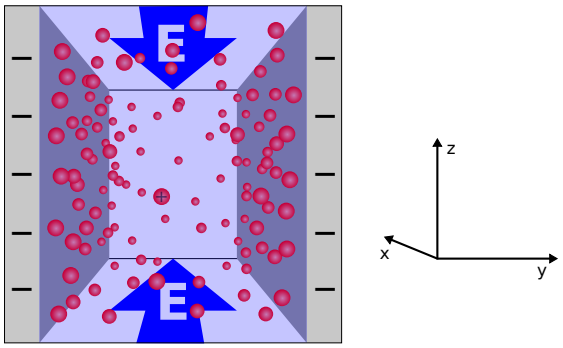
ESPResSo Tutorial

The Lattice Boltzmann Method in ESPResSo: Polymer Diffusion and Electroosmotic Flow

October 5, 2017

Institute for Computational Physics, Stuttgart University



Before you start:
With this tutorial you can get started using the Lattice-Boltzmann method for scientific applications. We give a brief introduction about the theory and how to use it in **ESPResSo**. We have selected three interesting problems for which LB can be applied and which are well understood. You can start with any of them.
The tutorial is relatively long and working through it carefully is work for at least a full day. You can however get a glimpse of different aspects by starting to work on the tasks.
Note: LB can not be used as a black box. It is unavoidable to spend time learning the theory and gaining practical experience.

Contents

1 Introduction

In this tutorial, you will learn basics about the Lattice Boltzmann Method (LBM) with special focus on the application on soft matter simulations or more precisely on how to apply it in combination with molecular dynamics to take into account hydrodynamic solvent effects without the need to introduce thousands of solvent particles.

The LBM – its theory as well as its applications – is still a very active field of research. After almost 20 years of development there are many cases in which the LBM has proven to be fruitful, in other cases the LBM is considered promising, and in some cases it has not been of any help. We encourage you to contribute to the scientific discussion of the LBM because there is still a lot that is unknown or only vaguely known about this fascinating method.

Tutorial Outline

This tutorial should enable you to start a scientific project applying the LB method with **ESPResSo**. In the first part we summarize a few basic ideas behind LB and describe the interface. In the second part we suggest three different classic examples where hydrodynamics are important. These are

- **Hydrodynamic resistance of settling particles.** We measure the drag force of single particles and arrays of particles when sedimenting in solution.
- **Polymer diffusion.** We show that the diffusion of polymers is accelerated by hydrodynamic interactions.
- **Poiseuille flow.** We reproduce the flow profile between two walls.

Notes on the ESPResSo version you will need

With Version 3.1 **ESPResSo** has learned GPU support for LB. We recommend however version 3.3, to have all features available. We absolutely recommend using the GPU code, as it is much (100x) faster than the CPU code.

For the tutorial you will have to compile in the following features: `PARTIAL_PERIODIC`, `EXTERNAL_FORCES`, `CONSTRAINTS`, `ELECTROSTATICS`, `LB_GPU`, `LB_BOUNDARIES_GPU`, `LENNARD_JONES`.

All necessary files for this tutorial are located in the directory `espresso/doc/tutorials/python/04-lattice_boltzmann/scripts`.

2 The LBM in brief

Linearized Boltzmann equation

Here we want to repeat a few very basic facts about the LBM. You will find much better introductions in various books and articles, e.g. [1, 2]. It will however help clarifying our choice of words and we will eventually say something about the implementation in **ESPResSo**. It is very loosely written, with the goal that the reader understands basic concepts and how they are implemented in **ESPResSo**.

The LBM essentially consists of solving a fully discretized version of the linearized Boltzmann equation. The Boltzmann equation describes the time evolution of the one particle distribution function $f(x, p, t)$, which is the probability to find a molecule in a phase space volume $\mathrm{d}x\mathrm{d}p$ at time t.The function f is normalized so that the integral over the whole phase space is the total mass of the particles:

$$\int f(x, p)\, \mathrm{d}x\mathrm{d}p = Nm,$$

where N denotes the particle number and m the particle mass. The quantity $\mathrm{f}(x, p)\, \mathrm{d}x\mathrm{d}p$ corresponds to the mass of particles in this particular cell of the phase space, the population.

