pylbm Documentation

Release 0.3.2

Benjamin Graille, Loïc Gouarin

Jun 24, 2019

CONTENTS

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

pylbm is licensed under the BSD license, enabling reuse with few restrictions.

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

To install pylbm, you have several ways. You can install it using conda

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

or using the last version on Pypi

```
pip install pylbm
```

You can also clone the project

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

and then use the command

```
python setup.py install
```

or if you don't have root privileges

```
python setup.py install --user
```

Once the package is installed you just have to understand how 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)
- the boundary conditions (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

ONE

DOCUMENTATION FOR USERS

1.1 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/

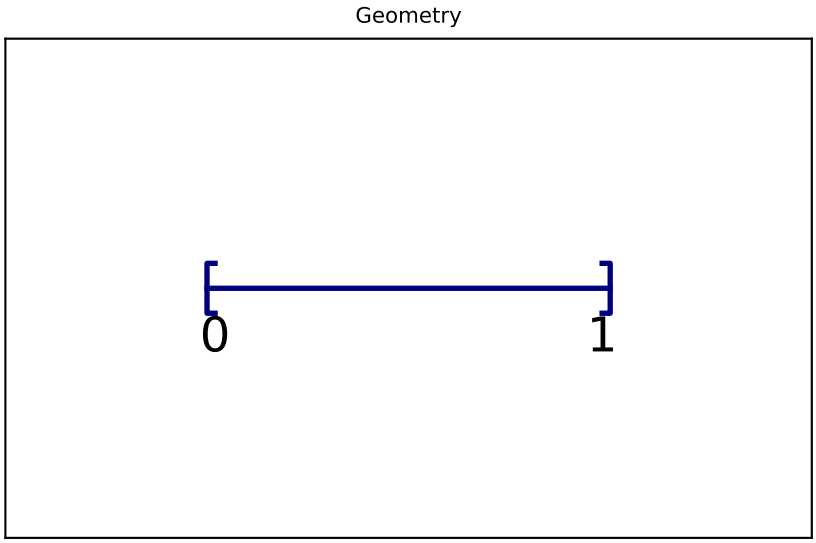
1.1.1 Examples in 1D

`script`

The segment $[0, 1]$

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

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
# License: BSD 3 clause

"""
Example of a 1D geometry: the segment [0,1]
"""
import pylbm
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.

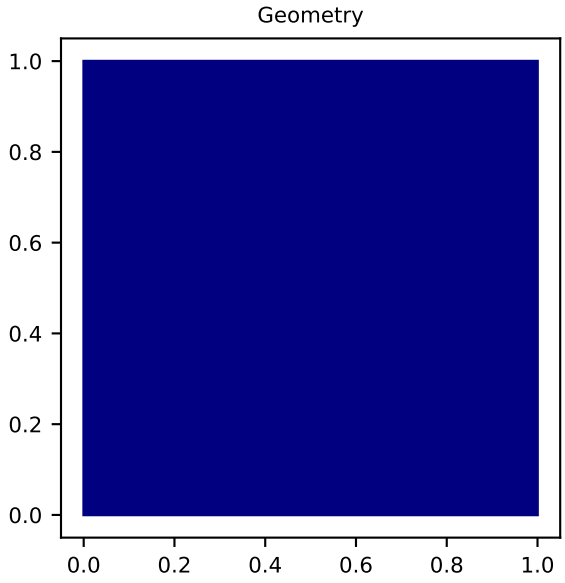
1.1.2 Examples in 2D

`script`

The square $[0, 1]^2$

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

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
# License: BSD 3 clause

"""
Example of a 2D geometry: the square [0,1]x[0,1]
"""
import pylbm
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 ($y = y_{\max}$)

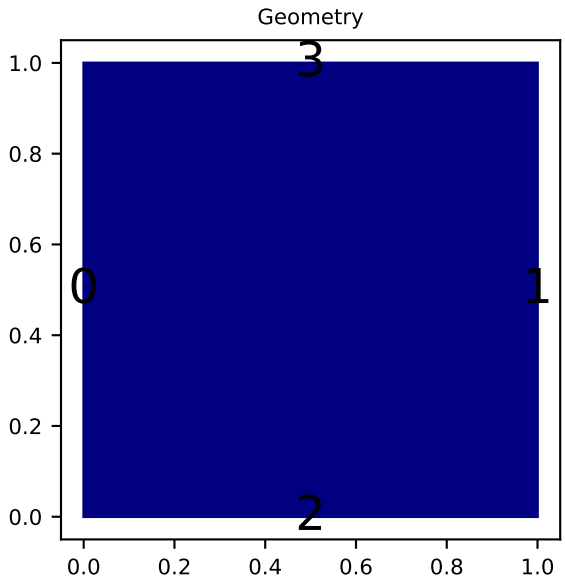
```
d = {'box':{'x': [0, 1], 'y': [0, 1], 'label':[0, 1, 2, 3]}}
g = pylbm.Geometry(d)
g.visualize(viewlabel = True)
```

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
```

(continues on next page)

(continued from previous page)

```
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
# License: BSD 3 clause

"""
Example of a 2D geometry: the square [0,1]x[0,1] with labels
"""
import pylbm
d = {'box':{'x': [0, 1], 'y': [0, 1], 'label':[0, 1, 2, 3]}}
g = pylbm.Geometry(d)
g.visualize(viewlabel = True)
```



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`

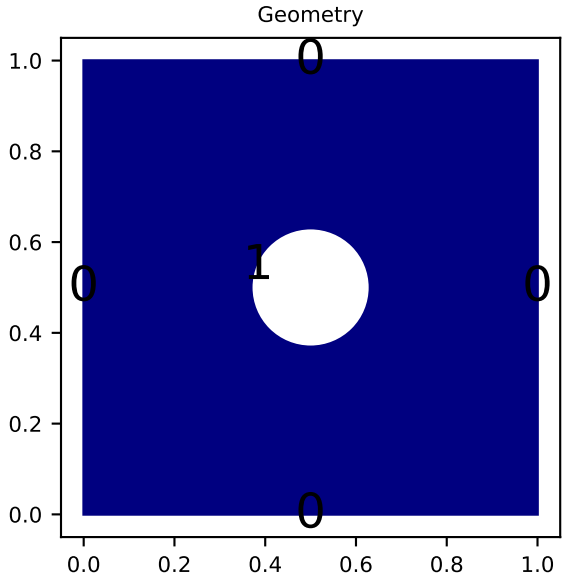
A square with a hole

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)
```

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
# License: BSD 3 clause

"""
Example of a 2D geometry: the square [0,1]x[0,1] with a circular hole
"""
import pylbm
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*

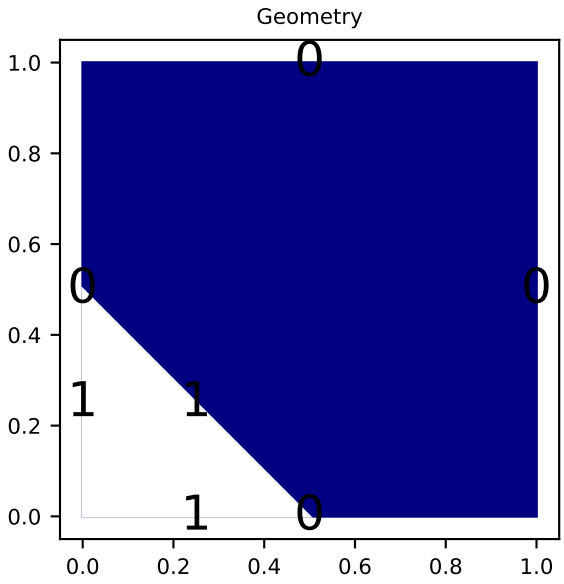
```
d = {'box':{'x': [0, 1], 'y': [0, 1], 'label':0},
      'elements':[pylbm.Triangle((0.,0.), (0.,.5), (.5, 0.), label = 1)],
}
g = pylbm.Geometry(d)
g.visualize(viewlabel=True)
```

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
```

(continues on next page)

(continued from previous page)

```
#
# License: BSD 3 clause

"""
Example of a 2D geometry: the square [0,1]x[0,1] with a triangular hole
"""
import pylbm
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*

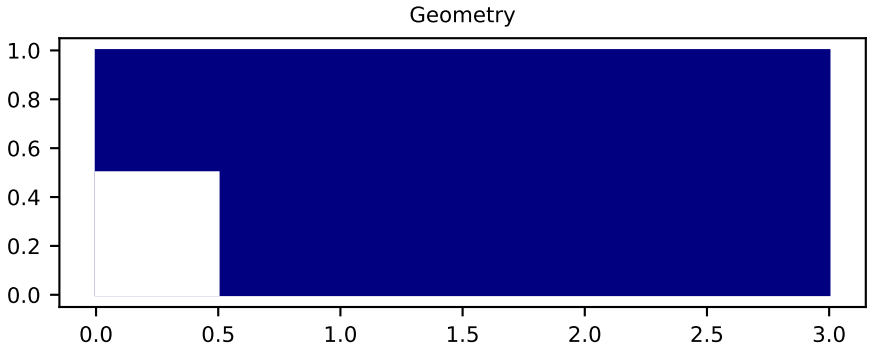
```
d = {'box':{'x': [0, 3], 'y': [0, 1], 'label':[1, 2, 0, 0]},
     'elements':[pylbm.Parallelogram((0.,0.), (.5,0.), (0., .5), label = 0)],
}
g = pylbm.Geometry(d)
g.visualize()
```

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
# License: BSD 3 clause

"""
Example of a 2D geometry: the square [0,1]x[0,1] with a step
```

(continues on next page)

(continued from previous page)

```
"""
import pylbm
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()
```

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
# License: BSD 3 clause

"""
Example of a complex geometry in 2D
"""
import pylbm
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()
# rounded inner angles
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()
```

1.1.3 Examples in 3D

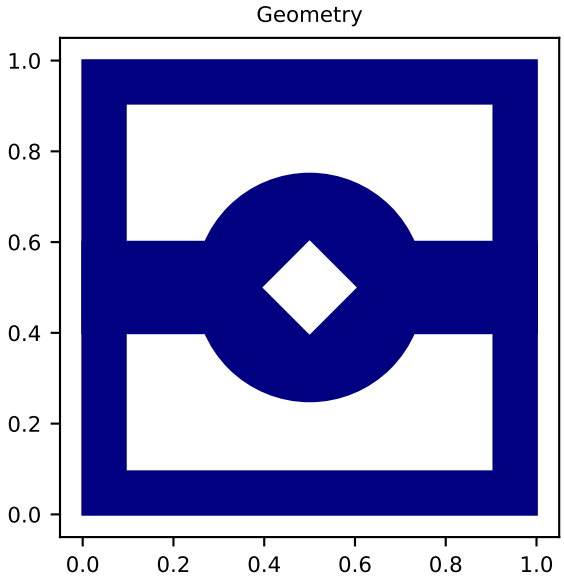
`script`

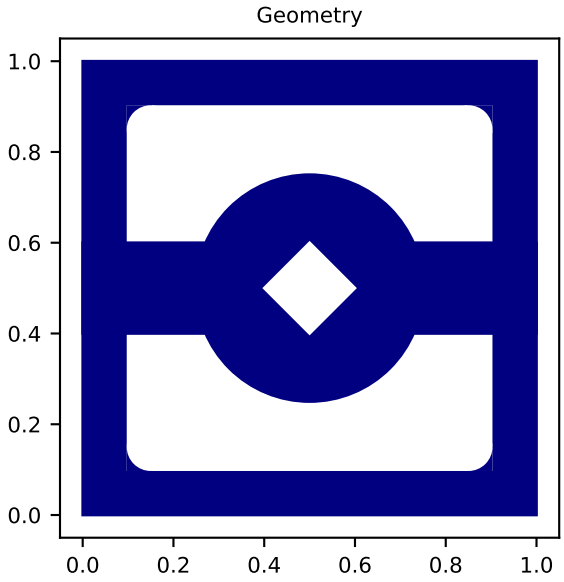
The cube $[0, 1]^3$

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

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
```

(continues on next page)


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


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

(continued from previous page)

```
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
# License: BSD 3 clause

"""
Example of a 3D geometry: the cube [0,1]x[0,1]x[0,1]
"""
from six.moves import range
import pylbm
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 ($y = y_{\min}$)
- 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)
```

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
# License: BSD 3 clause

"""
Example of a 3D geometry: the cube [0,1]x[0,1]x[0,1]
"""
import pylbm
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`.

1.2 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/

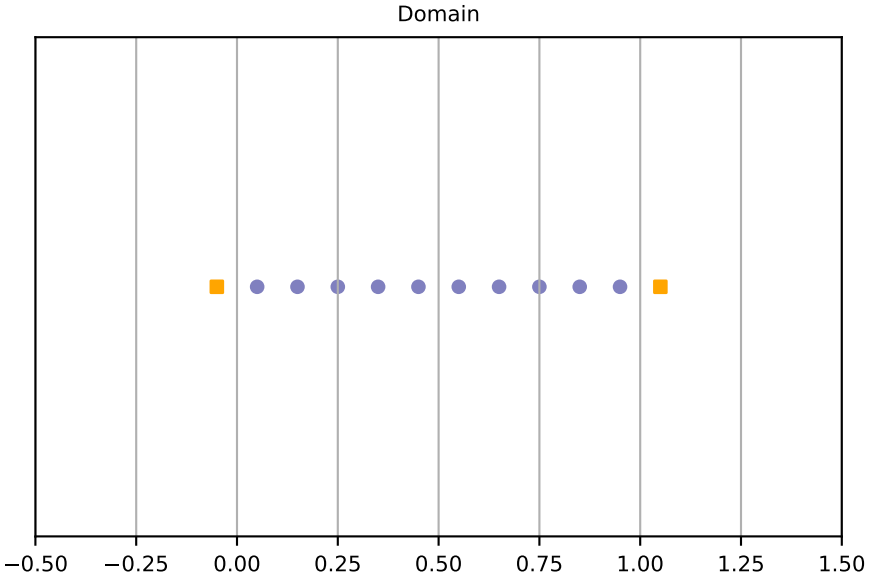
1.2.1 Examples in 1D

`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))}],
}
dom = pylbm.Domain(dico)
dom.visualize()
```

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
# License: BSD 3 clause

"""
Example of a segment in 1D with a D1Q3
"""
from six.moves import range
import pylbm
dico = {
    'box':{'x': [0, 1], 'label':0},
    'space_step':0.1,
    'schemes':[{'velocities':list(range(3))}],
}
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

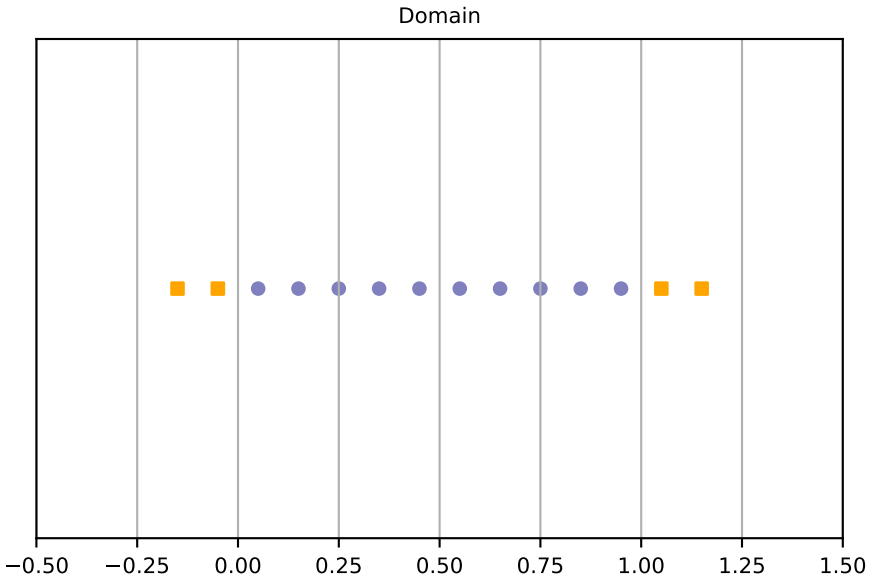
```
dico = {
    'box':{'x': [0, 1], 'label':0},
    'space_step':0.1,
    'schemes':[{'velocities':list(range(5))}],
}
dom = pylbm.Domain(dico)
dom.visualize()
```

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
# License: BSD 3 clause

"""
Example of a segment in 1D with a D1Q5
"""
from six.moves import range
import pylbm
```

(continues on next page)

(continued from previous page)

```
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`.

