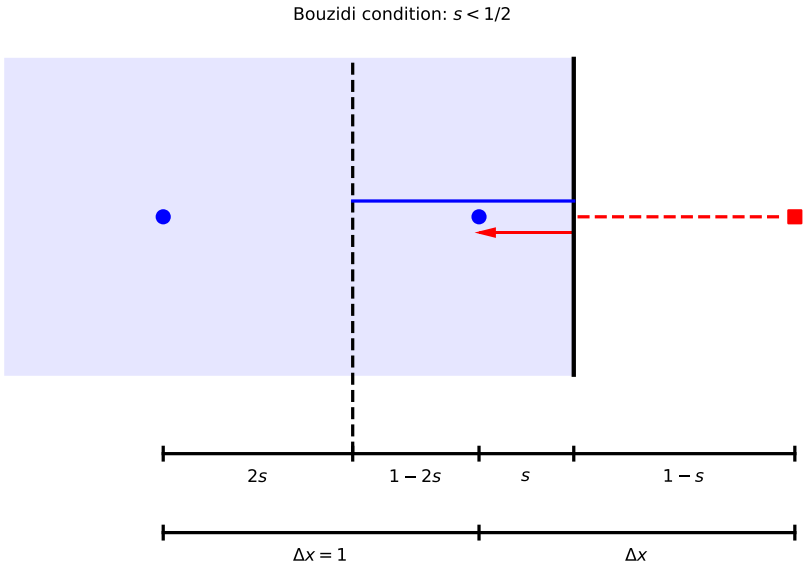
ff [array] The distribution functions

3.9.6 pylbm.boundary.BouzidiAntiBounceBack

class pylbm.boundary.**BouzidiAntiBounceBack**(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *generator*)

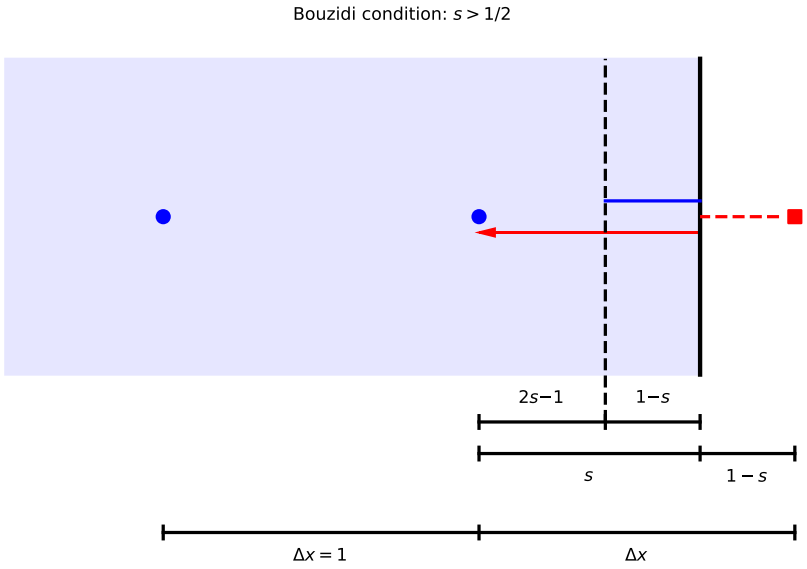
Boundary condition of type Bouzidi anti bounce-back

Notes



Attributes

function Return the generated function


Bouzidi condition: $s > 1/2$
$2s-1$
$1-s$
s
$1-s$
$\Delta x = 1$
Δx

Methods

fix_iload(self)	Transpose iload and istore.
generate(self, sorder)	Generate the numerical code.
move2gpu(self)	Move arrays needed to compute the boundary on the GPU memory.
prepare_rhs(self, simulation)	Compute the distribution function at the equilibrium with the value on the border.
set_iload(self)	Compute the indices that are needed (symmertic velocities and space indices).
set_rhs(self)	Compute and set the additional terms to fix the boundary values.
update(self, ff, **kwargs)	Update distribution functions with this boundary condition.

pylbm.boundary.BouzidiAntiBounceBack.fix_iload

method

BouzidiAntiBounceBack.**fix_iload**(*self*)

Transpose iload and istore.