Discretization

The LBM discretizes the Boltzmann equation not only in real space (the lattice!) and time, but also the velocity space is discretized. A surprisingly small number of velocities, in 3D usually 19, is sufficient to describe incompressible, viscous flow correctly. Mostly we will refer to the three-dimensional model with a discrete set of 19 velocities, which is conventionally called D3Q19. These velocities, $\vec{c}_i$, are chosen so that they correspond to the movement from one lattice node to another in one time step. A two step scheme is used to transport information through the system: In the streaming step the particles (in terms of populations) are transported to the cell where they corresponding velocity points to. In the collision step, the distribution functions in each cell are relaxed towards the local thermodynamic equilibrium. This will be described in more detail below.

The hydrodynamic fields, the density, the fluid momentum density, the pressure tensor can be calculated straightforwardly from the populations: They correspond to the moments of the distribution function:

$$\rho = \sum f_i \tag{1}$$

$$\vec{j} = \rho\vec{u} = \sum f_i \vec{c}_i \tag{2}$$

$$\Pi^{\alpha\beta} = \sum f_i \vec{c}_i^{\,\alpha} \vec{c}_i^{\,\beta} \tag{3}$$

Here the Greek indices denotes the cartesian axis and the Latin indices indicate the number in the disrete velocity set. Note that the pressure tensor is symmetric. It is easy

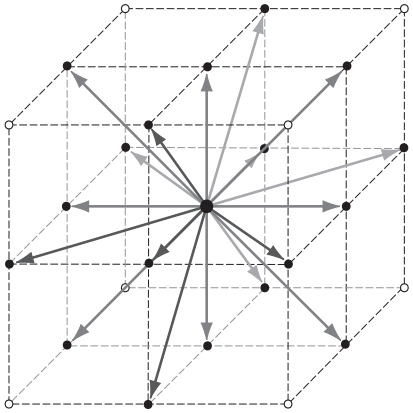


Figure 1: The 19 velocity vectors $\vec{c}_i$ for a D3Q19 lattice. From the central grid point, the velocity vectors point towards all 18 nearest neighbours marked by filled circles. The 19th velocity vector is the rest mode (zero velocity).

to see that these equations are linear transformations of the f_i and that they carry the most important information. They are 10 independent variables, but this is not enough to store the full information of 19 populations. Therefore 9 additional quantities are introduced. Together they form a different basis set of the 19-dimensional population space, the modes space and the modes are denoted by m_i. The 9 extra modes are referred to as kinetic modes or ghost modes. It is possible to explicitly write down the base transformation matrix, and its inverse and in the **ESPResSo** LBM implementation this basis transformation is made for every cell in every LBM step. It is possible to write a code that does not need this basis transformation, but it has been shown, that this only costs 20% of the computational time and allows for larger flexibility.

The second step: collision

The second step is the collision part, where the actual physics happens. For the LBM it is assumed that the collision process linearly relaxes the populations to the local equilibrium, thus that it is a linear (=matrix) operator acting on the populations in each LB cell. It should conserve the particle number and the momentum. At this point it is clear why the mode space is helpful. A 19 dimensional matrix that conserves the first 4 modes (with the eigenvalue 1) is diagonal in the first four rows and columns. Some struggling with lattice symmetries shows that four independent variables are anough to characterize the linear relaxation process so that all symmetries of the lattice are obeyed. Two of them are closely related to the shear and bulk viscosity of the fluid, and two of them do not have a direct physical equivalent. They are just called relaxation rates of the kinetic modes.

The equilibrium distribution to which the populations relax is obtained from maximizing the information entropy $\sum f_i \log f_i$ under the constraint that the density and velocity take their particular instantaneous values.

In mode space the equilbrium distribution is calculated much from the local density and velocity. The kinetic modes 11-19 have the value 0 in equilibrium. The collision operator is diagonal in mode space and has the form

$$m_i^\star = \gamma_i \left(m_i - m_i^{\mathrm{eq}}\right) + m_i^{\mathrm{eq}}.$$

Here $m_i^\star$ is the ith mode after the collision. In words we would say: Each mode is relaxed towards it's equilibrium value with a relaxation rate γ_i. The conserved modes are not relaxed, or, the corresponding relaxation parameter is one.