1.2.2 Examples in 2D

`script`

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

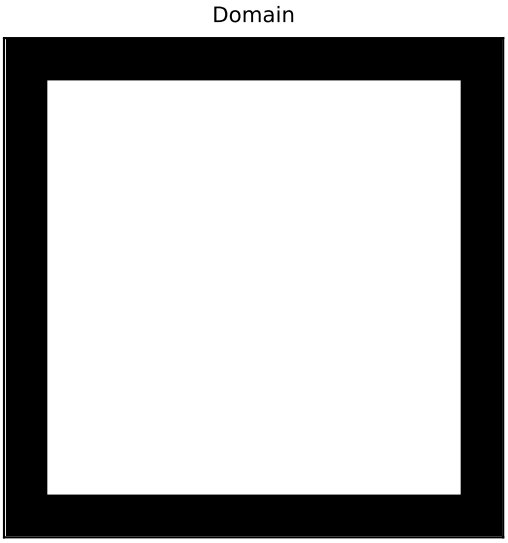
```
dico = {
    'box':{'x': [0, 1], 'y': [0, 1], 'label':0},
    'space_step':0.1,
    'schemes':[{'velocities':list(range(9))}],
}
dom = pylbm.Domain(dico)
```

(continues on next page)

(continued from previous page)

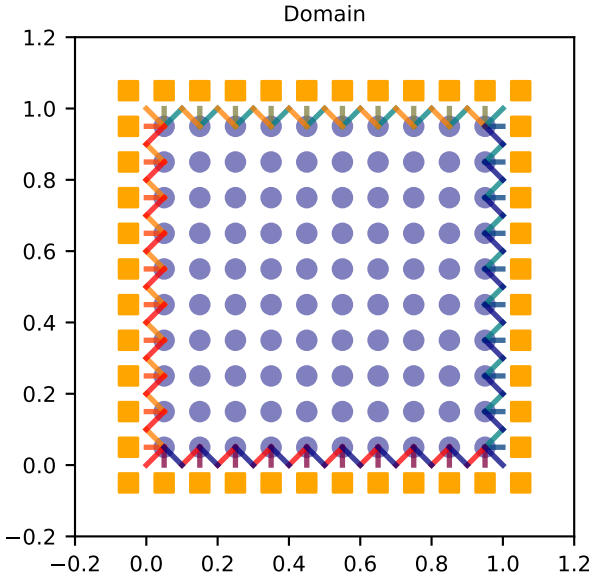
```
dom.visualize()
dom.visualize(view_distance=True)
```

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
# License: BSD 3 clause

"""
Example of a square in 2D with a D2Q9
"""
from six.moves import range
import pylbm
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

$$\begin{aligned} v_0 &= (0,0), \\ v_1 = (1,0), v_2 &= (0,1), v_3 = (-1,0), v_4 = (0,-1), \\ v_5 = (1,1), v_6 &= (-1,1), v_7 = (-1,-1), v_8 = (1,-1). \end{aligned} \tag{1.1}$$



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`

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

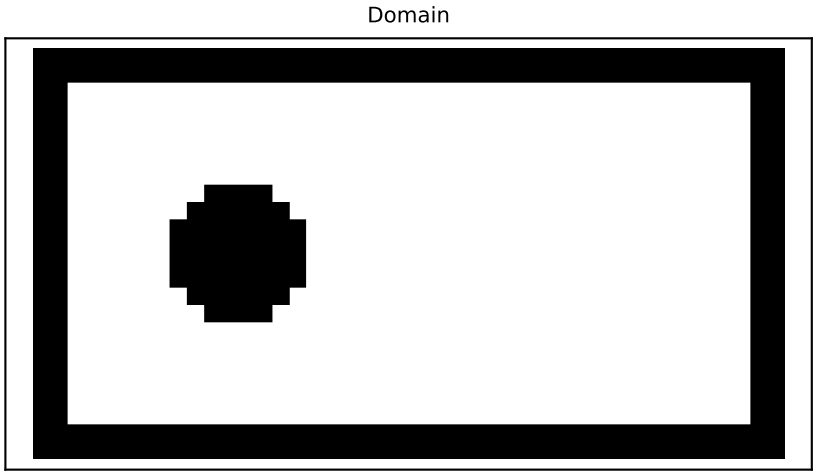
```
dico = {
    'box':{'x': [0, 2], 'y': [0, 1], 'label':0},
    'elements':[pylbm.Circle((0.5,0.5), 0.2)],
    'space_step':0.05,
    'schemes':[{'velocities':list(range(13))}],
}
dom = pylbm.Domain(dico)
dom.visualize()
dom.visualize(view_distance=True)
```

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
```

(continues on next page)

(continued from previous page)

```
# License: BSD 3 clause

"""
Example of a square in 2D with a circular hole with a D2Q13
"""
from six.moves import range
import pylbm
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)
```



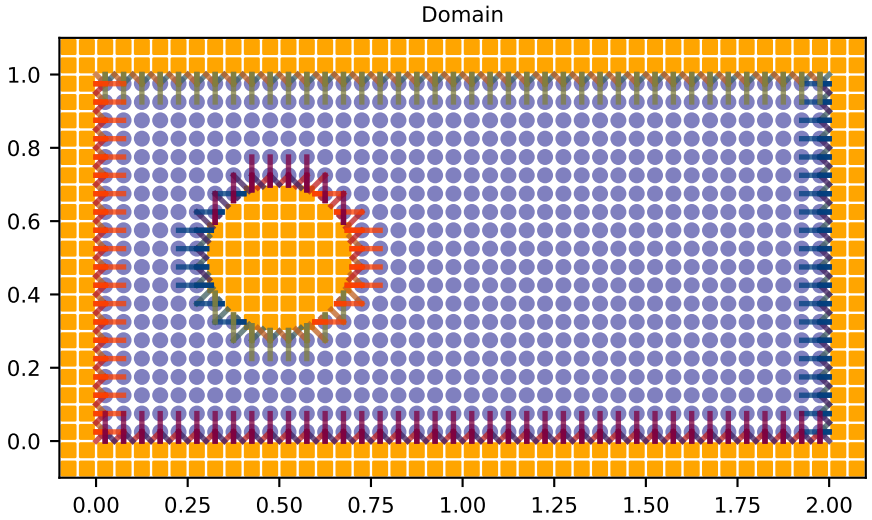
script

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))}],
```

(continues on next page)

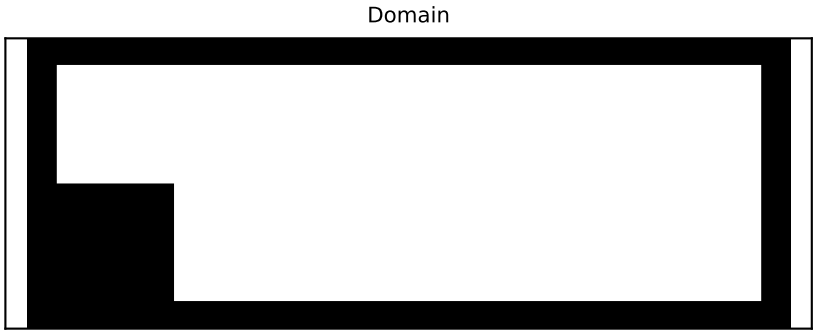


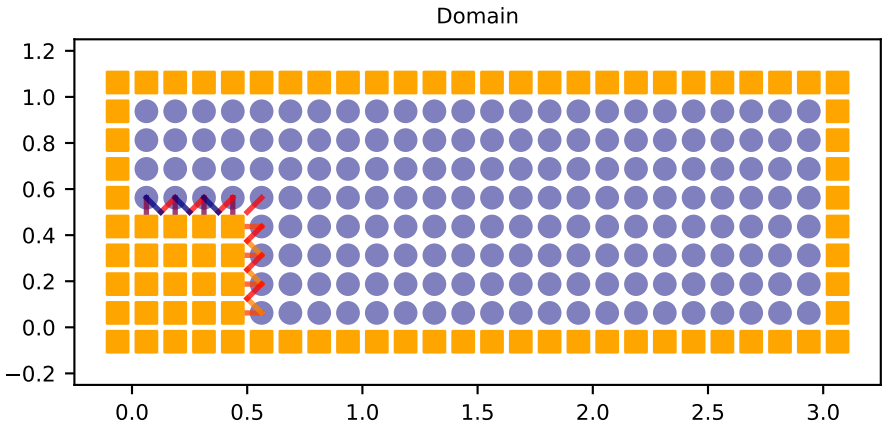
(continued from previous page)

```
}
dom = pylbm.Domain(dico)
dom.visualize()
dom.visualize(view_distance=True, label=1)
```

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
# License: BSD 3 clause

"""
Example of the backward facing step in 2D with a D2Q9
"""
from six.moves import range
import pylbm
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)
dom.visualize()
dom.visualize(view_distance=True, label=1)
```

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


Domain


Domain
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0.0
0.5
1.0
1.5
2.0
2.5
3.0

1.2.3 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)
```

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
#
# License: BSD 3 clause

"""
Example of the cube in 3D with a D3Q19
"""
from six.moves import range
import pylbm
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`.

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)
```

```
# Authors:
#     Loic Gouarin <loic.gouarin@math.u-psud.fr>
#     Benjamin Graille <benjamin.graille@math.u-psud.fr>
```

(continues on next page)

(continued from previous page)

```
#
# License: BSD 3 clause

"""
Example of the cube in 3D with a D3Q19
"""
from six.moves import range
import pylbm
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)
```

1.3 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.

1.3.1 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.,
```

(continues on next page)

(continued from previous page)

```
    '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,
```

(continues on next page)

(continued from previous page)

```
    '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
```

(continues on next page)

(continued from previous page)

```
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,
            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)),
```

(continues on next page)

(continued from previous page)

```
        '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

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

1.3.2 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],
            '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

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

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]
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).

1.4 The Boundary Conditions

The simulations are performed in a bounded domain with optional obstacles. Boundary conditions have then to be imposed on all the bounds. With pylbm, the user can use the classical boundary conditions (classical for the lattice Boltzmann method) that are already implemented or implement his own conditions.

Note that periodical boundary conditions are used as default conditions. The corresponding label is -1.

For a lattice Boltzmann method, we have to impose the incoming distribution functions on nodes outside the domain. We describe

- first, how the bounce back, the anti bounce back, and the Neumann conditions can be used,
- second, how personal boundary conditions can be implemented.

1.4.1 The classical conditions

The bounce back and anti bounce back conditions

The bounce back condition (*resp.* anti bounce back) is used to impose the odd moments (*resp.* even moments) on the bounds.

The Neumann conditions

1.4.2 How to implement new conditions

1.5 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

1.5.1 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.]]
```

(continues on next page)

(continued from previous page)

```
 [[ 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.]
```

(continues on next page)

(continued from previous page)

```
  [ 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.

1.5.2 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]}, [0, 9])
```

[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.]])
```

[ ]:
```
```

1.6 Tutorial

1.6.1 Transport in 1D

In this tutorial, we test the most simple lattice Boltzmann scheme $\mathrm{D_1Q_2}$ 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.

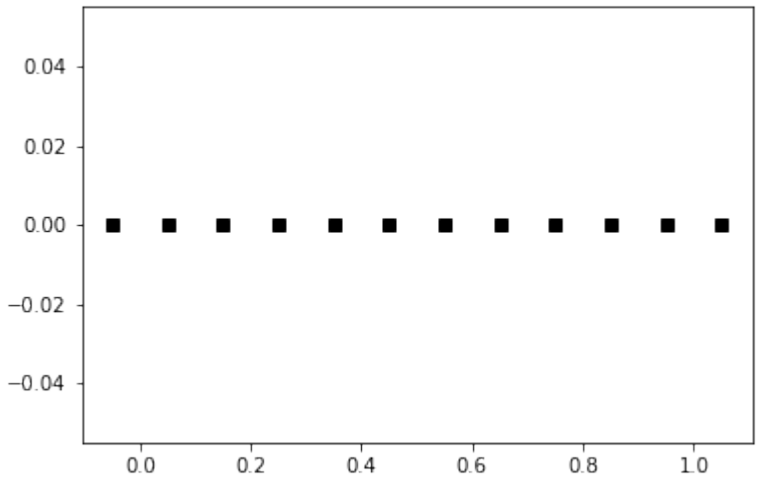
The numerical simulation of this equation by a lattice Boltzmann scheme consists in the approximation of the solution on discret points of $(0, 1)$ at discret instants.

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):
```

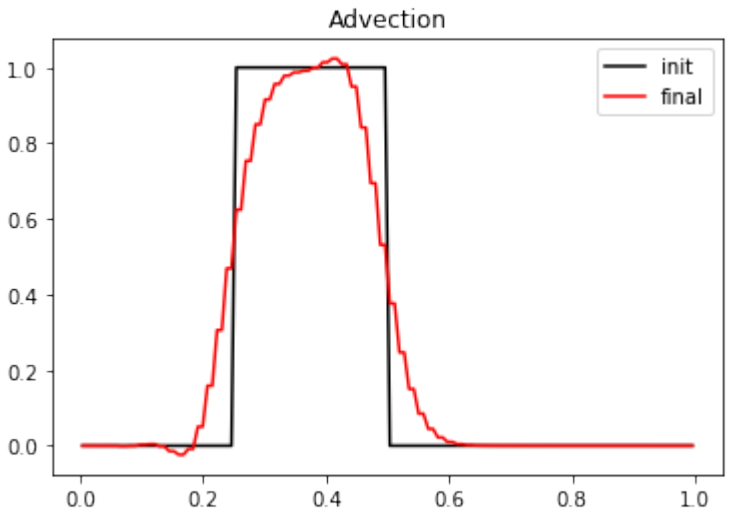
(continues on next page)

(continued from previous page)

```
    #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')
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 $m_1^e = {m_0}^2/2$.

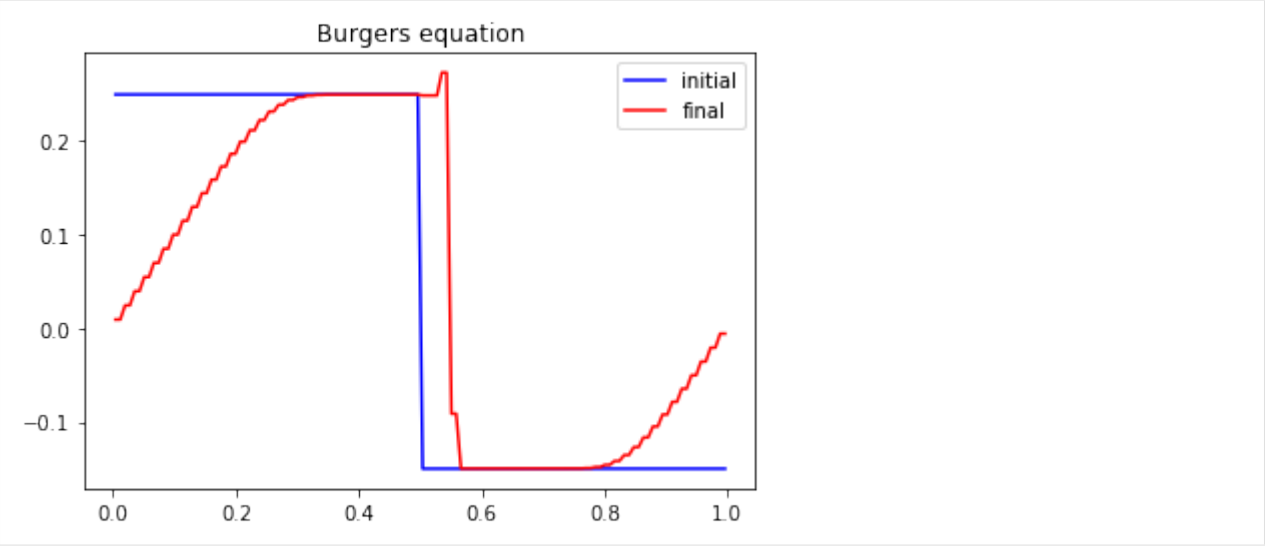
More generaly, the simulated equation is into the conservative form

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

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)

# 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.

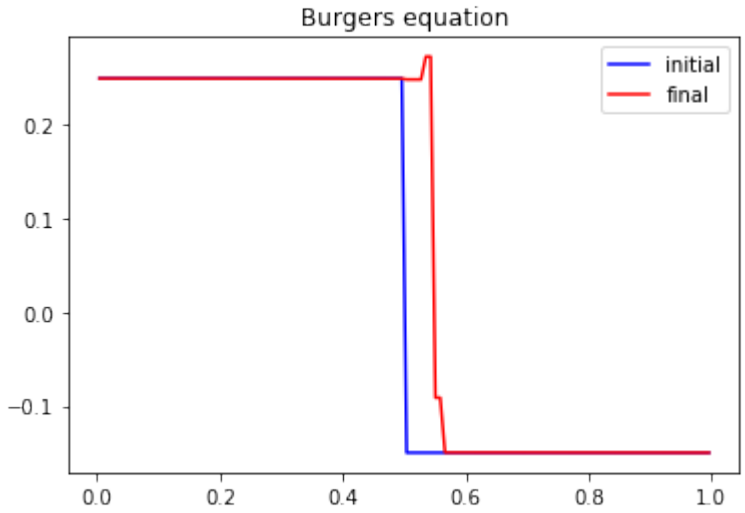
We modify the previous script to take into account these new boundary conditions.

[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
```

(continues on next page)

(continued from previous page)

```
    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()
```



1.6.2 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 $\mathrm{D}_1\mathrm{Q}_3$

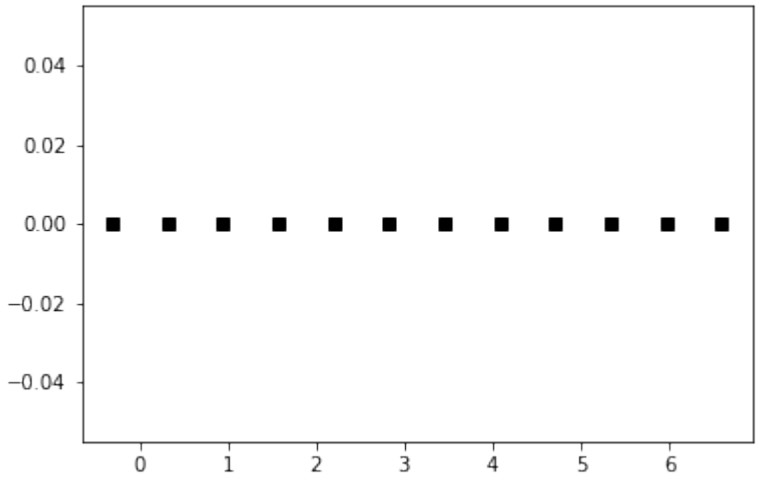
The numerical simulation of this equation by a lattice Boltzmann scheme consists in the approximation of the solution on discret points of $(0, 2\pi)$ at discret instants.