Must be fix in a future version.

pylbm.boundary.BouzidiAntiBounceBack.generate

method

BouzidiAntiBounceBack.**generate**(*self*, *sorder*)

Generate the numerical code.

Parameters

sorder [list] the order of nv, nx, ny and nz

pylbm.boundary.BouzidiAntiBounceBack.move2gpu

method

BouzidiAntiBounceBack.**move2gpu**(*self*)

Move arrays needed to compute the boundary on the GPU memory.

pylbm.boundary.BouzidiAntiBounceBack.prepare_rhs

method

BouzidiAntiBounceBack.**prepare_rhs**(*self*, *simulation*)

Compute the distribution function at the equilibrium with the value on the border.

Parameters

simulation [Simulation] simulation class

pylbm.boundary.BouzidiAntiBounceBack.set_iload

method

BouzidiAntiBounceBack.**set_iload**(*self*)
Compute the indices that are needed (symmertic velocities and space indices).

pylbm.boundary.BouzidiAntiBounceBack.set_rhs

method

BouzidiAntiBounceBack.**set_rhs**(*self*)
Compute and set the additional terms to fix the boundary values.

pylbm.boundary.BouzidiAntiBounceBack.update

method

BouzidiAntiBounceBack.**update**(*self*, *ff*, ***kwargs*)
Update distribution functions with this boundary condition.

Parameters

ff [array] The distribution functions

3.9.7 pylbm.boundary.Neumann

class pylbm.boundary.**Neumann**(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *generator*)
Boundary condition of type Neumann

Attributes

function Return the generated function

Methods

fix_iload(self)	Transpose iload and istore.
generate(self, sorder)	Generate the numerical code.
move2gpu(self)	Move arrays needed to compute the boundary on the GPU memory.
prepare_rhs(self, simulation)	Compute the distribution function at the equilibrium with the value on the border.
set_iload(self)	Compute the indices that are needed (symmertic velocities and space indices).
set_rhs(self)	Compute and set the additional terms to fix the boundary values.
update(self, ff, **kwargs)	Update distribution functions with this boundary condition.

pylbm.boundary.Neumann.fix_iload

method

Neumann.**fix_iload**(*self*)

Transpose iload and istore.

Must be fix in a future version.

pylbm.boundary.Neumann.generate

method

Neumann.**generate**(*self*, *sorder*)
Generate the numerical code.

Parameters

sorder [list] the order of nv, nx, ny and nz

pylbm.boundary.Neumann.move2gpu

method

Neumann.**move2gpu**(*self*)
Move arrays needed to compute the boundary on the GPU memory.

pylbm.boundary.Neumann.prepare_rhs

method

Neumann.**prepare_rhs**(*self*, *simulation*)
Compute the distribution function at the equilibrium with the value on the border.

Parameters

simulation [Simulation] simulation class

pylbm.boundary.Neumann.set_iload

method

Neumann.**set_iload**(*self*)
Compute the indices that are needed (symmertic velocities and space indices).

pylbm.boundary.Neumann.set_rhs

method

Neumann.**set_rhs**(*self*)
Compute and set the additional terms to fix the boundary values.

pylbm.boundary.Neumann.update

method

Neumann.**update**(*self*, *ff*, ***kwargs*)
Update distribution functions with this boundary condition.

Parameters

ff [array] The distribution functions

3.9.8 pylbm.boundary.NeumannX

class pylbm.boundary.**NeumannX**(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *generator*)

Boundary condition of type Neumann along the x direction

Attributes

function Return the generated function

Methods

fix_iload(self)	Transpose iload and istore.
generate(self, sorder)	Generate the numerical code.
move2gpu(self)	Move arrays needed to compute the boundary on the GPU memory.
prepare_rhs(self, simulation)	Compute the distribution function at the equilibrium with the value on the border.
set_iload(self)	Compute the indices that are needed (symmertic velocities and space indices).
set_rhs(self)	Compute and set the additional terms to fix the boundary values.
update(self, ff, **kwargs)	Update distribution functions with this boundary condition.

pylbm.boundary.NeumannX.fix_iload

method

NeumannX.**fix_iload**(*self*)

Transpose iload and istore.