By symmetry consideration one finds that only four independent relaxation rates are allowed. We summarize them here.

$$\begin{aligned} m_i^\star &= \gamma_i m_i \\ \gamma_1 = \cdots = \gamma_4 &= 1 \\ \gamma_5 &= \gamma_{\mathrm{b}} \\ \gamma_6 = \cdots = \gamma_{10} &= \gamma_{\mathrm{s}} \\ \gamma_{11} = \cdots = \gamma_{16} &= \gamma_{\mathrm{odd}} \\ \gamma_{17} = \cdots = \gamma_{19} &= \gamma_{\mathrm{even}} \end{aligned}$$

To include hydrodynamic fluctuations of the fluid, random fluctuations are added to the nonconserved modes $4 \ldots 19$ on every LB node so that the LB fluid temperature is well defined and the corresponding fluctuation formula, according to the fluctuation dissipation theorem holds. An extensive discussion of this topic is found in [3]

Particle coupling

Particles are coupled to the LB fluid with the force coupling: The fluid velocity at the position of a particle is calculated by a multilinear interpolation and a force is applied on the particle that is proportional to the velocity difference between particle and fluid:

$$\vec{F}_D = -\gamma \left(v - u\right) \tag{4}$$

The opposite force is distributed on the surrounding LB nodes. Additionally a random force is added to maintain a constant temperature, according to the fluctuation dissipation theorem.

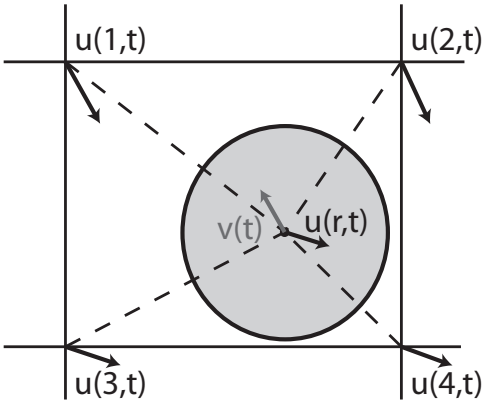


Figure 2: The coupling scheme between fluid and particles is based on the interpolation of the fluid velocity $\vec{u}$ from the grid nodes. This is done by linear interpolation. The difference between the actual particle velocity $\vec{v}(t)$ and the interpolated velocity $\vec{u}(\vec{r}, t)$ is used in the momentum exchange of Equation 4.

3 The LB interface in ESPResSo

ESPResSo features two virtually independent implementations of LB. One implementation uses CPUs and one uses a GPU to perform the computational work. For this, we provide two actor classes `LBFluid` and `LBFluid_GPU` with the module `espressomd.lb`, as well as the optional `LBBoundary` class found in `espressomd.lbboundaries`.

The LB lattice is a cubic lattice, with a lattice constant `agrid` that is the same in all spacial directions. The chosen box length must be an integer multiple of `agrid`. The LB lattice is shifted by 0.5 `agrid` in all directions: the node with integer coordinates $(0, 0, 0)$ is located at $(0.5a, 0.5a, 0.5a)$. The LB scheme and the MD scheme are not synchronized: In one LB time step, several MD steps may be performed. This allows to speed up the simulations and is adjusted with the parameter `tau`. The LB parameter `tau` must be an integer multiple of the MD timestep.

Even if MD features are not used, the System parameters `cell_system.skin` and `time_step` must be set. LB steps are performed in regular intervals, such that the timestep τ for LB is recovered.

Important Notice: All commands of the LB interface use MD units. This is convenient, as e.g. a particular viscosity can be set and the LB time step can be changed without altering the viscosity. On the other hand this is a source of a plethora of mistakes: The LBM is only reliable in a certain range of parameters (in LB units) and the unit conversion may take some of them far out of this range. So note that you always have to assure that you are not messing with that!