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
```

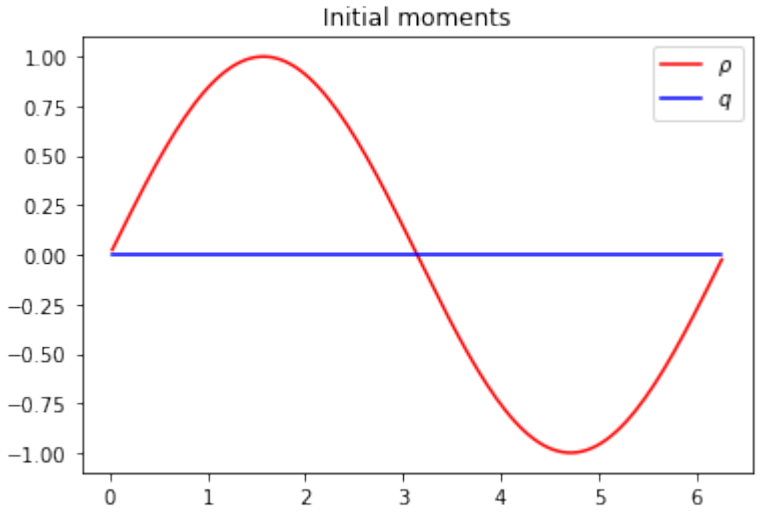
Periodic boundary conditions

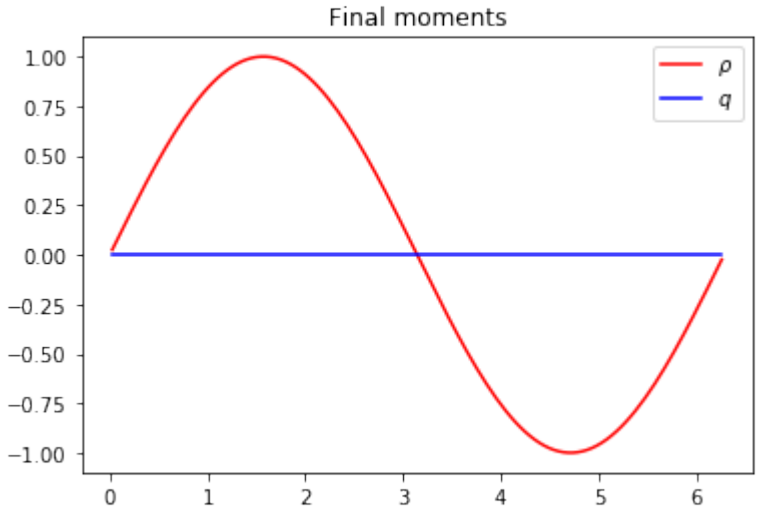
We implement the four elementary functions f2m, relaxation, m2f, and transport. In the transport function, the boundary conditions should be implemented: we will use periodic conditions by copying the informations in the phantom cells.

[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
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 $f_2(x_{N+1}) = -f_1(x_N)$.

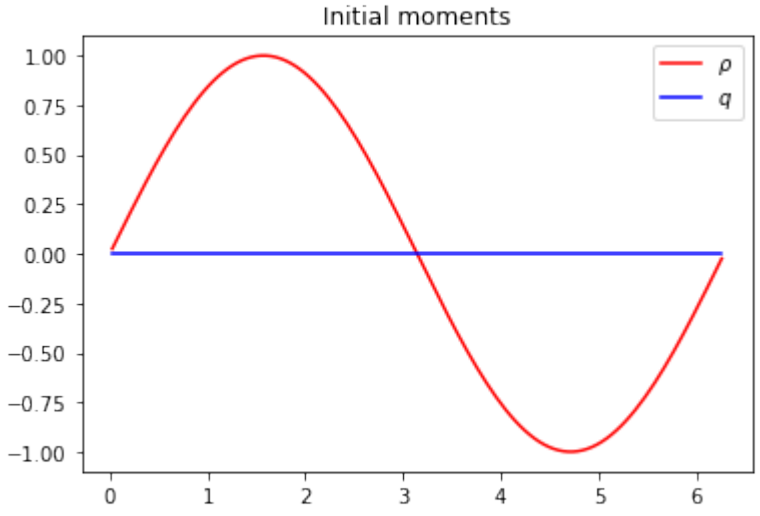
We modify the transport function to impose anti bounce back conditions. We can compare the solutions obtained with the two different boundary conditions.

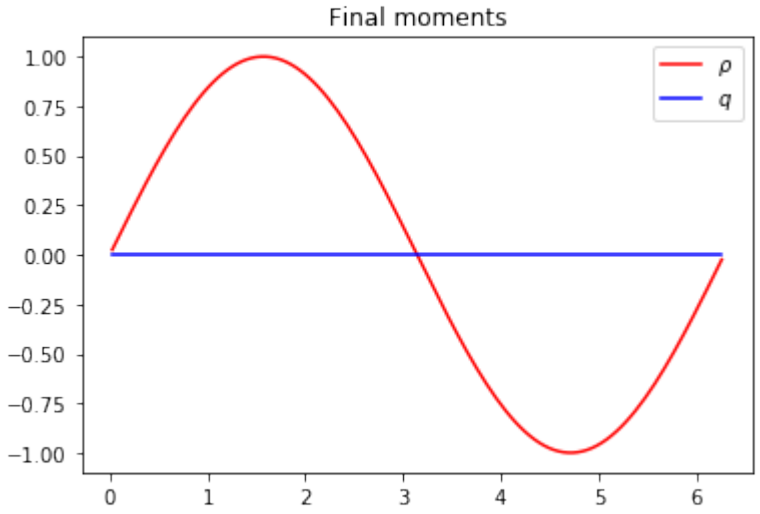
[7]:
```
def transport(f0, f1, f2):
    # anti bounce back boundary conditions
    f1[-1] = -f2[-2]
    f2[0] = -f1[1]
    # transport
    f1[1:-1] = f1[2:]
    f2[1:-1] = f2[:-2]
```

(continues on next page)

(continued from previous page)

```
# 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)
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()
```





[ ]:

1.6.3 The heat equation in 1D

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

[8]:
```
from __future__ import print_function, division
from six.moves import range
%matplotlib inline
```

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

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 $\mathrm{D}_1\mathrm{Q}_3$ 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,
- 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:

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

- 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

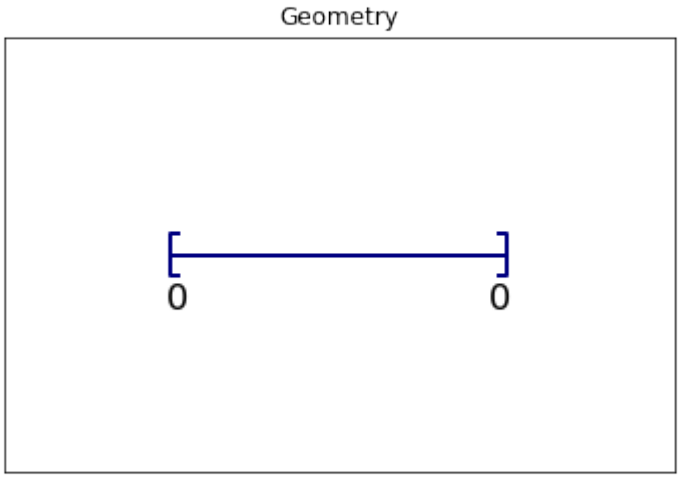
pylbm uses Python dictionary to describe the simulation. In the following, we will build this dictionary step by step.

The geometry

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

[9]:
```
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 informations
         spatial dimension: 1
         bounds of the box:
[[0. 1.]]
```

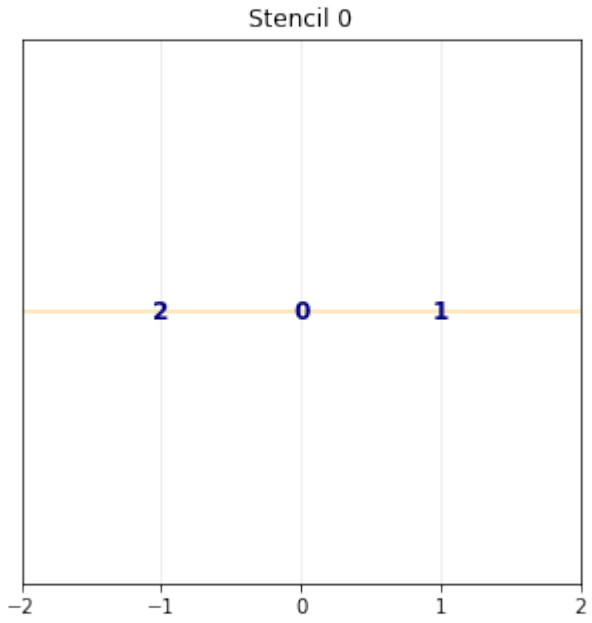


The stencil

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 $[0, 1, 2]$.

[10]:
```
dico_sten = {
    'dim':1,
    'schemes':[{'velocities':list(range(3))}],
}
sten = pylbm.Stencil(dico_sten)
print(sten)
sten.visualize()
```

```
Stencil informations
         * spatial dimension: 1
         * maximal velocity in each direction: [1]
         * minimal velocity in each direction: [-1]
         * Informations for each elementary stencil:
               stencil 0
                - number of velocities:  3
                - velocities: (0: 0), (1: 1), (2: -1),
```



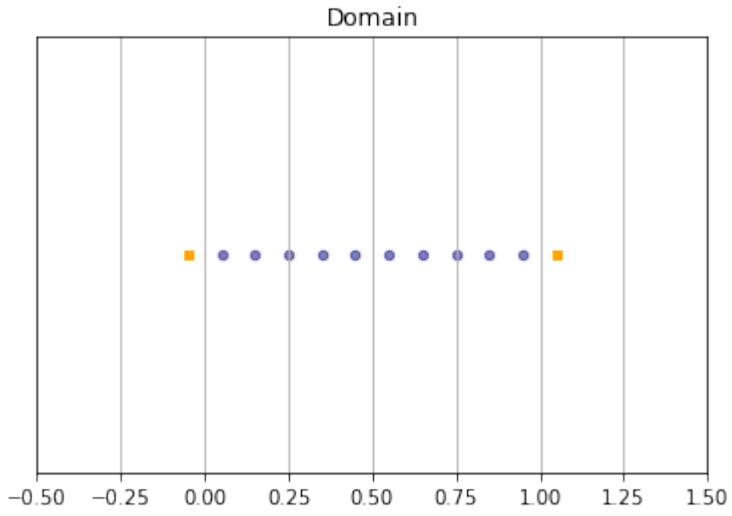
The domain

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.

[11]:
```
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 informations
         spatial dimension: 1
         space step: dx= 1.000e-01
```



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

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

[12]:
```
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,
```

(continues on next page)

(continued from previous page)

```
    '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.,))},
        }
    ],
}

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

```
[0] WARNING  pylbm.scheme in function __init__ line 194
The value 'space_step' is not given or wrong.
The scheme takes default value: dx = 1.
```

```
Scheme informations
         spatial dimension: dim=1
         number of schemes: nscheme=1
         number of velocities:
    Stencil.nv[0]=3
         velocities value:
    v[0] = (0: 0), (1: 1), (2: -1),
         polynomials:
    P[0] = 1, X, X**2/2,
         equilibria:
    EQ[0] = u, 0.0, 0.5*u,
         relaxation parameters:
    s[0] = 0.0, 0.666666666666667, 1.00000000000000,
         moments matrices
M      = Matrix([[1, 1, 1], [0, 10.0000000000000, -10.0000000000000], [0, 50.
↪0000000000000, 50.0000000000000]])
M^(-1) = Matrix([[1.00000000000000, 0, -0.0200000000000000], [0, 0.0500000000000000,␣
↪0.0100000000000000], [0, -0.0500000000000000, 0.0100000000000000]])
```

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).

[13]:
```
dico = {
    'box':{'x':[xmin, xmax], 'label':0},
    'space_step':dx,
    'scheme_velocity':la,
    'schemes':[
        {
            'velocities':list(range(3)),
            'conserved_moments':u,
```

(continues on next page)

(continued from previous page)

```
            '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.anti_bounce_back,}, 'value':None},
    },
    'generator': 'numpy'
}

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

```
Simulation informations:
Domain informations
         spatial dimension: 1
         space step: dx= 1.000e-01
Scheme informations
         spatial dimension: dim=1
         number of schemes: nscheme=1
         number of velocities:
    Stencil.nv[0]=3
         velocities value:
    v[0] = (0: 0), (1: 1), (2: -1),
         polynomials:
    P[0] = 1, X, X**2/2,
         equilibria:
    EQ[0] = u, 0.0, 0.5*u,
         relaxation parameters:
    s[0] = 0.0, 0.666666666666667, 1.00000000000000,
         moments matrices
M      = Matrix([[1, 1, 1], [0, 10.0000000000000, -10.0000000000000], [0, 50.
↪0000000000000, 50.0000000000000]])
M^(-1) = Matrix([[1.00000000000000, 0, -0.0200000000000000], [0, 0.0500000000000000,␣
↪0.0100000000000000], [0, -0.0500000000000000, 0.0100000000000000]])
```

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.

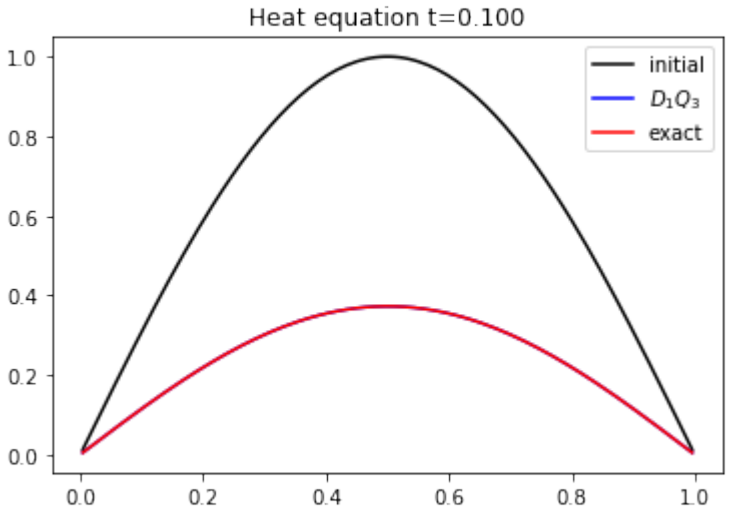
[14]:
```
import numpy as np
import sympy as sp
import pylab as plt
import pylbm

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)
```

(continues on next page)

(continued from previous page)

```
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.anti_bounce_back,}, 'value':None},
    },
    '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()
```

[14]:
```
<matplotlib.legend.Legend at 0x7f980defffd0>
```



[ ]:

[1]:
```
from __future__ import print_function, division
from six.moves import range
```

[2]:
```
%matplotlib inline
```

1.6.4 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.

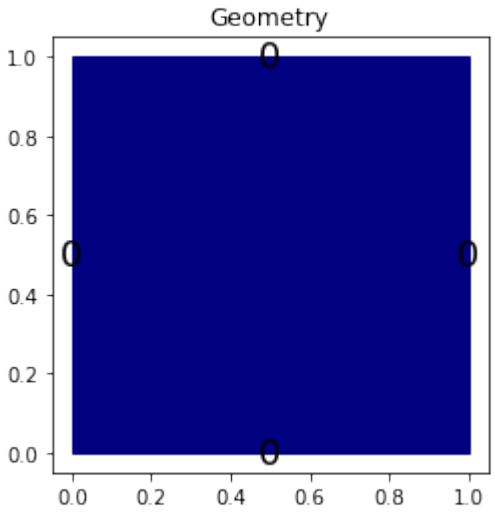
The geometry

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

[3]:
```
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)
```

```
/home/loic/miniconda3/envs/pylbm/lib/python3.6/site-packages/h5py/__init__.py:36:␣
↪FutureWarning: Conversion of the second argument of issubdtype from `float` to `np.
↪floating` is deprecated. In future, it will be treated as `np.float64 == np.
↪dtype(float).type`.
  from ._conv import register_converters as _register_converters
```

```
Geometry informations
         spatial dimension: 2
         bounds of the box:
[[0. 1.]
 [0. 1.]]
```

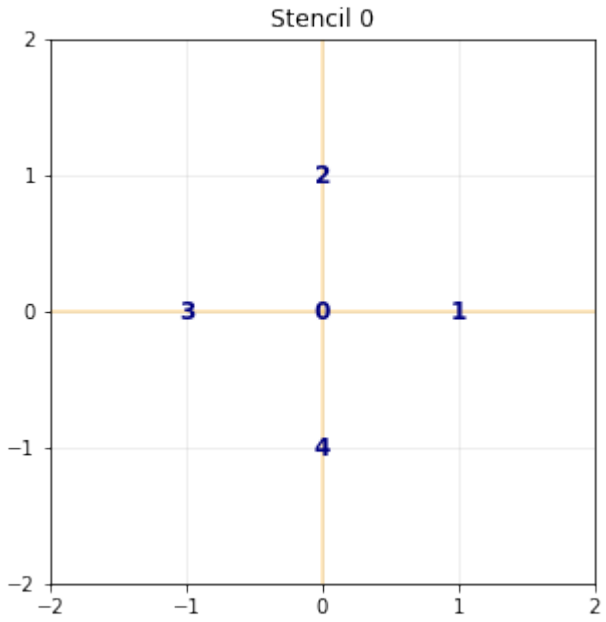


The stencil

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 $[0,1,2,3,4]$.

[4]:
```
dico_sten = {
    'dim':2,
    'schemes':[{'velocities':list(range(5))}],
}
sten = pylbm.Stencil(dico_sten)
print(sten)
sten.visualize()
```

```
Stencil informations
         * spatial dimension: 2
         * maximal velocity in each direction: [1 1]
         * minimal velocity in each direction: [-1 -1]
         * Informations 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.