Must be fix in a future version.

pylbm.boundary.NeumannX.generate

method

NeumannX.**generate**(*self*, *sorder*)

Generate the numerical code.

Parameters

sorder [list] the order of nv, nx, ny and nz

pylbm.boundary.NeumannX.move2gpu

method

NeumannX.**move2gpu**(*self*)

Move arrays needed to compute the boundary on the GPU memory.

pylbm.boundary.NeumannX.prepare_rhs

method

NeumannX.**prepare_rhs**(*self*, *simulation*)
Compute the distribution function at the equilibrium with the value on the border.

Parameters

simulation [Simulation] simulation class

pylbm.boundary.NeumannX.set_iload

method

NeumannX.**set_iload**(*self*)
Compute the indices that are needed (symmertic velocities and space indices).

pylbm.boundary.NeumannX.set_rhs

method

NeumannX.**set_rhs**(*self*)
Compute and set the additional terms to fix the boundary values.

pylbm.boundary.NeumannX.update

method

NeumannX.**update**(*self*, *ff*, ***kwargs*)
Update distribution functions with this boundary condition.

Parameters

ff [array] The distribution functions

3.9.9 pylbm.boundary.NeumannY

class pylbm.boundary.**NeumannY**(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *generator*)
Boundary condition of type Neumann along the y direction

Attributes

function Return the generated function

Methods

fix_iload(self)	Transpose iload and istore.
generate(self, sorder)	Generate the numerical code.
move2gpu(self)	Move arrays needed to compute the boundary on the GPU memory.
prepare_rhs(self, simulation)	Compute the distribution function at the equilibrium with the value on the border.

Continued on next page

Table 33 – continued from previous page

set_iload(self)	Compute the indices that are needed (symmertic velocities and space indices).
set_rhs(self)	Compute and set the additional terms to fix the boundary values.
update(self, ff, **kwargs)	Update distribution functions with this boundary condition.

pylbm.boundary.NeumannY.fix_iload

method

NeumannY.**fix_iload**(*self*)
Transpose iload and istore.

Must be fix in a future version.

pylbm.boundary.NeumannY.generate

method

NeumannY.**generate**(*self*, *sorder*)
Generate the numerical code.

Parameters

sorder [list] the order of nv, nx, ny and nz

pylbm.boundary.NeumannY.move2gpu

method

NeumannY.**move2gpu**(*self*)
Move arrays needed to compute the boundary on the GPU memory.

pylbm.boundary.NeumannY.prepare_rhs

method

NeumannY.**prepare_rhs**(*self*, *simulation*)
Compute the distribution function at the equilibrium with the value on the border.

Parameters

simulation [Simulation] simulation class

pylbm.boundary.NeumannY.set_iload

method

NeumannY.**set_iload**(*self*)
Compute the indices that are needed (symmertic velocities and space indices).

pylbm.boundary.NeumannY.set_rhs

method

NeumannY.**set_rhs**(*self*)
Compute and set the additional terms to fix the boundary values.

pylbm.boundary.NeumannY.update

method

NeumannY.**update**(*self*, *ff*, ***kwargs*)
Update distribution functions with this boundary condition.

Parameters

ff [array] The distribution functions

3.9.10 pylbm.boundary.NeumannZ

class pylbm.boundary.**NeumannZ**(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *generator*)
Boundary condition of type Neumann along the z direction

Attributes

function Return the generated function

Methods

fix_iload(self)	Transpose iload and istore.
generate(self, sorder)	Generate the numerical code.
move2gpu(self)	Move arrays needed to compute the boundary on the GPU memory.
prepare_rhs(self, simulation)	Compute the distribution function at the equilibrium with the value on the border.
set_iload(self)	Compute the indices that are needed (symmertic velocities and space indices).
set_rhs(self)	Compute and set the additional terms to fix the boundary values.
update(self, ff, **kwargs)	Update distribution functions with this boundary condition.

pylbm.boundary.NeumannZ.fix_iload

method

NeumannZ.**fix_iload**(*self*)
Transpose iload and istore.