One brief example: a certain velocity may be 10 in MD units. If the LB time step is

0.1 in MD units, and the lattice constant is 1, then it corresponds to a velocity of 1 $\frac{a}{\tau}$ in LB units. This is the maximum velocity of the discrete velocity set and therefore causes numerical instabilities like negative populations.

The `LBFluid` class

The `LBFluid` class provides an interface to the LB-Method in the **ESPResSo** core. When initializing an object, one can pass the aforementioned parameters as keyword arguments. Parameters are given in MD units. The available keyword arguments are:

`dens`	The density of the fluid.
`agrid`	The lattice constant of the fluid. It is used to determine the number of LB nodes per direction from `box_l`. *They have to be compatible.*
`visc`	The kinematic viscosity
`tau`	The time step of LB. It has to be an integer multiple of `System.time_step`.
`fric`	The friction coefficient γ for the coupling scheme.
`ext_force`	An external force applied to every node. This is given as a list, tuple or array with three components.

Using these arguments, one can initialize an LBFluid object. This object then needs to be added to the systen's actor list. The code below provides a minimum example.

```
from espressomd import system, lb

# initialize the System and set the necessary MD parameters
 for LB.
s = system.System()
s.box_l = [31, 41, 59]
s.time_step = 0.01
s.cell_system.skin = 0.4

# Initialize and LBFluid with the minimum set of valid
 parameters.
lbf = lb.LBFluid(agrid = 1, dens = 10, visc = .1, tau = 0.01)
# Activate the LB by adding it to the System's actor list.
s.actor.add(lbf)
```

Note: The same applies for the class `LBFluid_GPU`.

Sampling data from a node

The `LBFluid` class also provides a set of methods which can be used to sample data from the fluid nodes. For example

```
lbf[X,Y,Z].quantity
```

returns the quantity of the node with (X, Y, Z) coordinates. Note that the indexing in every direction starts with 0. The possible properties are:

`velocity`	the fluid velocity (list of three floats)
`pi`	the stress tensor (list of six floats: Π_{xx}, Π_{xy}, Π_{yy}, Π_{xz}, Π_{yz}, Π_{zz})
`pi_neq`	the nonequilbrium part of the stress tensor, components as above.
`population`	the 19 populations of the D3Q19 lattice.
`boundary`	The boundary index.
`density`	the local density.

The `LBBoundary` class

The `LBBoundary` class represents a boundary on the `LBFluid` lattice. It is dependent on the classes of the module `espressomd.shapes` as it derives its geometry from them. For the initialization, the arguments `shape` and `velocity` are supported. The `shape` argument takes an object from the `shapes` module and the `velocity` argument expects a list, tuple or array containing 3 floats. Setting the `velocity` will results in a slip boundary condition.

Note that the boundaries are not constructed through the periodic boundary. If, for example, one would set a sphere with its center in one of the corner of the boxes, a sphere fragment will be generated. To avoid this, make sure the sphere, or any other boundary, fits inside the original box.

This part of the LB implementation is still experimental, so please tell us about your experience with it. In general even the simple case of no-slip boundary is still an important research topic in the lb community and in combination with point particle coupling not much experience exists. This means: Do research on that topic, play around with parameters and figure out what happens.

Boundaries are initialized by passing a shape object to the `LBBoundary` class. One way to initialize a wall is:

```
from espressomd import lbboundaries
wall = lbboundaries.LBBoundary(shape=shapes.Wall(normal=[
 1,0,0], dist=1), velocity = [0, 0, 0.01])
```

```
s.lbboundaries.add(wall)
```

Note that all used variables are inherited from previous examples. This will create a wall a surface normal of $(1, 0, 0)$ at a distance of 1 from the origin of the coordinate system in direction of the normal vector. The wall exhibits a slip boundary condition with a velocity of $(0, 0, 0.01)$. For the a no-slip condition, leave out the velocity argument or set it so zero. Please refer to the user guide for a complete list of constraints.

Currently only the so called *link bounce back* method is implemented, where the effective hydrodynamic boundary is located midway between two nodes. This is the simplest and yet a rather effective approach for boundary implementation. Using the shape objects distance function, the nodes determine once during initialisation whether they are boundary or fluid nodes.