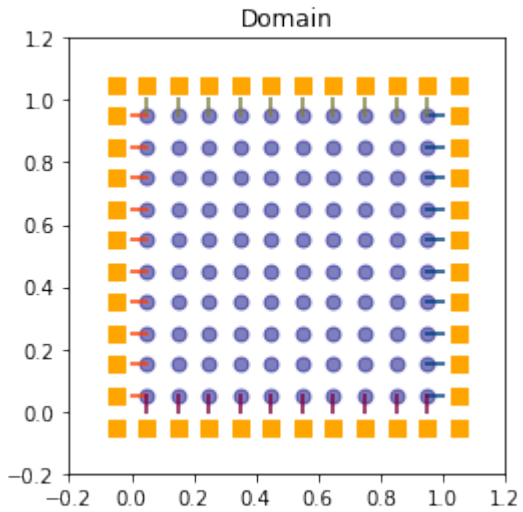
[5]:
```
N = 10
dx = (xmax-xmin)/N
dico_dom = {
    'box': {'x': [xmin, xmax], 'y':[ymin, ymax], 'label':0},
    'space_step':dx,
    'schemes':[
        {
            'velocities':list(range(5)),
        }
    ],
```

(continues on next page)

(continued from previous page)

```
}
dom = pylbm.Domain(dico_dom)
print(dom)
dom.visualize(view_distance=True)
```

```
Domain informations
        spatial dimension: 2
        space step: dx= 1.000e-01
```



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

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

[6]:
```
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)
```

(continues on next page)

(continued from previous page)

```
# 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],
            'init':{u:(solution, (0.,))},
        }
    ],
    'parameters':{LA: la},
}

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

```
[0] WARNING  pylbm.scheme in function __init__ line 194
The value 'space_step' is not given or wrong.
The scheme takes default value: dx = 1.
```

```
Scheme informations
         spatial dimension: dim=2
         number of schemes: nscheme=1
         number of velocities:
    Stencil.nv[0]=5
         velocities value:
    v[0] = (0: 0, 0), (1: 1, 0), (2: 0, 1), (3: -1, 0), (4: 0, -1),
         polynomials:
    P[0] = 1, X/LA, Y/LA, (X**2 + Y**2)/(2*LA**2), (X**2 - Y**2)/(2*LA**2),
         equilibria:
    EQ[0] = u, 0.0, 0.0, 0.5*u, 0.0,
         relaxation parameters:
    s[0] = 0.0, 0.400000000000000, 0.400000000000000, 1.00000000000000, 1.
↪00000000000000,
         moments matrices
M      = Matrix([[1, 1, 1, 1, 1], [0, 10.0/LA, 0, -10.0/LA, 0], [0, 0, 10.0/LA, 0,␣
↪-10.0/LA], [0, 50.0/LA**2, 50.0/LA**2, 50.0/LA**2, 50.0/LA**2], [0, 50.0/LA**2, -50.
↪0/LA**2, 50.0/LA**2, -50.0/LA**2]])
M^(-1) = Matrix([[1.00000000000000, 0, 0, -0.02*LA**2, 0], [0, 0.05*LA, 0, 0.
↪005*LA**2, 0.005*LA**2], [0, 0, 0.05*LA, 0.005*LA**2, -0.005*LA**2], [0, -0.05*LA,␣
↪0, 0.005*LA**2, 0.005*LA**2], [0, 0, -0.05*LA, 0.005*LA**2, -0.005*LA**2]])
```

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).

[7]:
```
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.anti_bounce_back,}, 'value':None},
    },
    'parameters':{LA: la},
}

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

```
Simulation informations:
Domain informations
         spatial dimension: 2
         space step: dx= 1.000e-01
Scheme informations
         spatial dimension: dim=2
         number of schemes: nscheme=1
         number of velocities:
    Stencil.nv[0]=5
         velocities value:
    v[0] = (0: 0, 0), (1: 1, 0), (2: 0, 1), (3: -1, 0), (4: 0, -1),
         polynomials:
    P[0] = 1, X/LA, Y/LA, (X**2 + Y**2)/(2*LA**2), (X**2 - Y**2)/(2*LA**2),
         equilibria:
    EQ[0] = u, 0.0, 0.0, 0.5*u, 0.0,
         relaxation parameters:
    s[0] = 0.0, 0.400000000000000, 0.400000000000000, 1.00000000000000, 1.
↪00000000000000,
         moments matrices
M      = Matrix([[1, 1, 1, 1, 1], [0, 10.0/LA, 0, -10.0/LA, 0], [0, 0, 10.0/LA, 0,␣
↪-10.0/LA], [0, 50.0/LA**2, 50.0/LA**2, 50.0/LA**2, 50.0/LA**2], [0, 50.0/LA**2, -50.
↪0/LA**2, 50.0/LA**2, -50.0/LA**2]])
M^(-1) = Matrix([[1.00000000000000, 0, 0, -0.02*LA**2, 0], [0, 0.05*LA, 0, 0.
↪005*LA**2, 0.005*LA**2], [0, 0, 0.05*LA, 0.005*LA**2, -0.005*LA**2], [0, -0.05*LA,␣
↪0, 0.005*LA**2, 0.005*LA**2], [0, 0, -0.05*LA, 0.005*LA**2, -0.005*LA**2]])
```

Run a simulation

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

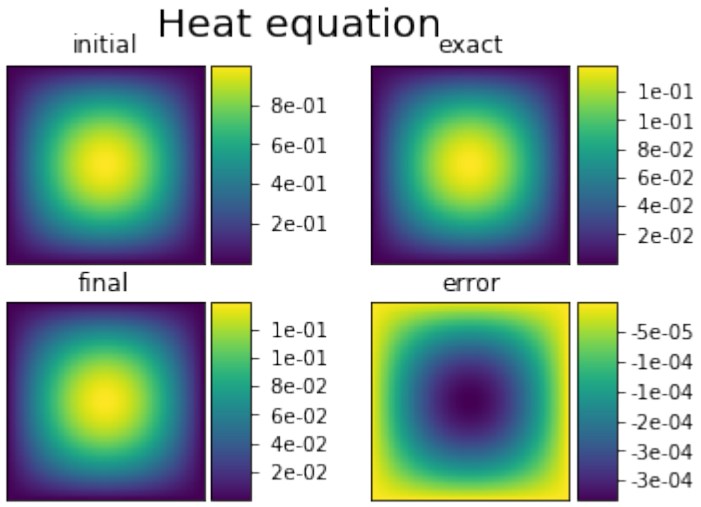
We compute the solution of the heat equation at $t = 0.1$. On the same graphic, we plot the initial condition, the exact solution and the numerical solution.

[8]:
```
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.anti_bounce_back,}, 'value':None},
    },
    'generator': 'cython',
```

(continues on next page)

(continued from previous page)

```
    'parameters':{LA: la},
}

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()
```



[ ]:

1.6.5 Poiseuille flow

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

[1]:
```
from __future__ import print_function, division
from six.moves import range
%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 $p = \rho\lambda^2/3$.

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.
```

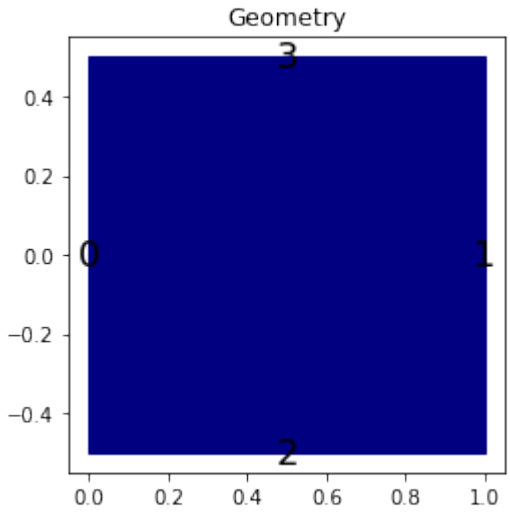
(continues on next page)

(continued from previous page)

```
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)
```

```
/home/loic/miniconda3/envs/pylbm/lib/python3.6/site-packages/h5py/__init__.py:36:␣
↪FutureWarning: Conversion of the second argument of issubdtype from `float` to `np.
↪floating` is deprecated. In future, it will be treated as `np.float64 == np.
↪dtype(float).type`.
  from ._conv import register_converters as _register_converters
```

```
Geometry informations
         spatial dimension: 2
         bounds of the box:
[[ 0.   1. ]
 [-0.5  0.5]]
```



The stencil

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

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

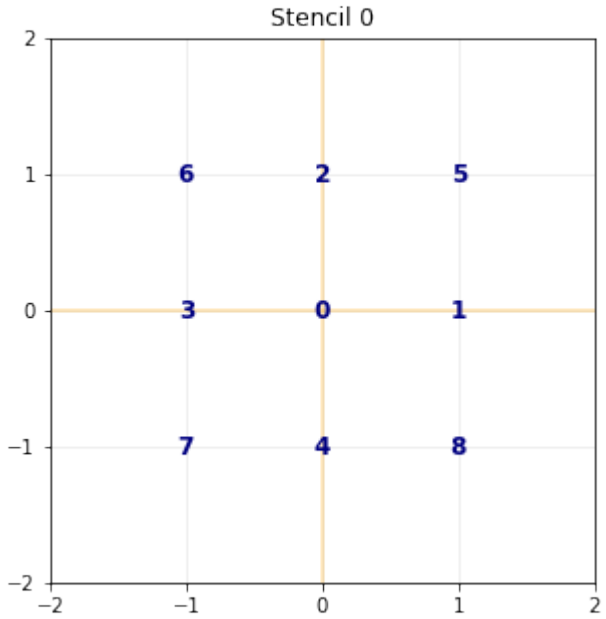
[3]:
```
dico_sten = {
    'dim':2,
    'schemes':[{'velocities':list(range(9))}],
}
sten = pylbm.Stencil(dico_sten)
print(sten)
sten.visualize()
```

```
Stencil informations
         * spatial dimension: 2
```

(continues on next page)

(continued from previous page)

```
        * maximal velocity in each direction: [1 1]
        * minimal velocity in each direction: [-1 -1]
        * Informations 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),
```

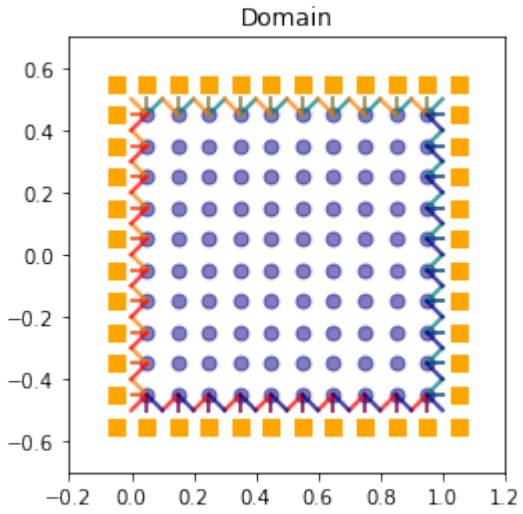


The domain

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 informations
        spatial dimension: 2
        space step: dx= 1.000e-01
```



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
```

(continues on next page)

(continued from previous page)

```
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
            ],
            'init':{rho:rhoo, qx:0., qy:0.},
        },
    ],
    'generator': 'cython',
}
sch = pylbm.Scheme(dico_sch)
print(sch)
```

```
Scheme informations
         spatial dimension: dim=2
         number of schemes: nscheme=1
         number of velocities:
    Stencil.nv[0]=9
         velocities value:
    v[0] = (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:
    P[0] = 1, LA*X, LA*Y, 3*X**2 + 3*Y**2 - 4, -21*X**2/2 - 21*Y**2/2 + 9*(X**2 +␣
↪Y**2)**2/2 + 4, 3*X*(X**2 + Y**2) - 5*X, 3*Y*(X**2 + Y**2) - 5*Y, X**2 - Y**2, X*Y
```

(continues on next page)

(continued from previous page)

```
        equilibria:
    EQ[0] = rho, qx, qy, -2*rho + 3.0*qx**2/LA**2 + 3.0*qy**2/LA**2, rho - 3.0*qx**2/
↪LA**2 - 3.0*qy**2/LA**2, -qx/LA, -qy/LA, 1.0*qx**2/LA**2 - 1.0*qy**2/LA**2, 1.
↪0*qx*qy/LA**2,
        relaxation parameters:
    s[0] = 0.0, 0.0, 0.0, 1.13122171945701, 1.13122171945701, 0.0257069408740360, 0.
↪0257069408740360, 0.0257069408740360, 0.0257069408740360,
        moments matrices
M      = Matrix([[1, 1, 1, 1, 1, 1, 1, 1, 1], [0, 1.0*LA, 0, -1.0*LA, 0, 1.0*LA, -1.
↪0*LA, -1.0*LA, 1.0*LA], [0, 0, 1.0*LA, 0, -1.0*LA, 1.0*LA, 1.0*LA, -1.0*LA, -1.
↪0*LA], [-4, -1.00000000000000, -1.00000000000000, -1.00000000000000, -1.
↪00000000000000, 2.00000000000000, 2.00000000000000, 2.00000000000000, 2.
↪00000000000000], [4, -2.00000000000000, -2.00000000000000, -2.00000000000000, -2.
↪00000000000000, 1.00000000000000, 1.00000000000000, 1.00000000000000, 1.
↪00000000000000], [0, -2.00000000000000, 0, 2.00000000000000, 0, 1.00000000000000,␣
↪-1.00000000000000, -1.00000000000000, 1.00000000000000], [0, 0, -2.00000000000000,␣
↪0, 2.00000000000000, 1.00000000000000, 1.00000000000000, -1.00000000000000, -1.
↪00000000000000], [0, 1.00000000000000, -1.00000000000000, 1.00000000000000, -1.
↪00000000000000, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1.00000000000000, -1.00000000000000, 1.
↪00000000000000, -1.000000000000000]])
M^(-1) = Matrix([[0.111111111111111, 0, 0, -0.111111111111111, 0.111111111111111, 0,␣
↪0, 0, 0], [0.111111111111111, 0.166666666666667/LA, 0, -0.0277777777777778, -0.
↪0555555555555556, -0.166666666666667, 0, 0.250000000000000, 0], [0.111111111111111,␣
↪0, 0.166666666666667/LA, -0.0277777777777778, -0.0555555555555556, 0, -0.
↪166666666666667, -0.250000000000000, 0], [0.111111111111111, -0.166666666666667/LA,␣
↪0, -0.0277777777777778, -0.0555555555555556, 0.166666666666667, 0, 0.
↪250000000000000, 0], [0.111111111111111, 0, -0.166666666666667/LA, -0.
↪0277777777777778, -0.0555555555555556, 0, 0.166666666666667, -0.250000000000000, 0],
↪ [0.111111111111111, 0.166666666666667/LA, 0.166666666666667/LA, 0.0555555555555556,
↪ 0.0277777777777778, 0.0833333333333333, 0.0833333333333333, 0, 0.250000000000000],␣
↪[0.111111111111111, -0.166666666666667/LA, 0.166666666666667/LA, 0.0555555555555556,
↪ 0.0277777777777778, -0.0833333333333333, 0.0833333333333333, 0, -0.
↪250000000000000], [0.111111111111111, -0.166666666666667/LA, -0.166666666666667/LA,␣
↪0.0555555555555556, 0.0277777777777778, -0.0833333333333333, -0.0833333333333333, 0,
↪ 0.250000000000000], [0.111111111111111, 0.166666666666667/LA, -0.166666666666667/
↪LA, 0.0555555555555556, 0.0277777777777778, 0.0833333333333333, -0.0833333333333333,
↪ 0, -0.250000000000000]])
```