Must be fix in a future version.

pylbm.boundary.NeumannZ.generate

method

NeumannZ.**generate**(*self*, *sorder*)
Generate the numerical code.

Parameters

sorder [list] the order of nv, nx, ny and nz

pylbm.boundary.NeumannZ.move2gpu

method

NeumannZ.**move2gpu**(*self*)
Move arrays needed to compute the boundary on the GPU memory.

pylbm.boundary.NeumannZ.prepare_rhs

method

NeumannZ.**prepare_rhs**(*self*, *simulation*)
Compute the distribution function at the equilibrium with the value on the border.

Parameters

simulation [Simulation] simulation class

pylbm.boundary.NeumannZ.set_iload

method

NeumannZ.**set_iload**(*self*)
Compute the indices that are needed (symmertic velocities and space indices).

pylbm.boundary.NeumannZ.set_rhs

method

NeumannZ.**set_rhs**(*self*)
Compute and set the additional terms to fix the boundary values.

pylbm.boundary.NeumannZ.update

method

NeumannZ.**update**(*self*, *ff*, ***kwargs*)
Update distribution functions with this boundary condition.

Parameters

ff [array] The distribution functions

3.10 the module algorithm

BaseAlgorithm(scheme, sorder, generator[, ...])	

Methods

PullAlgorithm(scheme, sorder, generator[, ...])	

Methods

3.10.1 pylbm.algorithm.BaseAlgorithm

class pylbm.algorithm.**BaseAlgorithm**(*scheme*, *sorder*, *generator*, *settings=None*)

Methods

call_function(self, function_name, simulation)	Call the generated function.
equilibrium(self)	Return the code expression which computes the equilibrium on the whole domain.
equilibrium_local(self, m)	Return symbolic expression which computes the equilibrium.
f2m(self)	Return the code expression which computes the moments from the distributed functions on the whole domain.
f2m_local(self, f, m[, with_rel_velocity])	Return symbolic expression which computes the moments from the distributed functions.
generate(self)	Define the routines which must be generate by code generator of SymPy for a given generator.
m2f(self)	Return the code expression which computes the distributed functions from the moments on the whole domain.
m2f_local(self, m, f[, with_rel_velocity])	Return symbolic expression which computes the distributed functions from the moments.
one_time_step(self)	Return the code expression which makes one time step of LBM algorithm on the whole inner domain.
one_time_step_local(self, f, fnew, m)	Return symbolic expression which makes one time step of LBM algorithm:
relaxation(self)	Return the code expression which computes the relaxation on the whole domain.
relaxation_local(self, m[, with_rel_velocity])	Return symbolic expression which computes the relaxation operator.
source_term(self)	Return the code expression which computes the source terms on the whole inner domain.
source_term_local(self, m)	Return symbolic expression which computes the source term using explicit Euler (should be more flexible in a near future).
transport(self)	Return the code expression of the lbm transport on the whole inner domain.
transport_local(self, f, fnew)	Return the symbolic expression of the lbm transport.

pylbm.algorithm.BaseAlgorithm.call_function

method

BaseAlgorithm.**call_function**(*self*, *function_name*, *simulation*, *m_user=None*, *f_user=None*, ***kwargs*)

Call the generated function.

pylbm.algorithm.BaseAlgorithm.equilibrium

method

BaseAlgorithm.**equilibrium**(*self*)

Return the code expression which computes the equilibrium on the whole domain.

pylbm.algorithm.BaseAlgorithm.equilibrium_local

method

BaseAlgorithm.**equilibrium_local**(*self*, *m*)

Return symbolic expression which computes the equilibrium.

Parameters

m [SymPy Matrix] indexed objects for the moments

pylbm.algorithm.BaseAlgorithm.f2m

method

BaseAlgorithm.**f2m**(*self*)

Return the code expression which computes the moments from the distributed functions on the whole domain.

pylbm.algorithm.BaseAlgorithm.f2m_local

method

BaseAlgorithm.**f2m_local**(*self*, *f*, *m*, *with_rel_velocity=False*)

Return symbolic expression which computes the moments from the distributed functions.