4 Drag force on objects

As a first test, we measure the drag force on different objects in a simulation box. Under low Reynolds number conditions, an object with velocity $\vec{v}$ experiences a drag force $\vec{F}_{\mathrm{D}}$ proportional to the velocity:

$$\vec{F}_{\mathrm{D}} = -\gamma\vec{v},$$

where γ is denoted the friction coefficient. In general γ is a tensor thus the drag force is generally not parallel to the velocity. For spherical particles the drag force is given by Stokes' law:

$$\vec{F}_{\mathrm{D}} = -6\pi\eta a\vec{v},$$

where a is the radius of the sphere.

In this task you will measure the drag force on falling objects with LB and **ESPResSo**. In the sample script `lb_stokes_force.py` a spherical object at rest is centered in a square channel. Bounce back boundary conditions are assumed on the sphere. At the channel boundary the velocity is fixed by using appropriate boundary conditions. Within a few hundred or thousand integration steps a steady state develops and the force on the sphere converges.

Radius dependence of the drag force

Measure the drag force for three different input radii of the sphere. How good is the agreement with Stokes' law? Calculate an effective radius from Stokes' law and the drag force measured in the simulation. Is there a clear relation to the input radius? Remember how the bounce back boundary condition work and how good spheres can be represented by them.

Visualization of the flow field

The script produces `vtk` files of the flow field. Visualize the flow field with `paraview`. Open `paraview` by typing it on the command line. Make sure you are in the folder where the files are located. So the agenda is:

- Click in the menu `File`, `Open...`
- Choose the files with flow field `fluid...vtk`
- Click `Apply`
- Add a stream tracer filter `Filters`, `Alphabetical`, `Stream tracer`
- Change the seed type from `point source` to `high resolution line source`
- Click `Apply`
- Rotate the visualization box to see the stream lines.
- Use the play button in the bar below the menu bar to show the time evolution.

System size dependence

Measure the drag force for a fixed radius but varying system size. Does the drag force increase of decrease with the system size? Can you find a qualitative explanation?

5 Polymer Diffusion

In these exercises we want to use the LBM-MD-Hybrid to reproduce a classic result of polymer physics: The dependence of the diffusion coefficient of a polymer on its chain length. If no hydrodynamic interactions are present, one expects a scaling law $D \propto N^{-1}$ and if they are present, a scaling law $D \propto N^{-\nu}$ is expected. Here ν is the Flory exponent that plays a very prominent role in polymer physics. It has a value of $\sim 3/5$ in good solvent conditions in 3D. Discussions of these scaling laws can be found in polymer physics textbooks like [4–6].

The reason for the different scaling law is the following: When being transported, every monomer creates a flow field that follows the direction of its motion. This flow field makes it easier for other monomers to follow its motion. This makes a polymer long enough diffuse more like compact object including the fluid inside it, although it does not have clear boundaries. It can be shown that its motion can be described by its hydrodynamic radius. It is defined as:

$$\langle \frac{1}{R_h} \rangle = \langle \frac{1}{N^2} \sum_{i \neq j} \frac{1}{|r_i - r_j|} \rangle \tag{5}$$

This hydrodynamic radius exhibits the scaling law $R_h \propto N^\nu$ and the diffusion coefficient of long polymer is proportional to its inverse. For shorter polymers there is a transition region. It can be described by the Kirkwood-Zimm model:

$$D = \frac{D_0}{N} + \frac{k_B T}{6\pi\eta}\langle\frac{1}{R_h}\rangle \tag{6}$$

Here D_0 is the monomer diffusion coefficient and η the viscosity of the fluid. For a finite system size the second part of the diffusion is subject of a $1/L$ finite size effect, because hydrodynamic interactions are proportional to the inverse distance and thus long ranged. It can be taken into account by a correction:

$$D = \frac{D_0}{N} + \frac{k_B T}{6\pi\eta}\langle\frac{1}{R_h}\rangle\left(1 - \langle\frac{R_h}{L}\rangle\right) \tag{7}$$

It is quite difficult to prove this formula with good accuracy. It will need quite some computer time and a careful analysis. So please don't be too disappointed if you don't manage to do so.