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)
    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
```

(continues on next page)

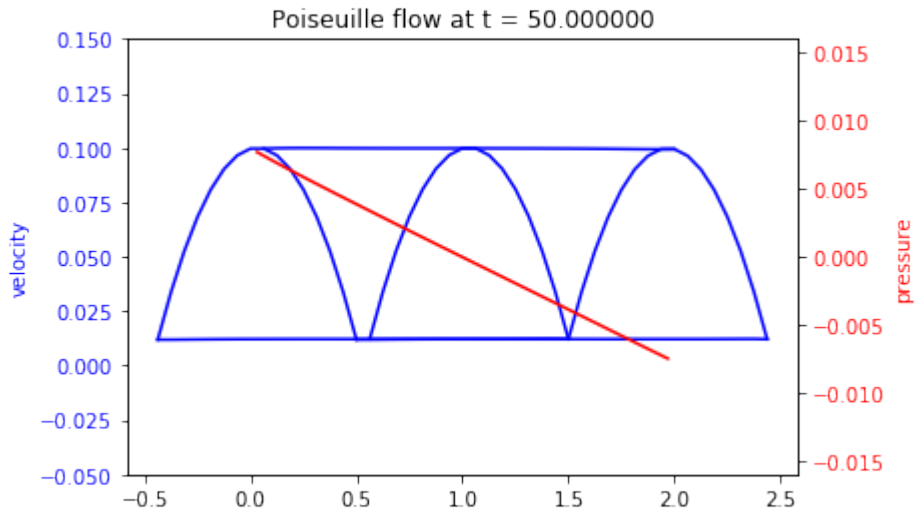
(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
print("Numerical pressure gradient: {0:10.3e}".format(num_pressure_gradient))
```




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

[ ]:

1.6.6 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]:
```
from __future__ import print_function, division
from six.moves import range
%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

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
```

(continues on next page)

(continued from previous page)

```
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':[
        {
            '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.Bouzidi_bounce_back}, 'value':None},
        1:{'method':{0: pylbm.bc.Bouzidi_bounce_back}, 'value':bc}
    },
    'generator': 'cython',
}
```

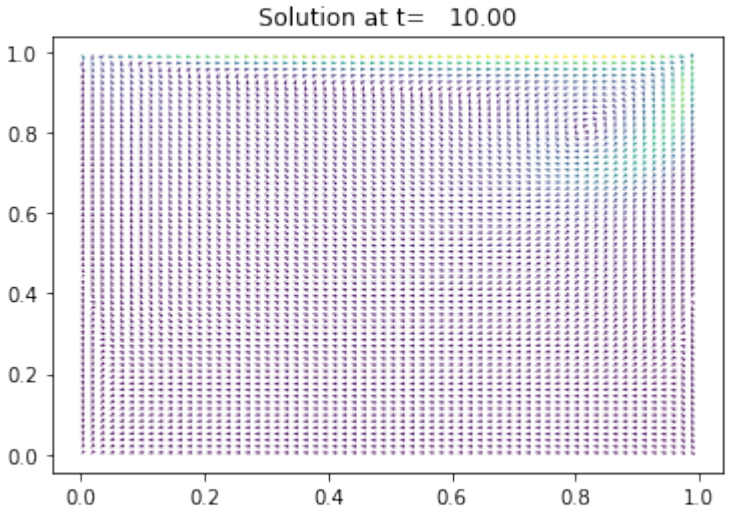
(continues on next page)

(continued from previous page)

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

```
/home/loic/miniconda3/envs/pylbm/lib/python3.6/site-packages/h5py/__init__.py:36:␣
↪FutureWarning: Conversion of the second argument of issubdtype from `float` to `np.
↪floating` is deprecated. In future, it will be treated as `np.float64 == np.
↪dtype(float).type`.
  from ._conv import register_converters as _register_converters
```

```
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
```

(continues on next page)

(continued from previous page)

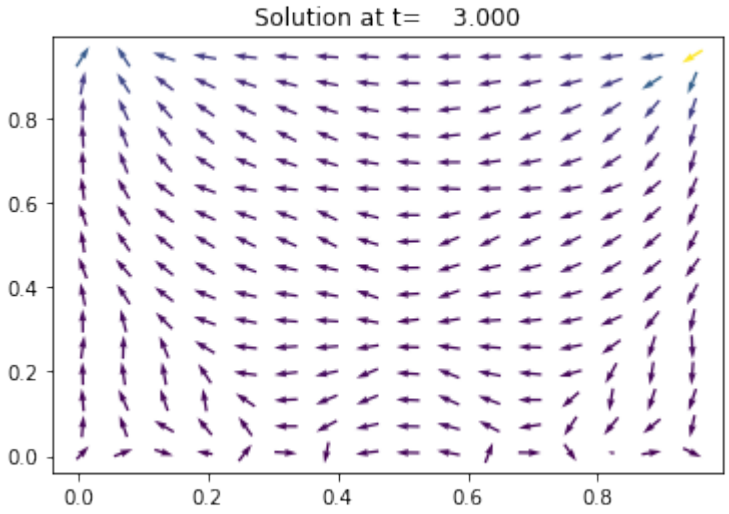
```
xmin, xmax, ymin, ymax, zmin, zmax = 0., 1., 0., 1., 0., 1.
dummy = 3.0/(la*rhoo*dx)

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))

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
```

(continues on next page)

(continued from previous page)

```
            ],
            'init':{rho:rhoo, qx:0., qy:0., qz:0.},
        },
    ],
    'boundary_conditions':{
        0:{'method':{0: pylbm.bc.Bouzidi_bounce_back}, 'value':None},
        1:{'method':{0: pylbm.bc.Bouzidi_bounce_back}, 'value':bc}
    },
    'generator': 'cython',
}

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

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



[ ]:

1.6.7 Von Karman vortex street

In this tutorial, we consider the classical 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 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]:
```
from __future__ import print_function, division
from six.moves import range
%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):
```

(continues on next page)

(continued from previous page)

```
    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
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,
```

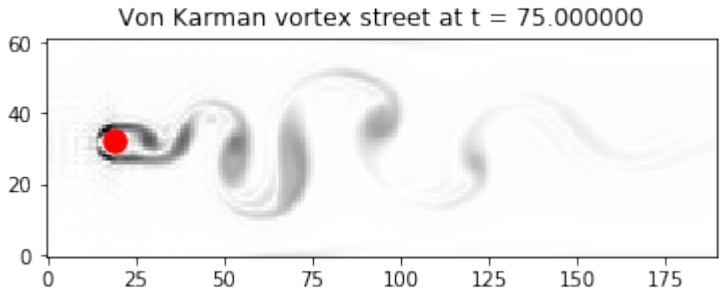
(continues on next page)

(continued from previous page)

```
                -qx/LA, -qy/LA,
                qx2-qy2, qxy
            ],
            'init':{rho:rhoo, qx:0., qy:0.},
        },
    ],
    'boundary_conditions':{
        0:{'method':{0: pylbm.bc.Bouzidi_bounce_back}, 'value':bc_in},
        1:{'method':{0: pylbm.bc.Bouzidi_bounce_back}, 'value':None},
        2:{'method':{0: pylbm.bc.Neumann_x}, 'value':None},
    },
    'generator': 'cython',
}

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

viewer = pylbm.viewer.matplotlibViewer
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()
```

```
/home/loic/miniconda3/envs/pylbm/lib/python3.6/site-packages/h5py/__init__.py:36:␣
↪FutureWarning: Conversion of the second argument of issubdtype from `float` to `np.
↪floating` is deprecated. In future, it will be treated as `np.float64 == np.
↪dtype(float).type`.
  from ._conv import register_converters as _register_converters
```

```
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]
```



1.6.8 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]:
```
from __future__ import print_function, division
%matplotlib inline
import sympy as sp
import numpy as np
import pylbm
```

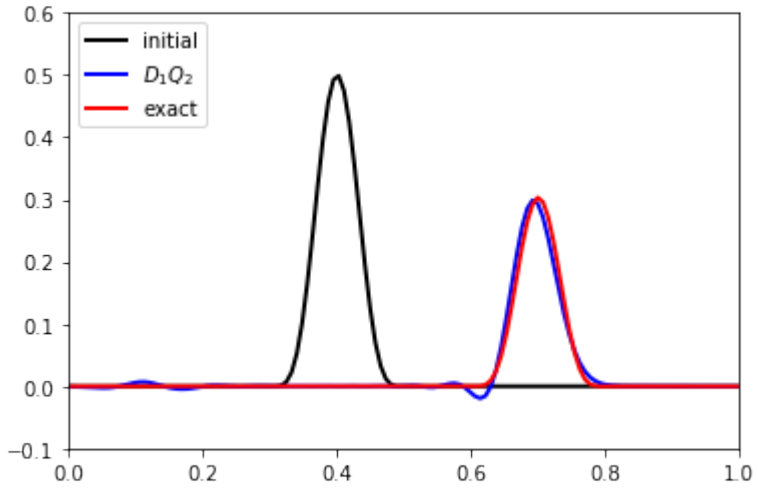
```
/home/loic/miniconda3/envs/pylbm/lib/python3.6/site-packages/h5py/__init__.py:36:␣
↪FutureWarning: Conversion of the second argument of issubdtype from `float` to `np.
↪floating` is deprecated. In future, it will be treated as `np.float64 == np.
↪dtype(float).type`.
  from ._conv import register_converters as _register_converters
```

[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.matplotlibViewer
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.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 !

[ ]:
```
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},
```

(continues on next page)

(continued from previous page)

```
            'init':{u:(init,)},
        },
    ],
    'generator': 'cython',
    'parameters': {LA: 1., C: c, 'time': t},
}

sol = pylbm.Simulation(dico) # build the simulation
viewer = pylbm.viewer.matplotlibViewer
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.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.

1.7 1D examples

1.8 2D examples

1.9 3D examples

CHAPTER

TWO

DOCUMENTATION OF THE CODE

The most important classes

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

2.1 pylbm.Geometry

class pylbm.**Geometry**(*dico*)

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

Parameters **dico** (a dictionary that contains the following *key:value*) –

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

dim

number of spatial dimensions (1, 2, or 3)

Type int

bounds

the bounds of the box in each spatial direction

Type numpy array

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

Type list of integers

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

Type list of elements

Examples

see demo/examples/geometry/

__init__(*dico*)
Initialize self. See help(type(self)) for accurate signature.

Methods

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

2.2 pylbm.Domain

class pylbm.**Domain**(*dico=None*, *geometry=None*, *stencil=None*, *space_step=None*, *verif=True*)
Create a domain that defines the fluid part and the solid part and computes the distances between these two states.

Parameters **dico** (a 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

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.

If the geometry and/or the stencil were previously generated, it can be used directly as following

```
>>> Domain(dico, geometry = geom, stencil = sten)
```

where geom is an object of the class *Geometry* and sten an object of the class *Stencil* In that case, dico does not need to contain the informations for generate the geometry and/or the stencil

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

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

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

Type int

globalbounds
the bounds of the box in each spatial direction

Type numpy array

bounds
the local bounds of the process in each spatial direction

Type numpy array

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

Type double

type
type of data (example: 'float64')

Type string

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

global_size
number of points in each direction

Type list of int

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

Type list of int

coords
coordinates of the domain

Type numpy array

x
first coordinate of the domain

Type numpy array

y
second coordinate of the domain (None if dim<2)

Type numpy array

z
third coordinate of the domain (None if dim<3)

Type numpy array

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

Type numpy array

distance
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.

Type numpy array

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

Type numpy array

valin
value in the fluid domain

Type int

valout
value in the fluid domain

Type int

x_halo

y_halo

z_halo

shape_halo

shape_in

Examples

see demo/examples/domain/

__init__(*dico=None*, *geometry=None*, *stencil=None*, *space_step=None*, *verif=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

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

Attributes

shape_halo	shape of the whole domain with the halo points.
shape_in	shape of the interior domain.
x	x component of the coordinates in the interior domain.
x_halo	x component of the coordinates of the whole domain (halo points included).
y	y component of the coordinates in the interior domain.
y_halo	y component of the coordinates of the whole domain (halo points included).
z	z component of the coordinates in the interior domain.
z_halo	z component of the coordinates of the whole domain (halo points included).

2.3 pylbm.Scheme

class pylbm.**Scheme**(*dico*, *stencil=None*, *check_inverse=False*)

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
- generator : a generator for the code, optional (see `Generator`)
- ode_solver : a method to integrate the source terms, optional (see `ode_solver`)
- test_stability : boolean (optional)

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.

dim
spatial dimension

Type int

dx
space step

Type double

dt
time step

Type double

la
scheme velocity, ratio dx/dt

Type double

nscheme
number of elementary schemes

Type int

stencil
a stencil of velocities

Type object of class *Stencil*

P
list of polynomials that define the moments

Type list of sympy matrix

EQ
list of the equilibrium functions

Type list of sympy matrix

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

Type list of list of doubles

M
the symbolic matrix of the moments

Type sympy matrix

Mnum
the numeric matrix of the moments (m = Mnum F)

Type numpy array

invM
the symbolic inverse matrix

Type sympy matrix

invMnum
the numeric inverse matrix (F = invMnum m)

Type numpy array

generator
the used generator (`NumpyGenerator`, `CythonGenerator`, ...)

Type `Generator`

ode_solver
the used ODE solver (`explicit_euler`, `heun`, ...)

Type `ode_solver`,

Examples

see demo/examples/scheme/

__init__(*dico*, *stencil=None*, *check_inverse=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(dico[, stencil, check_inverse])	Initialize self.
compute_amplification_matrix(wave_vector)	compute the amplification matrix of one time step of the scheme for the given wave vector.
compute_amplification_matrix_relaxation()	compute the amplification matrix of the relaxation.
compute_consistency(dicocons)	compute the consistency of the scheme.
create_moments_matrices()	Create the moments matrices M and M^{-1} used to transform the repartition functions into the moments
equilibrium(mm)	Compute the equilibrium
f2m(ff, mm)	Compute the moments m from the distribution functions f
generate(backend, sorder, valin)	Generate the code by using the appropriated generator
is_L2_stable([Nk])	test the L2 stability of the scheme
is_monotonically_stable()	test the monotonical stability of the scheme.
m2f(mm, ff)	Compute the distribution functions f from the moments m
onetimestep(mm, ff, ff_new, in_or_out, valin)	Compute one time step of the Lattice Boltzmann method
relaxation(m)	The relaxation phase on the moments m
set_boundary_conditions(f, m, bc, interface)	Apply the boundary conditions
set_initialization(scheme)	set the initialization functions for the conserved moments.
set_source_terms(scheme)	set the source terms functions for the conserved moments.

Continued on next page

Table 5 – continued from previous page

source_term(m[, tn, dt, x, y, z])	The integration of the source term on the moments m
transport(f)	The transport phase on the distribution functions f
vp_amplification_matrix(wave_vector)	compute the eigenvalues of the amplification matrix for a given wave vector.

2.4 pylbm.Simulation

class pylbm.**Simulation**(*dico*, *domain=None*, *scheme=None*, *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')*) –

dim
spatial dimension

Type int

type
the type of the values

Type float64

domain
the domain given in argument

Type *Domain*

scheme
the scheme given in argument

Type Scheme

m
a numpy array that contains the values of the moments in each point

Type numpy array

F
a numpy array that contains the values of the distribution functions in each point

Type numpy array

m_halo
a numpy array that contains the values of the moments in each point

Type numpy array

F_halo
a numpy array that contains the values of the distribution functions in each point

Type numpy array

Examples

see demo/examples/

Access to the distribution functions and the moments.

In 1D:

```
>>>F[n][k][i]
>>>m[n][k][i]
```

get the kth distribution function of the nth elementary scheme and the kth moment of the nth elementary scheme at the point x[0][i].

In 2D:

```
>>>F[n][k][j, i]
>>>m[n][k][j, i]
```

get the kth distribution function of the nth elementary scheme and the kth moment of the nth elementary scheme at the point x[0][i], x[1][j].