Parameters

f [SymPy Matrix] indexed objects for the distributed functions

m [SymPy Matrix] indexed objects for the moments

with_rel_velocity [boolean] check if the scheme uses relative velocity. If yes, the conserved moments must be computed first. (default is False)

pylbm.algorithm.BaseAlgorithm.generate

method

BaseAlgorithm.**generate**(*self*)

Define the routines which must be generate by code generator of SymPy for a given generator.

pylbm.algorithm.BaseAlgorithm.m2f

method

BaseAlgorithm.**m2f**(*self*)

Return the code expression which computes the distributed functions from the moments on the whole domain.

pylbm.algorithm.BaseAlgorithm.m2f_local

method

BaseAlgorithm.**m2f_local**(*self*, *m*, *f*, *with_rel_velocity=False*)

Return symbolic expression which computes the distributed functions from the moments.

Parameters

m [SymPy Matrix] indexed objects for the moments

f [SymPy Matrix] indexed objects for the distributed functions

with_rel_velocity [boolean] check if the scheme uses relative velocity. If yes, the conserved moments must be computed first. (default is False)

pylbm.algorithm.BaseAlgorithm.one_time_step

method

BaseAlgorithm.**one_time_step**(*self*)

Return the code expression which makes one time step of LBM algorithm on the whole inner domain.

pylbm.algorithm.BaseAlgorithm.one_time_step_local

method

BaseAlgorithm.**one_time_step_local**(*self*, *f*, *fnew*, *m*)

Return symbolic expression which makes one time step of LBM algorithm:

- transport
- compute the moments from the distributed functions
- source terms with dt/2 (with the moments)
- relaxation (with the moments)
- source terms with dt/2 (with the moments)
- compute the new distributed functions from the moments

Parameters

f [SymPy Matrix] indexed objects for the old distributed functions

fnew [SymPy Matrix] indexed objects for the new distributed functions

m [SymPy Matrix] indexed objects for the moments

pylbm.algorithm.BaseAlgorithm.relaxation

method

BaseAlgorithm.**relaxation**(*self*)
Return the code expression which computes the relaxation on the whole domain.

pylbm.algorithm.BaseAlgorithm.relaxation_local

method

BaseAlgorithm.**relaxation_local**(*self*, *m*, *with_rel_velocity=False*)
Return symbolic expression which computes the relaxation operator.

Parameters

m [SymPy Matrix] indexed objects for the moments

with_rel_velocity [boolean] check if the scheme uses relative velocity. (default is False)

pylbm.algorithm.BaseAlgorithm.source_term

method

BaseAlgorithm.**source_term**(*self*)
Return the code expression which computes the source terms on the whole inner domain.

pylbm.algorithm.BaseAlgorithm.source_term_local

method

BaseAlgorithm.**source_term_local**(*self*, *m*)
Return symbolic expression which computes the source term using explicit Euler (should be more flexible in a near future).

Parameters

m [SymPy Matrix] indexed objects for the moments

pylbm.algorithm.BaseAlgorithm.transport

method

BaseAlgorithm.**transport**(*self*)
Return the code expression of the lbm transport on the whole inner domain.

pylbm.algorithm.BaseAlgorithm.transport_local

method

BaseAlgorithm.**transport_local**(*self*, *f*, *fnew*)
Return the symbolic expression of the lbm transport.

Parameters

f [SymPy Matrix] indexed objects of rhs for the distributed functions

fnew [SymPy Matrix] indexed objects of lhs for the distributed functions

coords	
relative_velocity	
restore_conserved_moments	

3.10.2 pylbm.algorithm.PullAlgorithm

class pylbm.algorithm.**PullAlgorithm**(*scheme*, *sorder*, *generator*, *settings=None*)