We want to determine the diffusion coefficient from the mean square distance that a particle travels in the time t. For large t it is be proportional to the time and the diffusion coefficient occurs as prefactor:

$$\frac{\partial\langle r^2(t)\rangle}{\partial t} = 2dD. \tag{8}$$

Here d denotes the dimensionality of the system, in our case 3. This equation can be found in virtually any simulation textbook, like [7]. We will therefore set up a polymer in an LB fluid, simulate for an appropriate amount of time, calculate the mean square displacement as a function of time and obtain the diffusion coefficient from a linear fit. However we make a couple of steps in between and divide the full problem into subproblems that allow to (hopefully) fully understand the process.

5.1 Step 1: Diffusion of a single particle

Our first step is to investigate the diffusion of a single particle that is coupled to an LB fluid by the point coupling method. Take a look at the script `single_particle_diffusion.py`. The script takes the LB-friction coefficient as an argument. Start with an friction coefficient of 1.0:

```
/path/to/pypresso single_particle_diffusion.py 1.0
```

In this script an LB fluid and a single particle are created and thermalized. The random forces on the particle and within the LB fluid will cause the particle to move. The mean squared displacement is calculated during the simulation via a multiple-tau correlator. Run the simulation script and plot the output data `msd_1.0.dat`. To load the file into a numpy array, one can use `numpy.loadtxt`. Zoom in on the origin of the plot. What do you see for short times? What do you see on a longer time scale? Produce a double-logarithmic plot to assess the power law.

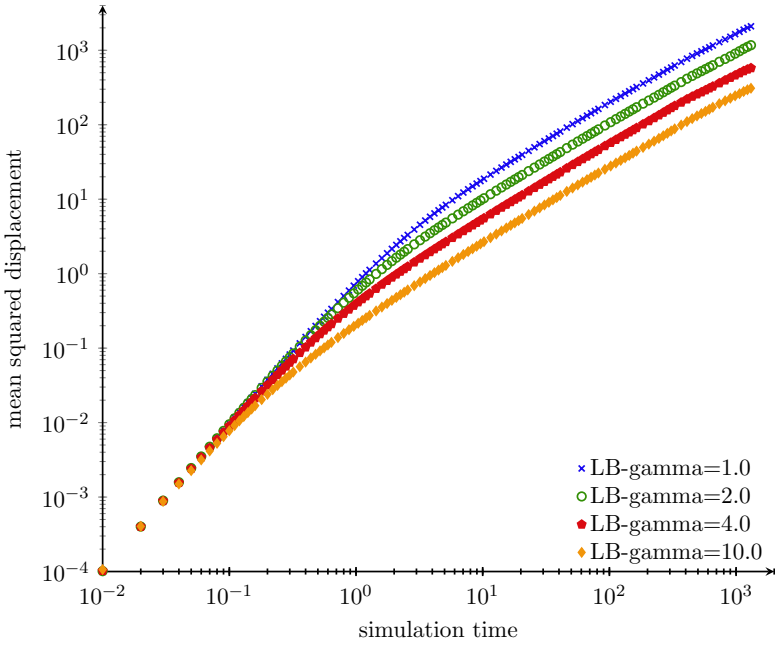


Figure 3: Mean squared displacement of a single particle for different values of LB friction coefficient.

Can you give an explanation for the quadratic time dependency for short times? Use the function `curve_fit` from the module `scipy.optimize` to produce a fit for the linear regime and determine the diffusion coefficient.

Run the simulation again with different values for the friction coefficient, e.g. 1. 2. 4. 10. Calculate the diffusion coefficient for all cases and plot them as a function of γ. What relation do you observe?