Notes

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

__init__(*dico*, *domain=None*, *scheme=None*, *sorder=None*, *dtype='float64'*, *check_inverse=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(dico[, domain, scheme, sorder, ...])	Initialize self.
boundary_condition()	perform the boundary conditions
equilibrium()	set the moments to the equilibrium values (the array _m is modified)
f2m()	compute the moments from the distribution functions (the array _m is modified)
initialization(dico)	initialize all the numy array with the initial conditions
m2f()	compute the distribution functions from the moments (the array _F is modified)
one_time_step()	compute one time step
relaxation()	compute the relaxation phase on moments (the array _m is modified)
source_term([fraction_of_time_step])	compute the source term phase on moments (the array _m is modified)
time_info()	
transport()	compute the transport phase on distribution functions (the array _F is modified)

Attributes

F	get the distribution function i in the interior domain.
F_halo	get the distribution function i on the whole domain with halo points.
m	get the moment i in the interior domain.
m_halo	get the moment i on the whole domain with halo points.

The modules

2.5 the module stencil

Stencil(dico)	Create the stencil of velocities used by the scheme(s).
OneStencil(v, nv, num2index, nv_ptr)	Create a stencil of a LBM scheme.
Velocity([dim, num, vx, vy, vz])	Create a velocity.

2.5.1 pylbm.stencil.Stencil

class pylbm.stencil.**Stencil**(*dico*)

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 the dictionaries that contain the key:value velocities

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

dim

the spatial dimension (1, 2 or 3).

Type int

unique_velocities

array of all velocities involved in the stencils. Each unique velocity appeared only once.

Type NumPy array

uvx

the x component of the unique velocities.

Type NumPy array

uvy

the y component of the unique velocities.

Type NumPy array

uvz

the z component of the unique velocities.

Type NumPy array

unum

the numbering of the unique velocities.

Type NumPy array

vmax
the maximal velocity in norm for each spatial direction.

Type int

vmin
the minimal velocity in norm for each spatial direction.

Type int

nstencils
the number of elementary stencils.

Type int

nv
the number of velocities for each elementary stencil.

Type list of integers

v
list of all the velocities for each elementary stencil.

Type list of velocities

vx
the x component of the velocities for the stencil k.

Type NumPy array

vy
the y component of the velocities for the stencil k.

Type NumPy array

vz
the z component of the velocities for the stencil k.

Type NumPy array

num
the numbering of the velocities for the stencil k.

Type NumPy array

nv_ptr
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]]

Type list of integers

unvtot
the number of unique velocities involved in the stencils.

Type int

Notes

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

Examples

```
>>> s = Stencil({'dim': 1,
              'schemes':[{'velocities': range(9)}, ],
                 })
>>> s
Stencil informations
  * spatial dimension: 1
  * maximal velocity in each direction: [4 None None]
  * minimal velocity in each direction: [-4 None None]
  * Informations 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':range(9)},
                            {'velocities':range(50)},
                           ],
                })
>>> s
Stencil informations
  * spatial dimension: 2
  * maximal velocity in each direction: [4 3 None]
  * minimal velocity in each direction: [-3 -3 None]
  * Informations 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: 50
        - 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), (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), (49: 4, 0),
```

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,  4])
```

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,  0])
```

__init__(*dico*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(dico)	Initialize self.
append(object)	
clear()	
copy()	
count(value)	
extend(iterable)	
get_all_velocities([ischeme])	get all the velocities for all the stencils in one array
get_stencil(k)	
get_symmetric([axis])	get the symetrics velocities.
index(value, [start, [stop]])	Raises ValueError if the value is not present.
insert	L.insert(index, object) – insert object before index
is_symmetric()	check if all the velocities have their symetric.
pop([index])	Raises IndexError if list is empty or index is out of range.
remove(value)	Raises ValueError if the value is not present.
reverse	L.reverse() – reverse *IN PLACE*
sort([key, reverse])	
visualize([viewer_mod, k, unique_velocities])	plot the velocities

Attributes

num	num[k] the numbering of the velocities for the stencil k.
unum	the numbering of the unique velocities.
unvtot	the number of unique velocities involved in the stencils.
uvx	the x component of the unique velocities.
uvy	the y component of the unique velocities.
uvz	the z component of the unique velocities.
vmax	the maximal velocity in norm for each spatial direction.
vmin	the minimal velocity in norm for each spatial direction.
vx	vx[k] the x component of the velocities for the stencil k.
vy	vy[k] the y component of the velocities for the stencil k.
vz	vz[k] the z component of the velocities for the stencil k.

2.5.2 pylbm.stencil.OneStencil

class pylbm.stencil.**OneStencil**(*v*, *nv*, *num2index*, *nv_ptr*)

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

v
the list of the velocities of that stencil

Type list

nv
size of the list v

Type int

num2index
link between the velocity number and its position in the unique velocities array

Type list of integers

num
the numbering of the velocities

vx
the x component of the velocities

vy
the y component of the velocities

vz

__init__(*v*, *nv*, *num2index*, *nv_ptr*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(v, nv, num2index, nv_ptr)	Initialize self.

Attributes

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.

2.5.3 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.

dim
The dimension of the velocity.

Type int

num
The number of the velocity in the numbering convention of Lattice-Boltzmann scheme.

vx
The x component of the velocity vector.

Type int

vy
The y component of the velocity vector.

Type int

vz
The z component of the velocity vector.

Type int

v

Type list

Examples

Create a velocity with the dimension and the number

```
>>> v = Velocity(dim = 1, num = 2)
>>> v
velocity 2
 vx: -1
```

Create a velocity with a direction

```
>>> v = Velocity(vx=1, vy=1)
>>> v
velocity 5
 vx: 1
 vy: 1
```

Notes

```
from __future__ import print_function, division
from six.moves import range
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.colors as colors
import matplotlib.cm as cm
from matplotlib.patches import FancyArrowPatch
```

(continues on next page)

(continued from previous page)

```
from mpl_toolkits.mplot3d import Axes3D, proj3d

import pylbm
import numpy as np

class Arrow3D(FancyArrowPatch):
    def __init__(self, xs, ys, zs, *args, **kwargs):
        FancyArrowPatch.__init__(self, (0,0), (0,0), *args, **kwargs)
        self._verts3d = xs, ys, zs

    def draw(self, renderer):
        xs3d, ys3d, zs3d = self._verts3d
        xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d, renderer.M)
        self.set_positions((xs[0],ys[0]),(xs[1],ys[1]))
        FancyArrowPatch.draw(self, renderer)

def Velocities_1D(n):
    dim = 1
    fig = plt.figure(dim, figsize=(8, 4), facecolor='white')
    fig.clf()
    xmin, xmax, ymin, ymax = 1000, -1000, -1, 1
    e = 0.2
    for k in range((2*n+1)**dim):
        v = pylbm.stencil.Velocity(dim = dim, num = k)
        x = v.vx
        xmin = min(xmin, x)
        xmax = max(xmax, x)
        couleur_texte = 0.
        couleur_trait = 0.5
        plt.text(x, 0., str(v.num), color=[couleur_texte]*3,
                 horizontalalignment='center',verticalalignment='center',
                 fontsize=15)

    plt.plot([xmin, xmax], [0, 0], ':', color=[couleur_trait]*3)
    plt.text(0., ymax+2*e, "Velocities numbering {0:1d}D".format(dim),fontsize=20,
        verticalalignment='center', horizontalalignment='center', color='b')
    plt.arrow(xmin-e, ymin-e, 1, 0, head_width=0.05*dim, head_length=0.1, fc='b',␣
↪ec='b')
    plt.text(xmin-e+.5, ymin-1.5*e, 'x', color='b',
        verticalalignment='center', horizontalalignment='center')
    plt.axis('off')
    plt.xlim(xmin-2*e, xmax+2*e)
    plt.ylim(ymin-2*e, ymax+2*e)
    plt.draw()

def Velocities_2D(n):
    dim = 2
    fig = plt.figure(dim, figsize=(8, 8), facecolor='white')
    fig.clf()
    xmin, xmax, ymin, ymax = 1000, -1000, 1000, -1000
    e = .5
    for k in range((2*n+1)**dim):
        v = pylbm.stencil.Velocity(dim = dim, num = k)
        x = v.vx
        y = v.vy
        xmin = min(xmin, x)
        xmax = max(xmax, x)
```

(continues on next page)

(continued from previous page)

```
        ymin = min(ymin, y)
        ymax = max(ymax, y)
        couleur_texte = 0.
        couleur_trait = 0.5
        plt.text(x, y, str(v.num), color=[couleur_texte]*3,
                 horizontalalignment='center',verticalalignment='center',
                 fontsize=15)
    for x in range(xmin, xmax+1):
        plt.plot([x, x], [ymin, ymax], ':', color=[couleur_trait]*3)
    for y in range(ymin, ymax+1):
        plt.plot([xmin, xmax], [y, y], ':', color=[couleur_trait]*3)
    plt.text(0., ymax+2*e, "Velocities numbering {0:1d}D".format(dim),fontsize=20,
        verticalalignment='center', horizontalalignment='center', color='b')
    plt.arrow(xmin-e, ymin-e, 1, 0, head_width=0.05*dim, head_length=0.1, fc='b',␣
↪ec='b')
    plt.arrow(xmin-e, ymin-e, 0, 1, head_width=0.05*dim, head_length=0.1, fc='b',␣
↪ec='b')
    plt.text(xmin-e+.5, ymin-1.5*e, 'x', color='b',
        verticalalignment='center', horizontalalignment='center')
    plt.text(xmin-1.5*e, ymin-e+.5, 'y', color='b',
        verticalalignment='center', horizontalalignment='center')
    plt.axis('off')
    plt.xlim(xmin-2*e, xmax+2*e)
    plt.ylim(ymin-2*e, ymax+2*e)
    plt.draw()

def Velocities_3D(n):
    dim = 3
    couleur_tour = "k"
    fig = plt.figure(dim, figsize=(8, 8), facecolor='white')
    fig.clf()
    ax = fig.add_subplot(111, projection='3d')
    xmin, xmax, ymin, ymax, zmin, zmax = 1000, -1000, 1000, -1000, 1000, -1000
    e = .5
    for k in range((2*n+1)**dim):
        v = pylbm.stencil.Velocity(dim = dim, num = k)
        x = v.vx
        y = v.vy
        z = v.vz
        xmin = min(xmin, x)
        xmax = max(xmax, x)
        ymin = min(ymin, y)
        ymax = max(ymax, y)
        zmin = min(zmin, z)
        zmax = max(zmax, z)
        couleur_texte = [.5+.5*x, 0., .5-.5*x]
        couleur_trait = 0.5
        ax.text(x, y, z, str(v.num), color=couleur_texte,
                 horizontalalignment='center',verticalalignment='center',
                 fontsize=15)
    for x in range(xmin, xmax+1):
        for y in range(ymin, ymax+1):
            ax.plot([x, x], [y, y], [zmin, zmax], ':', color=[couleur_trait]*3)
    for x in range(xmin, xmax+1):
        for z in range(zmin, zmax+1):
            ax.plot([x, x], [ymin, ymax], [z, z], ':', color=[couleur_trait]*3)
    for z in range(zmin, zmax+1):
```

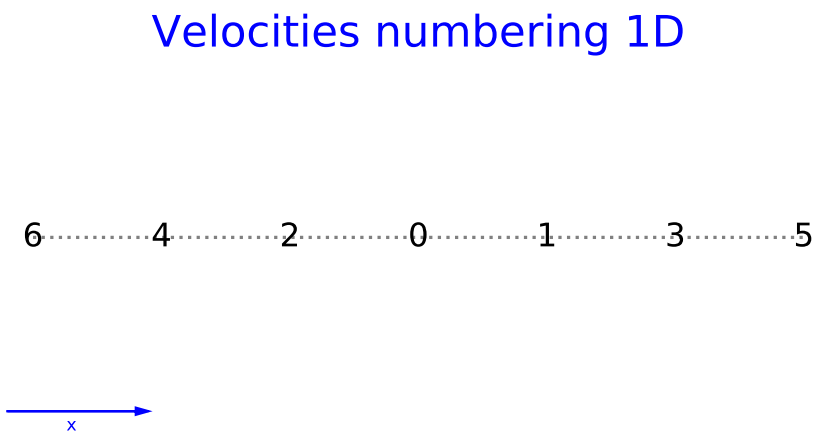
(continues on next page)

(continued from previous page)

```
        for y in range(ymin, ymax+1):
            ax.plot([xmin, xmax], [y, y], [z, z], ':', color=[couleur_trait]*3)

    XS, YS = np.meshgrid([-1,1],[-1,1])
    ZS = np.zeros(XS.shape)
    couleur_plan = .8
    for x in [-1,0,1]:
        ax.plot_surface(ZS+x, XS, YS,
            rstride=1, cstride=1, color=[.5+.5*x, 0., .5-.5*x],
            shade=False, alpha=0.2,
            antialiased=False, linewidth=0)
    ax.text(0., 0., zmax+2*e, "Velocities numbering {0:1d}D".format(dim),␣
↪fontsize=20,
        verticalalignment='center', horizontalalignment='center', color=couleur_
↪tour)
    vx = Arrow3D([xmax+e,xmax+e+1],[ymin-e,ymin-e],[zmin-e,zmin-e],
        mutation_scale=20, lw=1, arrowstyle="-|>", color=couleur_tour)
    ax.add_artist(vx)
    vy = Arrow3D([xmax+e,xmax+e],[ymin-e,ymin-e+1],[zmin-e,zmin-e],
        mutation_scale=20, lw=1, arrowstyle="-|>", color=couleur_tour)
    ax.add_artist(vy)
    vz = Arrow3D([xmax+e,xmax+e],[ymin-e,ymin-e],[zmin-e,zmin-e+1],
        mutation_scale=20, lw=1, arrowstyle="-|>", color=couleur_tour)
    ax.add_artist(vz)
    ax.text(xmax+e+.8, ymin-.8*e, zmin-e, 'x', color=couleur_tour,
        verticalalignment='center', horizontalalignment='center')
    ax.text(xmax+e, ymin-e+.8, zmin-1.2*e, 'y', color=couleur_tour,
        verticalalignment='center', horizontalalignment='center')
    ax.text(xmax+e, ymin-1.2*e, zmin-e+.8, 'z', color=couleur_tour,
        verticalalignment='center', horizontalalignment='center')
    ax.set_xlim(xmin-e, xmax+e)
    ax.set_ylim(ymin-e, ymax+e)
    ax.set_zlim(zmin-e, zmax+e)
    ax.azim = 34
    ax.elev = 20
    plt.axis('off')
    plt.draw()

def Velocities(dim, n):
    if dim == 1:
        Velocities_1D(n)
    elif dim == 2:
        Velocities_2D(n)
    elif dim == 3:
        Velocities_3D(n)
    else:
        print("error of dimension")
    plt.show()
```

```
Velocities(1, 3)
Velocities(2, 2)
Velocities(3, 1)
```

__init__ (*dim=None*, *num=None*, *vx=None*, *vy=None*, *vz=None*)
Initialize self. See help(type(self)) for accurate signature.



Methods

__init__([dim, num, vx, vy, vz])	Initialize self.
get_symmetric([axis])	return the symmetric velocity.
set_symmetric()	create the symetric velocity.