Methods

call_function(self, function_name, simulation)	Call the generated function.
equilibrium(self)	Return the code expression which computes the equilibrium on the whole domain.
equilibrium_local(self, m)	Return symbolic expression which computes the equilibrium.
f2m(self)	Return the code expression which computes the moments from the distributed functions on the whole domain.
f2m_local(self, f, m[, with_rel_velocity])	Return symbolic expression which computes the moments from the distributed functions.
generate(self)	Define the routines which must be generate by code generator of SymPy for a given generator.
m2f(self)	Return the code expression which computes the distributed functions from the moments on the whole domain.
m2f_local(self, m, f[, with_rel_velocity])	Return symbolic expression which computes the distributed functions from the moments.
one_time_step(self)	Return the code expression which makes one time step of LBM algorithm on the whole inner domain.
one_time_step_local(self, f, fnew, m)	Return symbolic expression which makes one time step of LBM algorithm using the pull algorithm.
relaxation(self)	Return the code expression which computes the relaxation on the whole domain.
relaxation_local(self, m[, with_rel_velocity])	Return symbolic expression which computes the relaxation operator.
source_term(self)	Return the code expression which computes the source terms on the whole inner domain.
source_term_local(self, m)	Return symbolic expression which computes the source term using explicit Euler (should be more flexible in a near future).
transport(self)	Return the code expression of the lbm transport on the whole inner domain.
transport_local(self, f, fnew)	Return the symbolic expression of the lbm transport.

pylbm.algorithm.PullAlgorithm.call_function

method

PullAlgorithm.**call_function**(*self*, *function_name*, *simulation*, *m_user=None*, *f_user=None*, ***kwargs*)
Call the generated function.

pylbm.algorithm.PullAlgorithm.equilibrium

method

PullAlgorithm.**equilibrium**(*self*)
Return the code expression which computes the equilibrium on the whole domain.

pylbm.algorithm.PullAlgorithm.equilibrium_local

method

PullAlgorithm.**equilibrium_local**(*self*, *m*)
Return symbolic expression which computes the equilibrium.

Parameters

m [SymPy Matrix] indexed objects for the moments

pylbm.algorithm.PullAlgorithm.f2m

method

PullAlgorithm.**f2m**(*self*)
Return the code expression which computes the moments from the distributed functions on the whole domain.

pylbm.algorithm.PullAlgorithm.f2m_local

method

PullAlgorithm.**f2m_local**(*self*, *f*, *m*, *with_rel_velocity=False*)
Return symbolic expression which computes the moments from the distributed functions.

Parameters

f [SymPy Matrix] indexed objects for the distributed functions

m [SymPy Matrix] indexed objects for the moments

with_rel_velocity [boolean] check if the scheme uses relative velocity. If yes, the conserved moments must be computed first. (default is False)

pylbm.algorithm.PullAlgorithm.generate

method

PullAlgorithm.**generate**(*self*)
Define the routines which must be generate by code generator of SymPy for a given generator.

pylbm.algorithm.PullAlgorithm.m2f

method

PullAlgorithm.**m2f**(*self*)

Return the code expression which computes the distributed functions from the moments on the whole domain.

pylbm.algorithm.PullAlgorithm.m2f_local

method

PullAlgorithm.**m2f_local**(*self*, *m*, *f*, *with_rel_velocity=False*)

Return symbolic expression which computes the distributed functions from the moments.

Parameters

m [SymPy Matrix] indexed objects for the moments

f [SymPy Matrix] indexed objects for the distributed functions

with_rel_velocity [boolean] check if the scheme uses relative velocity. If yes, the conserved moments must be computed first. (default is False)

pylbm.algorithm.PullAlgorithm.one_time_step

method

PullAlgorithm.**one_time_step**(*self*)

Return the code expression which makes one time step of LBM algorithm on the whole inner domain.

pylbm.algorithm.PullAlgorithm.one_time_step_local

method

PullAlgorithm.**one_time_step_local**(*self*, *f*, *fnew*, *m*)

Return symbolic expression which makes one time step of LBM algorithm using the pull algorithm. The difference with the basic algorithm is the transport and the computation of the moments from the distributed functions is made in one step.

- compute the moments from the distributed functions + transport
- source terms with dt/2 (with the moments)
- relaxation (with the moments)
- source terms with dt/2 (with the moments)
- compute the new distributed functions from the moments

Parameters

f [SymPy Matrix] indexed objects for the old distributed functions

fnew [SymPy Matrix] indexed objects for the new distributed functions

m [SymPy Matrix] indexed objects for the moments

pylbm.algorithm.PullAlgorithm.relaxation

method

PullAlgorithm.**relaxation**(*self*)
Return the code expression which computes the relaxation on the whole domain.

pylbm.algorithm.PullAlgorithm.relaxation_local

method

PullAlgorithm.**relaxation_local**(*self*, *m*, *with_rel_velocity=False*)
Return symbolic expression which computes the relaxation operator.