Attributes

v	velocity

2.6 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:

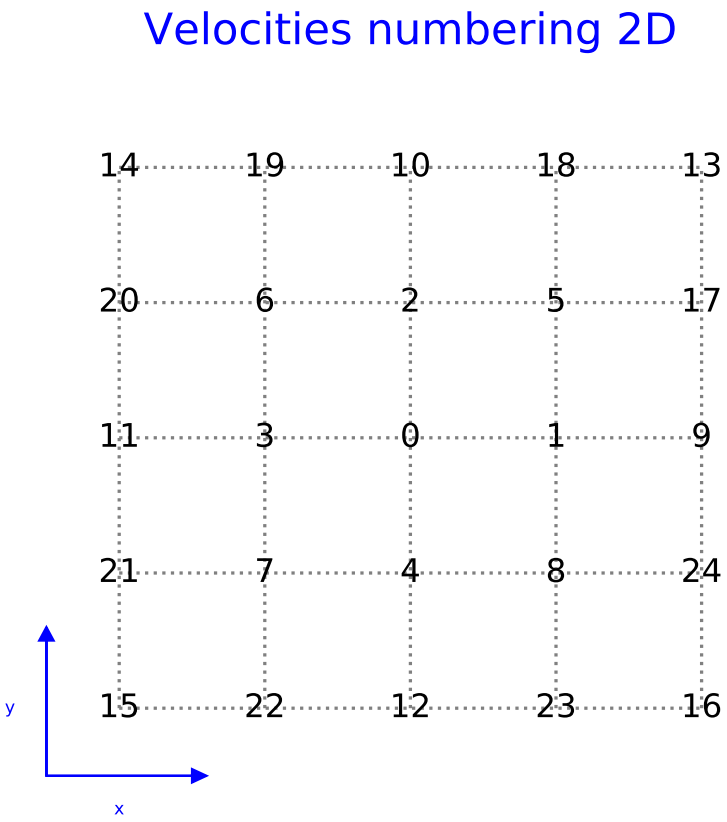
Circle(center, radius[, label, isfluid])	Class Circle
Ellipse(center, v1, v2[, label, isfluid])	Class Ellipse
Parallelogram(point, vecta, vectb[, label, ...])	Class Parallelogram
Triangle(point, vecta, vectb[, label, isfluid])	Class Triangle

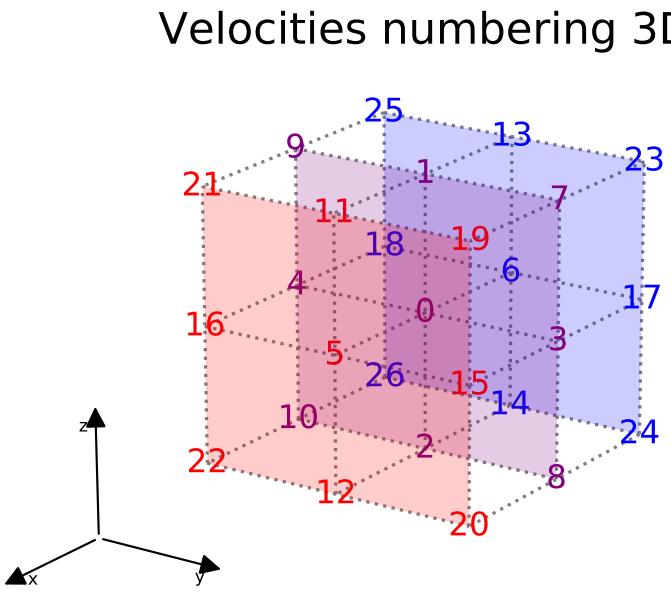
2.6.1 pylbm.elements.Circle

class pylbm.elements.**Circle**(*center*, *radius*, *label=0*, *isfluid=False*)
Class Circle

Parameters

- **center** (*a list that contains the two coordinates of the center*) –
- **radius** (*a positive float for the radius*) –
- **label** (*list of one integer (default [0])*) –
- **isfluid** (*boolean*) –


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

label
the list of the label of the edge

Type list of integers

isfluid
True if the circle is added and False if the circle is deleted

Type boolean

Examples

the circle centered in (0, 0) with radius 1

```
>>> center = [0., 0.]
>>> radius = 1.
>>> Circle(center, radius)
    Circle([0 0],1) (solid)
```

__init__(*center*, *radius*, *label=0*, *isfluid=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(center, radius[, label, isfluid])	Initialize self.
distance(grid, v[, dmax])	Compute the distance in the v direction between the circle and the points defined by (x, y).
get_bounds()	Get the bounds of the circle.
point_inside(grid)	return a boolean array which defines if a point is inside or outside of the circle.
test_label()	test if the number of labels is equal to the number of bounds.

2.6.2 pylbm.elements.Ellipse

class pylbm.elements.**Ellipse**(*center*, *v1*, *v2*, *label=0*, *isfluid=False*)
Class Ellipse

Parameters

- **center** (*a list that contains the two coordinates of the center*) –
- **v1** (*a vector*) –
- **v2** (*a second vector (v1 and v2 have to be othogonal)*) –
- **label** (*list of one integer (default [0])*) –
- **isfluid** (*boolean*) –
  - True if the ellipse is added
  - False if the ellipse is deleted

number_of_bounds
1

Type int

center
the coordinates of the center of the ellipse

Type numpy array

v1
the coordinates of the first vector

Type numpy array

v2
the coordinates of the second vector

Type numpy array

label
the list of the label of the edge

Type list of integers

isfluid
True if the ellipse is added and False if the ellipse is deleted

Type boolean

number_of_bounds
number of edges (1)

Type int

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([0 0], [2 0], [0 1]) (solid)
```

__init__(*center*, *v1*, *v2*, *label=0*, *isfluid=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(center, v1, v2[, label, isfluid])	Initialize self.
distance(grid, v[, dmax])	Compute the distance in the v direction between the ellipse and the points defined by (x, y).
get_bounds()	Get the bounds of the ellipse.
point_inside(grid)	return a boolean array which defines if a point is inside or outside of the ellipse.
test_label()	test if the number of labels is equal to the number of bounds.

2.6.3 pylbm.elements.Parallelogram

class pylbm.elements.**Parallelogram**(*point*, *vecta*, *vectb*, *label=0*, *isfluid=False*)
Class Parallelogram

Parameters

- **point** (*the coordinates of the first point of the parallelogram*) –
- **vecta** (*the coordinates of the first vector*) –
- **vectb** (*the coordinates of the second vector*) –
- **label** (*list of 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([0 0],[0 1],[1 0]) (solid)
```

number_of_bounds
4

Type int

point
the coordinates of the first point of the parallelogram

Type numpy array

vecta
the coordinates of the first vector

Type numpy array

vectb
the coordinates of the second vector

Type numpy array

label
the list of the label of the edge

Type list of integers

isfluid
True if the parallelogram is added and False if the parallelogram is deleted

Type boolean

number_of_bounds
number of edges (4)

Type int

__init__ (*point*, *vecta*, *vectb*, *label=0*, *isfluid=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(point, vecta, vectb[, label, isfluid])	Initialize self.
distance(grid, v[, dmax])	Compute the distance in the v direction between the parallelogram and the points defined by (x, y).
get_bounds()	return the bounds of the parallelogram.
point_inside(grid)	return a boolean array which defines if a point is inside or outside of the parallelogram.
test_label()	test if the number of labels is equal to the number of bounds.

2.6.4 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 of 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([0 0],[0 1],[1 0]) (solid)
```

point
the coordinates of the first point of the triangle

Type numpy array

vecta
the coordinates of the first vector

Type numpy array

vectb
the coordinates of the second vector

Type numpy array

label
the list of the label of the edge

Type list of integers

isfluid
True if the triangle is added and False if the triangle is deleted

Type boolean

number_of_bounds
number of edges

Type int

__init__(*point*, *vecta*, *vectb*, *label=0*, *isfluid=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(point, vecta, vectb[, label, isfluid])	Initialize self.
distance(grid, v[, dmax])	Compute the distance in the v direction between the triangle and the points defined by (x, y).
get_bounds()	return the smallest box where the triangle is.
point_inside(grid)	return a boolean array which defines if a point is inside or outside of the triangle.
test_label()	test if the number of labels is equal to the number of bounds.

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
Cylinder_Circle(center, v1, v2, w[, label, ...])	Class Cylinder_Circle
Cylinder_Ellipse(center, v1, v2, w[, label, ...])	Class Cylinder_Ellipse
Cylinder_Triangle(center, v1, v2, w[, ...])	Class Cylinder_Triangle

2.6.5 pylbm.elements.Sphere

class pylbm.elements.**Sphere**(*center*, *radius*, *label=0*, *isfluid=False*)
Class Sphere

Parameters

- **center** (*a list that contains the three coordinates of the center*) –
- **radius** (*a positive float for the radius*) –
- **label** (*list of one integer (default [0])*) –
- **isfluid** (*boolean*) –

- True if the sphere is added
- False if the sphere is deleted

number_of_bounds
1

Type int

center
the coordinates of the center of the sphere

Type numpy array

radius
positive float for the radius of the sphere

Type double

label
the list of the label of the edge

Type list of integers

isfluid
True if the sphere is added and False if the sphere is deleted

Type boolean

number_of_bounds
number of edges (1)

Type int

Examples

the sphere centered in (0, 0, 0) with radius 1

```
>>> center = [0., 0., 0.]
>>> radius = 1.
>>> Sphere(center, radius)
    Sphere([0 0 0],1) (solid)
```

__init__(*center*, *radius*, *label=0*, *isfluid=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(center, radius[, label, isfluid])	Initialize self.
distance(grid, v[, dmax])	Compute the distance in the v direction between the sphere and the points defined by (x, y, z).
get_bounds()	Get the bounds of the sphere.
point_inside(grid)	return a boolean array which defines if a point is inside or outside of the sphere.
test_label()	test if the number of labels is equal to the number of bounds.

2.6.6 pylbm.elements.Ellipsoid

class pylbm.elements.**Ellipsoid**(*center*, *v1*, *v2*, *v3*, *label=0*, *isfluid=False*)

Class Ellipsoid

Parameters

- **center** (*a list that contains the three coordinates of the center*) –
- **v1** (*a vector*) –
- **v2** (*a vector*) –
- **v3** (*a vector (v1, v2, and v3 have to be orthogonal)*) –
- **label** (*list of one integer (default [0])*) –
- **isfluid** (*boolean*) –
  - True if the ellipsoid is added
  - False if the ellipsoid is deleted

number_of_bounds

1

Type int

center

the coordinates of the center of the sphere

Type numpy array

v1

the coordinates of the first vector

Type numpy array

v2

the coordinates of the second vector

Type numpy array

v3

the coordinates of the third vector

Type numpy array

label

the list of the label of the edge

Type list of integers

isfluid

True if the ellipsoid is added and False if the ellipsoid is deleted

Type boolean

number_of_bounds

number of edges (1)

Type int

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([0 0 0], [3 0 0], [0 2 0], [0 0 1]) (solid)
```

__init__(*center*, *v1*, *v2*, *v3*, *label=0*, *isfluid=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(center, v1, v2, v3[, label, isfluid])	Initialize self.
distance(grid, v[, dmax])	Compute the distance in the v direction between the ellipsoid and the points defined by (x, y, z).
get_bounds()	Get the bounds of the ellipsoid.
point_inside(grid)	return a boolean array which defines if a point is inside or outside of the ellipsoid.
test_label()	test if the number of labels is equal to the number of bounds.

2.6.7 pylbm.elements.Parallelepiped

class pylbm.elements.**Parallelepiped**(*point*, *v0*, *v1*, *v2*, *label=0*, *isfluid=False*)
Class Parallelepiped

Parameters

- **point** (*a list that contains the three coordinates of the first point*) –
- **v0** (*a list of the three coordinates of the first vector that defines the edge*) –
- **v1** (*a list of the three coordinates of the second vector that defines the edge*) –
- **v2** (*a list of the three coordinates of the third vector that defines the edge*) –
- **label** (*list of 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

number_of_bounds
6

Type int

point
the coordinates of the first point of the parallelepiped

Type numpy array

v0
the three coordinates of the first vector

Type list of doubles

v1
the three coordinates of the second vector

Type list of doubles

v2
the three coordinates of the third vector

Type list of doubles

label
the list of the label of the edge

Type list of integers

isfluid
True if the parallelepiped is added and False if the parallelepiped is deleted

Type boolean

number_of_bounds
number of edges (6)

Type int

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([0 0 0], [1 0 0], [0 1 0], [0 0 1]) (solid)
```

__init__(*point*, *v0*, *v1*, *v2*, *label=0*, *isfluid=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(point, v0, v1, v2[, label, isfluid])	Initialize self.
change_of_variables()	
distance(grid, v[, dmax])	Compute the distance in the v direction between the cylinder and the points defined by (x, y, z).
get_bounds()	Get the bounds of the cylinder.
point_inside(grid)	return a boolean array which defines if a point is inside or outside of the cylinder.
test_label()	test if the number of labels is equal to the number of bounds.

2.6.8 pylbm.elements.Cylinder_Circle

class pylbm.elements.**Cylinder_Circle**(*center*, *v1*, *v2*, *w*, *label=0*, *isfluid=False*)
Class Cylinder_Circle

Parameters

- **center** (*a list that contains the three coordinates of the center*) –
- **v0** (*a list of the three coordinates of the first vector that defines the circular section*) –
- **v1** (*a list of the three coordinates of the second vector that defines the circular section*) –
- **w** (*a list of the three coordinates of the vector that defines the direction of the side*) –
- **label** (*list of 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

number_of_bounds
3

Type int

center
the coordinates of the center of the cylinder

Type numpy array

v0
the three coordinates of the first vector that defines the base section

Type list of doubles

v1
the three coordinates of the second vector that defines the base section

Type list of doubles

w
the three coordinates of the vector that defines the direction of the side

Type list of doubles

label
the list of the label of the edge

Type list of integers

isfluid
True if the cylinder is added and False if the cylinder is deleted

Type boolean

number_of_bounds
number of edges (3)

Type int

Examples

the vertical canonical cylinder centered in (0, 0, 1/2) with radius 1

```
>>> center = [0., 0., 0.5]
>>> v0, v1 = [1., 0., 0.], [0., 1., 0.]
>>> w = [0., 0., 1.]
>>> Cylinder_Circle(center, v0, v1, w)
    Cylinder_Circle([0 0 0.5], [1 0 0], [0 1 0], [0 0 1]) (solid)
```

__init__(*center*, *v1*, *v2*, *w*, *label=0*, *isfluid=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(center, v1, v2, w[, label, isfluid])	Initialize self.
change_of_variables()	
distance(grid, v[, dmax])	Compute the distance in the v direction between the cylinder and the points defined by (x, y, z).
get_bounds()	Get the bounds of the cylinder.
point_inside(grid)	return a boolean array which defines if a point is inside or outside of the cylinder.
test_label()	test if the number of labels is equal to the number of bounds.

2.6.9 pylbm.elements.Cylinder_Ellipse

class pylbm.elements.**Cylinder_Ellipse**(*center*, *v1*, *v2*, *w*, *label=0*, *isfluid=False*)
Class Cylinder_Ellipse

Parameters

- **center** (*a list that contains the three coordinates of the center*) –
- **v0** (*a list of the three coordinates of the first vector that defines the circular section*) –
- **v1** (*a list of the three coordinates of the second vector that defines the circular section*) –
- **w** (*a list of the three coordinates of the vector that defines the direction of the side*) –
- **label** (*list of 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.

number_of_bounds
3

Type int

center
the coordinates of the center of the cylinder

Type numpy array

v0
the three coordinates of the first vector that defines the base section

Type list of doubles

v1
the three coordinates of the second vector that defines the base section

Type list of doubles

w
the three coordinates of the vector that defines the direction of the side

Type list of doubles

label
the list of the label of the edge

Type list of integers

isfluid
True if the cylinder is added and False if the cylinder is deleted

Type boolean

number_of_bounds
number of edges (3)

Type int

number_of_bounds
number of edges (3)

Type int

Examples

the vertical canonical cylinder centered in (0, 0, 1/2) with radius 1

```
>>> center = [0., 0., 0.5]
>>> v0, v1 = [1., 0., 0.], [0., 1., 0.]
>>> w = [0., 0., 1.]
>>> Cylinder_Ellipse(center, v0, v1, w)
    Cylinder_Ellipse([0 0 0.5], [1 0 0], [0 1 0], [0 0 1]) (solid)
```

__init__(*center*, *v1*, *v2*, *w*, *label=0*, *isfluid=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(center, v1, v2, w[, label, isfluid])	Initialize self.
`change_of_variables`()	
`distance`(grid, v[, dmax])	Compute the distance in the v direction between the cylinder and the points defined by (x, y, z).
`get_bounds`()	Get the bounds of the cylinder.
`point_inside`(grid)	return a boolean array which defines if a point is inside or outside of the cylinder.
`test_label`()	test if the number of labels is equal to the number of bounds.

2.6.10 pylbm.elements.Cylinder_Triangle

class pylbm.elements.**Cylinder_Triangle**(*center*, *v1*, *v2*, *w*, *label=0*, *isfluid=False*)

Class Cylinder_Triangle

Parameters

- **center** (*a list that contains the three coordinates of the center*) –
- **v0** (*a list of the three coordinates of the first vector that defines the triangular section*) –
- **v1** (*a list of the three coordinates of the second vector that defines the triangular section*) –
- **w** (*a list of the three coordinates of the vector that defines the direction of the side*) –
- **label** (*list of 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

number_of_bounds

5

Type int

center

the coordinates of the center of the cylinder

Type numpy array

v0

the three coordinates of the first vector that defines the base section

Type list of doubles

v1

the three coordinates of the second vector that defines the base section

Type list of doubles

w
the three coordinates of the vector that defines the direction of the side

Type list of doubles

label
the list of the label of the edge

Type list of integers

isfluid
True if the cylinder is added and False if the cylinder is deleted

Type boolean

number_of_bounds
number of edges (3)

Type int

Examples

the vertical canonical cylinder centered in (0, 0, 1/2)