Parameters

m [SymPy Matrix] indexed objects for the moments

with_rel_velocity [boolean] check if the scheme uses relative velocity. (default is False)

pylbm.algorithm.PullAlgorithm.source_term

method

PullAlgorithm.**source_term**(*self*)
Return the code expression which computes the source terms on the whole inner domain.

pylbm.algorithm.PullAlgorithm.source_term_local

method

PullAlgorithm.**source_term_local**(*self*, *m*)
Return symbolic expression which computes the source term using explicit Euler (should be more flexible in a near future).

Parameters

m [SymPy Matrix] indexed objects for the moments

pylbm.algorithm.PullAlgorithm.transport

method

PullAlgorithm.**transport**(*self*)
Return the code expression of the lbm transport on the whole inner domain.

pylbm.algorithm.PullAlgorithm.transport_local

method

PullAlgorithm.**transport_local**(*self*, *f*, *fnew*)
Return the symbolic expression of the lbm transport.

Parameters

f [SymPy Matrix] indexed objects of rhs for the distributed functions

fnew [SymPy Matrix] indexed objects of lhs for the distributed functions

coords	
relative_velocity	
restore_conserved_moments	

3.11 the module storage

Array(nv, gspace_size, vmax[, sorder, ...])	This class defines the storage of the moments and distribution functions in pylbm.
SOA(nv, gspace_size, vmax, mpi_topo[, ...])	This class defines a structure of arrays to store the unknowns of the lattice Boltzmann schemes.
AOS(nv, gspace_size, vmax, mpi_topo[, ...])	This class defines an array of structures to store the unknowns of the lattice Boltzmann schemes.

3.11.1 pylbm.storage.Array

class pylbm.storage.**Array**(*nv*, *gspace_size*, *vmax*, *sorder=None*, *mpi_topo=None*, *dtype=<class 'numpy.float64'>*, *gpu_support=False*)

This class defines the storage of the moments and distribution functions in pylbm.

It sets the storage in memory and how to access.

Parameters

nv: int number of velocities

gspace_size: list number of points in each direction including the fictitious point

vmax: list the size of the fictitious points in each direction

sorder: list the order of nv, nx, ny and nz Default is None which mean [nv, nx, ny, nz]

mpi_topo: MpiTopology the mpi topology

dtype: type the type of the array. Default is numpy.double

gpu_support [bool] true if GPU is needed

Attributes

array

nspace the space size.

nv the number of velocities.

shape the shape of the array that stores the data.

size the size of the array that stores the data.

Methods

generate(self, generator)	generate periodic conditions functions for loo.py backend.

Continued on next page

Table 39 – continued from previous page

set_conserved_moments(self, consm)	add conserved moments information to have a direct access.
update(self)	update ghost points on the interface with the datas of the neighbors.

pylbm.storage.Array.generate

method

`Array.`**`generate`**(*self*, *generator*)
generate periodic conditions functions for loo.py backend.

pylbm.storage.Array.set_conserved_moments

method

`Array.`**`set_conserved_moments`**(*self*, *consm*)
add conserved moments information to have a direct access.

Parameters

consm [dict] set the name and the location of the conserved moments. The format is

key : the conserved moment (sympy symbol or string)

value : list of 2 integers

first item : the scheme number

second item : the index of the conserved moment in this scheme

pylbm.storage.Array.update

method

`Array.`**`update`**(*self*)
update ghost points on the interface with the datas of the neighbors.

3.11.2 pylbm.storage.SOA

class `pylbm.storage.`**`SOA`**(*nv*, *gspace_size*, *vmax*, *mpi_topo*, *dtype=<class 'numpy.float64'>*, *gpu_support=False*)
This class defines a structure of arrays to store the unknowns of the lattice Boltzmann schemes.