```
>>> center = [0., 0., 0.5]
>>> v0, v1 = [1., 0., 0.], [0., 1., 0.]
>>> w = [0., 0., 1.]
>>> Cylinder_Triangle(center, v0, v1, w)
    Cylinder_Triangle([0 0 0.5], [1 0 0], [0 1 0], [0 0 1]) (solid)
```

__init__(*center*, *v1*, *v2*, *w*, *label=0*, *isfluid=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(center, v1, v2, w[, label, isfluid])	Initialize self.
change_of_variables()	
distance(grid, v[, dmax])	Compute the distance in the v direction between the cylinder and the points defined by (x, y, z).
get_bounds()	Get the bounds of the cylinder.
point_inside(grid)	return a boolean array which defines if a point is inside or outside of the cylinder.
test_label()	test if the number of labels is equal to the number of bounds.

2.7 the module geometry

Geometry(dico)	Create a geometry that defines the fluid part and the solid part.

2.7.1 pylbm.geometry.Geometry

class pylbm.geometry.**Geometry**(*dico*)
Create a geometry that defines the fluid part and the solid part.

Parameters **dico** (a dictionary that contains the following *key:value*) –

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

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

Type int

bounds
the bounds of the box in each spatial direction

Type numpy array

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

Type list of integers

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

Type list of elements

Examples

see demo/examples/geometry/

__init__(*dico*)
Initialize self. See help(type(self)) for accurate signature.

Methods

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

2.8 the module domain

Domain([dico, geometry, stencil, ...])	Create a domain that defines the fluid part and the solid part and computes the distances between these two states.

2.8.1 pylbm.domain.Domain

class pylbm.domain.**Domain**(*dico=None*, *geometry=None*, *stencil=None*, *space_step=None*, *verif=True*)

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

Parameters **dico** (a 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`

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.

If the geometry and/or the stencil were previously generated, it can be used directly as following

```
>>> Domain(dico, geometry = geom, stencil = sten)
```

where geom is an object of the class *Geometry* and sten an object of the class *Stencil* In that case, dico does not need to contain the informations for generate the geometry and/or the stencil

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

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

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

Type int

globalbounds
the bounds of the box in each spatial direction

Type numpy array

bounds
the local bounds of the process in each spatial direction

Type numpy array

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

Type double

type
type of data (example: 'float64')

Type string

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

global_size
number of points in each direction

Type list of int

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

Type list of int

coords
coordinates of the domain

Type numpy array

x
first coordinate of the domain

Type numpy array

y
second coordinate of the domain (None if dim<2)

Type numpy array

z
third coordinate of the domain (None if dim<3)

Type numpy array

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

Type numpy array

distance
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.

Type numpy array

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

Type numpy array

valin
value in the fluid domain

Type int

valout
value in the fluid domain

Type int

x_halo

y_halo

z_halo

shape_halo

shape_in

Examples

see demo/examples/domain/

__init__(*dico=None*, *geometry=None*, *stencil=None*, *space_step=None*, *verif=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

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

Attributes

shape_halo	shape of the whole domain with the halo points.

Continued on next page

Table 31 – continued from previous page

shape_in	shape of the interior domain.
x	x component of the coordinates in the interior domain.
x_halo	x component of the coordinates of the whole domain (halo points included).
y	y component of the coordinates in the interior domain.
y_halo	y component of the coordinates of the whole domain (halo points included).
z	z component of the coordinates in the interior domain.
z_halo	z component of the coordinates of the whole domain (halo points included).

2.9 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.

2.9.1 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 of int*) – number of points in each direction including the fictitious point
- **vmax** (*list of int*) – the size of the fictitious points in each direction
- **sorder** (*list of int*) – the order of nv, nx, ny and nz Default is None which mean [nv, nx, ny, nz]
- **mpi_topo** – the mpi topology
- **dtype** (*type*) – the type of the array. Default is numpy.double

array

nspace

nv

shape

size

__init__(*nv*, *gspace_size*, *vmax*, *sorder=None*, *mpi_topo=None*, *dtype=<class 'numpy.float64'>*, *gpu_support=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(nv, gspace_size, vmax[, sorder, ...])	Initialize self.
generate()	generate periodic conditions functions for loo.py backend.
set_conserved_moments(consm, nv_ptr)	add conserved moments information to have a direct access.
update()	update ghost points on the interface with the datas of the neighbors.

Attributes

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.

2.9.2 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 of int*) – number of points in each direction including the fictitious point
- **vmax** (*list of int*) – the size of the fictitious points in each direction
- **mpi_topo** – the mpi topology
- **dtype** (*type*) – the type of the array. Default is numpy.double

array

nspace

nv

shape

size

__init__(*nv*, *gspace_size*, *vmax*, *mpi_topo*, *dtype=<class 'numpy.float64'>*, *gpu_support=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(nv, gspace_size, vmax, mpi_topo[, ...])	Initialize self.
`generate`()	generate periodic conditions functions for loo.py backend.
`reshape`()	reshape.
`set_conserved_moments`(consm, nv_ptr)	add conserved moments information to have a direct access.
`update`()	update ghost points on the interface with the datas of the neighbors.

Attributes

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.

2.9.3 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 of int*) – number of points in each direction including the fictitious point
- **vmax** (*list of int*) – the size of the fictitious points in each direction
- **mpi_topo** – the mpi topology
- **dtype** (*type*) – the type of the array. Default is numpy.double

array

nspace

nv

shape

size

__init__(*nv*, *gspace_size*, *vmax*, *mpi_topo*, *dtype=<class 'numpy.float64'>*, *gpu_support=False*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(nv, gspace_size, vmax, mpi_topo[, ...])	Initialize self.
`generate`()	generate periodic conditions functions for loo.py backend.
`reshape`()	

Continued on next page

Table 37 – continued from previous page

set_conserved_moments(consm, nv_ptr)	add conserved moments information to have a direct access.
update()	update ghost points on the interface with the datas of the neighbors.

Attributes

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.

2.10 the module bounds

The module bounds contains the classes needed to treat the boundary conditions with the LBM formalism

The classes are

Boundary(domain, dico)	Construct the boundary problem by defining the list of indices on the border and the methods used on each label.
Boundary_method(istore, ilabel, distance, ...)	Set boundary method.
bounce_back(istore, ilabel, distance, ...)	Boundary condition of type bounce-back
anti_bounce_back(istore, ilabel, distance, ...)	Boundary condition of type anti bounce-back
Neumann(istore, ilabel, distance, stencil, ...)	Boundary condition of type Neumann

2.10.1 pylbm.boundary.Boundary

class pylbm.boundary.**Boundary**(*domain*, *dico*)

Construct the boundary problem by defining the list of indices on the border and the methods used on each label.

Parameters

- **domain** (*Domain class*) –
- **dico** (*a dictionary that 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

bv

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.

Type dictionnary

methods

list of boundary methods used in the LBM scheme The list contains Boundary_method instance.

Type list

__init__(*domain*, *dico*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(domain, dico)	Initialize self.

2.10.2 pylbm.boundary.Boundary_method

class pylbm.boundary.**Boundary_method**(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *backend*)
Set boundary method.

Parameters **None** –

feq
the equilibrium values of the distribution function on the border

Type NumPy array

rhs
the additional terms to fix the boundary values

Type NumPy array

distance
distance to the border (needed for Bouzidi type conditions)

Type NumPy array

istore

Type NumPy array

ilabel

Type NumPy array

iload

Type list

value_bc
the prescribed values on the border

Type dictionnary

__init__(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *backend*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(istore, ilabel, distance, stencil, ...)	Initialize self.
fix_iload()	Transpose iload and istore.
move2gpu()	Move arrays needed to compute the boundary on the GPU memory.

Continued on next page

Table 41 – continued from previous page

prepare_rhs(simulation)	Compute the distribution function at the equilibrium with the value on the border.
update(ff)	Update distribution functions with this boundary condition.

2.10.3 pylbm.boundary.bounce_back

class pylbm.boundary.**bounce_back**(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *backend*)
Boundary condition of type bounce-back

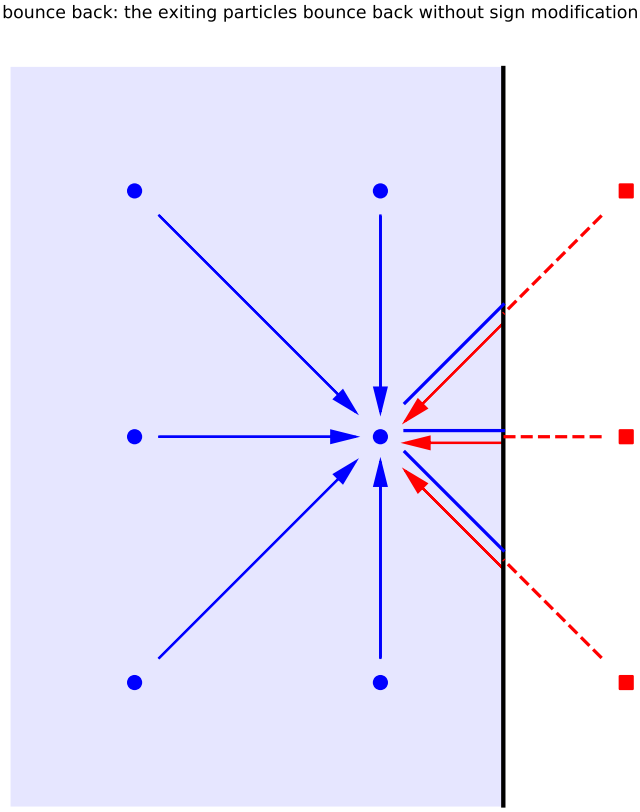
Notes

```
from __future__ import print_function, division
from six.moves import range

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

L = 2
H = 2
color_in = 'b'
color_out = 'r'

fig = plt.figure(1, figsize=(8, 8), facecolor='white')
ax = fig.add_subplot(111, aspect='equal')
ax.plot([0, 0], [-H, H-1], 'k-', linewidth = 2)
ax.add_patch(Rectangle((0, -H), -L, 2*H-1, alpha=0.1, fill=True, color=color_in))
# inner points
mesh_x = np.arange(-L,0) + 0.5
mesh_y = np.arange(-H,H-1) + 0.5
mesh_Y, mesh_X = np.meshgrid(mesh_y, mesh_x)
ax.scatter(mesh_X, mesh_Y, marker='o', color=color_in)
# outer points
mesh_x = np.arange(0,L-1) + 0.5
mesh_y = np.arange(-H,H-1) + 0.5
mesh_Y, mesh_X = np.meshgrid(mesh_y, mesh_x)
ax.scatter(mesh_X, mesh_Y, marker='s', color=color_out)
# inner arrows
e = 0.1
x, y = -0.5, -0.5
for i in [-1,0]:
    for j in [-1,0,1]:
        if i != 0 or j != 0:
            ax.arrow(x+i*(1-e), y+j*(1-e), -i*(1-2*e), -j*(1-2*e),
                     length_includes_head=True,
                     head_width=.5*e,
                     head_length=e,
                     fc=color_in,
                     ec=color_in)
# outer arrows
for j in [-1,0,1]:
    vx = np.array([x+e, x+0.5])
    vy = np.array([y+j*e, y+j*.5])
```

(continues on next page)

(continued from previous page)

```
    ax.plot(vx, vy+.25*(1+.5*abs(j))*e, c=color_in)
    ax.arrow(vx[1], vy[1]-.25*(1+.5*abs(j))*e, -0.5+e, j*(e-.5),
             length_includes_head=True,
             head_width=.5*e,
             head_length=e,
             fc=color_out,
             ec=color_out)
    ax.plot([x+1-e, x+0.5], [y+(1-e)*j,y+.5*j], c=color_out, linestyle='--')
ax.axis('off')
plt.title("bounce back: the exiting particles bounce back without sign
↪modification")
plt.show()
```



__init__(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *backend*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(istore, ilabel, distance, stencil, ...)	Initialize self.
fix_iload()	Transpose iload and istore.
generate(sorder)	Generate the numerical code.
move2gpu()	Move arrays needed to compute the boundary on the GPU memory.
prepare_rhs(simulation)	Compute the distribution function at the equilibrium with the value on the border.
set_iload()	Compute the indices that are needed (symmertic velocities and space indices).
set_rhs()	Compute and set the additional terms to fix the boundary values.
update(ff)	Update distribution functions with this boundary condition.

Attributes

function

2.10.4 pylbm.boundary.anti_bounce_back

class pylbm.boundary.**anti_bounce_back**(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *backend*)
Boundary condition of type anti bounce-back

Notes

```
from __future__ import print_function, division
from six.moves import range

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

L = 2
H = 2
color_in = 'b'
color_out = 'r'

fig = plt.figure(1, figsize=(8, 8), facecolor='white')
ax = fig.add_subplot(111, aspect='equal')
ax.plot([0, 0], [-H, H-1], 'k-', linewidth = 2)
ax.add_patch(Rectangle((0, -H), -L, 2*H-1, alpha=0.1, fill=True, color=color_in))
# inner points
mesh_x = np.arange(-L,0) + 0.5
mesh_y = np.arange(-H,H-1) + 0.5
mesh_Y, mesh_X = np.meshgrid(mesh_y, mesh_x)
```

(continues on next page)

(continued from previous page)

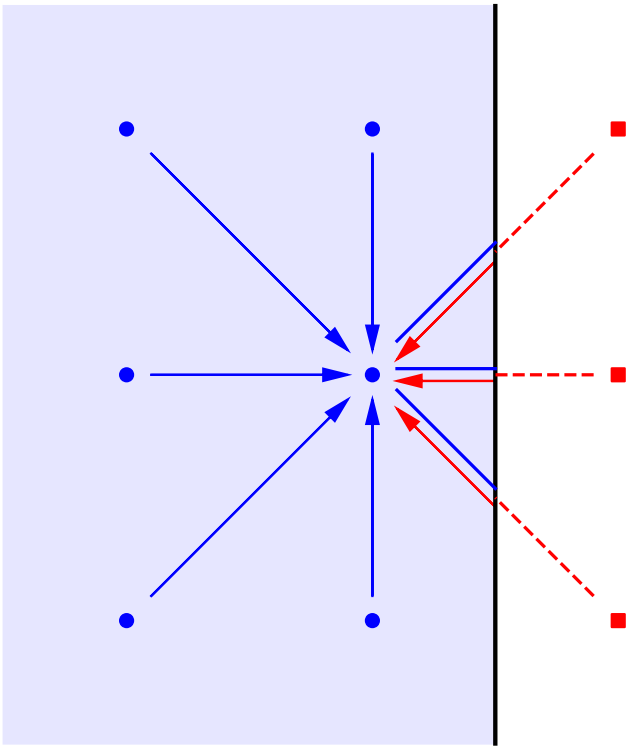
```
ax.scatter(mesh_X, mesh_Y, marker='o', color=color_in)
# outer points
mesh_x = np.arange(0,L-1) + 0.5
mesh_y = np.arange(-H,H-1) + 0.5
mesh_Y, mesh_X = np.meshgrid(mesh_y, mesh_x)
ax.scatter(mesh_X, mesh_Y, marker='s', color=color_out)
# inner arrows
e = 0.1
x, y = -0.5, -0.5
for i in [-1,0]:
    for j in [-1,0,1]:
        if i != 0 or j != 0:
            ax.arrow(x+i*(1-e), y+j*(1-e), -i*(1-2*e), -j*(1-2*e),
                     length_includes_head=True,
                     head_width=.5*e,
                     head_length=e,
                     fc=color_in,
                     ec=color_in)
# outer arrows
for j in [-1,0,1]:
    vx = np.array([x+e, x+0.5])
    vy = np.array([y+j*e, y+j*.5])
    ax.plot(vx, vy+.25*(1+.5*abs(j))*e, c=color_in)
    ax.arrow(vx[1], vy[1]-.25*(1+.5*abs(j))*e, -0.5+e, j*(e-.5),
             length_includes_head=True,
             head_width=.5*e,
             head_length=e,
             fc=color_out,
             ec=color_out)
    ax.plot([x+1-e, x+0.5], [y+(1-e)*j,y+.5*j], c=color_out, linestyle='--')
ax.axis('off')
plt.title("anti bounce back: the exiting particles bounce back with sign␣
↪modification")
plt.show()
```

__init__(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *backend*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(istore, ilabel, distance, stencil, ...)	Initialize self.
fix_iload()	Transpose iload and istore.
generate(sorder)	Generate the numerical code.
move2gpu()	Move arrays needed to compute the boundary on the GPU memory.
prepare_rhs(simulation)	Compute the distribution function at the equilibrium with the value on the border.
set_iload()	Compute the indices that are needed (symmertic velocities and space indices).
set_rhs()	Compute and set the additional terms to fix the boundary values.
update(ff)	Update distribution functions with this boundary condition.

anti bounce back: the exiting particles bounce back with sign modification



Attributes

function

2.10.5 pylbm.boundary.Neumann

class pylbm.boundary.**Neumann**(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *backend*)
Boundary condition of type Neumann

__init__(*istore*, *ilabel*, *distance*, *stencil*, *value_bc*, *nspace*, *backend*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(istore, ilabel, distance, stencil, ...)	Initialize self.
fix_iload()	Transpose iload and istore.
generate(sorder)	Generate the numerical code.
move2gpu()	Move arrays needed to compute the boundary on the GPU memory.
prepare_rhs(simulation)	Compute the distribution function at the equilibrium with the value on the border.
set_iload()	Compute the indices that are needed (symmertic velocities and space indices).
set_rhs()	Compute and set the additional terms to fix the boundary values.
update(ff)	Update distribution functions with this boundary condition.

Attributes

function

CHAPTER

THREE

REFERENCES

CHAPTER

FOUR

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

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z