Parameters

nv: int number of velocities

gspace_size: list number of points in each direction including the fictitious point

vmax: list the size of the fictitious points in each direction

mpi_topo: MpiTopology the mpi topology

dtype: type the type of the array. Default is numpy.double

gpu_support: bool True if GPU is needed

Attributes

array

nspace the space size.

nv the number of velocities.

shape the shape of the array that stores the data.

size the size of the array that stores the data.

Methods

generate(self, generator)	generate periodic conditions functions for loo.py backend.
reshape(self)	reshape.
set_conserved_moments(self, consm)	add conserved moments information to have a direct access.
update(self)	update ghost points on the interface with the datas of the neighbors.

pylbm.storage.SOA.generate

method

SOA.**generate**(*self*, *generator*)
generate periodic conditions functions for loo.py backend.

pylbm.storage.SOA.reshape

method

SOA.**reshape**(*self*)
reshape.

pylbm.storage.SOA.set_conserved_moments

method

SOA.**set_conserved_moments**(*self*, *consm*)
add conserved moments information to have a direct access.

Parameters

consm [dict] set the name and the location of the conserved moments. The format is

key : the conserved moment (sympy symbol or string)

value : list of 2 integers

first item : the scheme number

second item : the index of the conserved moment in this scheme

pylbm.storage.SOA.update

method

SOA.**update**(*self*)
update ghost points on the interface with the datas of the neighbors.

3.11.3 pylbm.storage.AOS

class pylbm.storage.**AOS**(*nv*, *gspace_size*, *vmax*, *mpi_topo*, *dtype=<class 'numpy.float64'>*, *gpu_support=False*)
This class defines an array of structures to store the unknowns of the lattice Boltzmann schemes.

Parameters

nv: int number of velocities

gspace_size: list number of points in each direction including the fictitious point

vmax: list the size of the fictitious points in each direction

mpi_topo: MpiTopology the mpi topology

dtype: type the type of the array. Default is numpy.double

gpu_support: bool True if GPU is needed

Attributes

array

nspace the space size.

nv the number of velocities.

shape the shape of the array that stores the data.

size the size of the array that stores the data.

Methods

generate(self, generator)	generate periodic conditions functions for loo.py backend.
reshape(self)	
set_conserved_moments(self, consm)	add conserved moments information to have a direct access.
update(self)	update ghost points on the interface with the datas of the neighbors.

pylbm.storage.AOS.generate

method

AOS.**generate**(*self*, *generator*)
generate periodic conditions functions for loo.py backend.

pylbm.storage.AOS.reshape

method

AOS.**reshape**(*self*)

pylbm.storage.AOS.set_conserved_moments

method

AOS.**set_conserved_moments**(*self*, *consm*)
add conserved moments information to have a direct access.

Parameters

consm [dict] set the name and the location of the conserved moments. The format is

key : the conserved moment (sympy symbol or string)

value : list of 2 integers

first item : the scheme number

second item : the index of the conserved moment in this scheme

pylbm.storage.AOS.update

method

AOS.**update**(*self*)
update ghost points on the interface with the datas of the neighbors.

CHAPTER

FOUR

REFERENCES

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- search

BIBLIOGRAPHY

[dH92] D. D'HUMIERES, *Generalized Lattice-Boltzmann Equations*, Rarefied Gas Dynamics: Theory and Simulations, **159**, pp. 450-458, AIAA Progress in astronomics and aeronautics (1992).

[D08] F. DUBOIS, *Equivalent partial differential equations of a lattice Boltzmann scheme*, Computers and Mathematics with Applications, **55**, pp. 1441-1449 (2008).

[G14] B. GRAILLE, *Approximation of mono-dimensional hyperbolic systems: a lattice Boltzmann scheme as a relaxation method*, Journal of Comutational Physics, **266** (3179757), pp. 74-88 (2014).

[QdHL92] Y.H. QIAN, D. D'HUMIERES, and P. LALLEMAND, *Lattice BGK Models for Navier-Stokes Equation*, Europhys. Lett., **17** (6), pp. 479-484 (1992).

INDEX

F

G

I

L

M

N

O

P

R

S

T

U

V