5.2 Step 2: Diffusion of a polymer

One of the typical applications of **ESPResSo** is the simulation of polymer chains with a bead-spring-model. For this we need a repulsive interaction between all beads, for which one usually takes a shifted and truncated Lennard-Jones (so called Weeks-Chandler-Anderson) interaction, and additionally a bonded interaction between adjacent beads to hold the polymer together. You have already learned that the command

```
system.non_bonded_inter[0,0].lennard_jones.set_params(
    epsilon = 1.0, sigma = 1.0,
    shift = 0.25, cutof = 1.226)
```

creates a Lennard-Jones interaction with $\varepsilon = 1.$, $\sigma = 1.$, $r_{\text{cut}} = 1.125$ and $\varepsilon_{\text{shift}} = 0.25$ between particles of type 0, which is the desired repulsive interaction. The command

```
from espressomd import interactions
fene = interactions.FeneBond(k = 7,d_r_max = 2)
```

creates a `FeneBond` object (see **ESPResSo** manual for the details). Still **ESPResSo** does not know between which beads this interaction should be applied. This can be either be specified explicitly or done with the `polymer` module. This creates a given number of beads, links them with the given bonded interaction and places them following a certain algorithm. We will use the pruned self-avoiding walk: The monomers are set according to a pruned self-avoiding walk (in 3D) with a fixed distance between adjacent bead positions. The syntax is:

```
from espressomd import polymer
# mpc: monomers per chain
mpc = 30
poly = polymer.Polymer(N_P=1, MPC = mpc, bond=fene,
 bond_length = 1)
```

Using a random walk to create a polymer causes trouble: The random walk may cross itself (or closely approach itself) and the LJ potential is very steep. This would raise the potential energy enormously and would make the monomers shoot through the simulation box. The pruned self-avoiding walk should prevent that, but to be sure we perform some MD steps with a capped LJ potential, this means forces above a certain threshold will be set to the threshold in order to prevent the system from exploding. To see how this is done, look at the script `polymer_diffusion.py`. It contains a quite long warmup command so that also longer polymers are possible. You can probably make it shorter.

It is called in the following way:

```
/path/to/pypresso polymer_diffusion.py $N_monomers
```

This allows to quickly change the number of monomers without editing the script. For the warmup a Langevin thermostat is used to keep the temperature constant. Furthermore we want to compute the diffusion constant of the polymer for different numbers of monomers. For this purpose we can again use the multiple tau correlator. Have a look at the **ESPResSo** -script for the single particle diffusion and add the adapted commands for the polymer. Find out how many integration steps are necessary to capture the long-time diffusion regime of the polymer. The script already computes the time averaged

hydrodynamic radius and stores it in a file `rh_nom_xx.dat` where `xx` is the number of monomers.

Run the script for different numbers of monomers and determine the evolution of the diffusion coefficient as a function of the chain length. Compare the results of your **ESPResSo** simulations with the given Kirkwood-Zimm formula (eq. 7).

6 Poiseuille flow ESPResSo

Poisseuille flow is the flow through a pipe or (in our case) a slit under a homogenous force density, e.g. gravity. In the limit of small Reynolds numbers, the flow can be described with the Stokes equation. We assume the slit being infinitely extended in y and z direction and a force density f on the fluid in y direction. No slip-boundary conditions (i.e. $\vec{u} = 0$) are located at $z = \pm l/2$. Assuming invariance in y and z direction and a steady state the Stokes equation is simplified to:

$$\eta \partial_x^2 u_y = f \tag{9}$$

where f denotes the force density and η the dynamic viscosity. This can be integrated twice and the integration constants are chosen so that $u_y = 0$ at $z = \pm l/2$ and we obtain:

$$u_y = \frac{f}{2\eta}\left(l^2/4 - x^2\right) \tag{10}$$

With that knowledge investigate the script `poisseuille.py`. Note the use of the `lbboundaries` module. Two walls are created with normal vectors $(\pm 1, 0, 0)$. An external force is applied to every node. After 5000 LB updates the steady state should be reached.

Task: Write a loop that prints the fluid velocity at the nodes (0,0,0) to (16,0,0) and the node position to a file. Use the `lbf.[node].quantity` method for that. Hint: to write to a file, first open a file and then use the `write()` method to write into it. Do not forget to close the file afterwards. Example:

```
f = open("file.dat", "w")
f.write("Hello world!\n")
f.close()
```

Use the data to fit a parabolic function. Can you confirm the analytic solution?

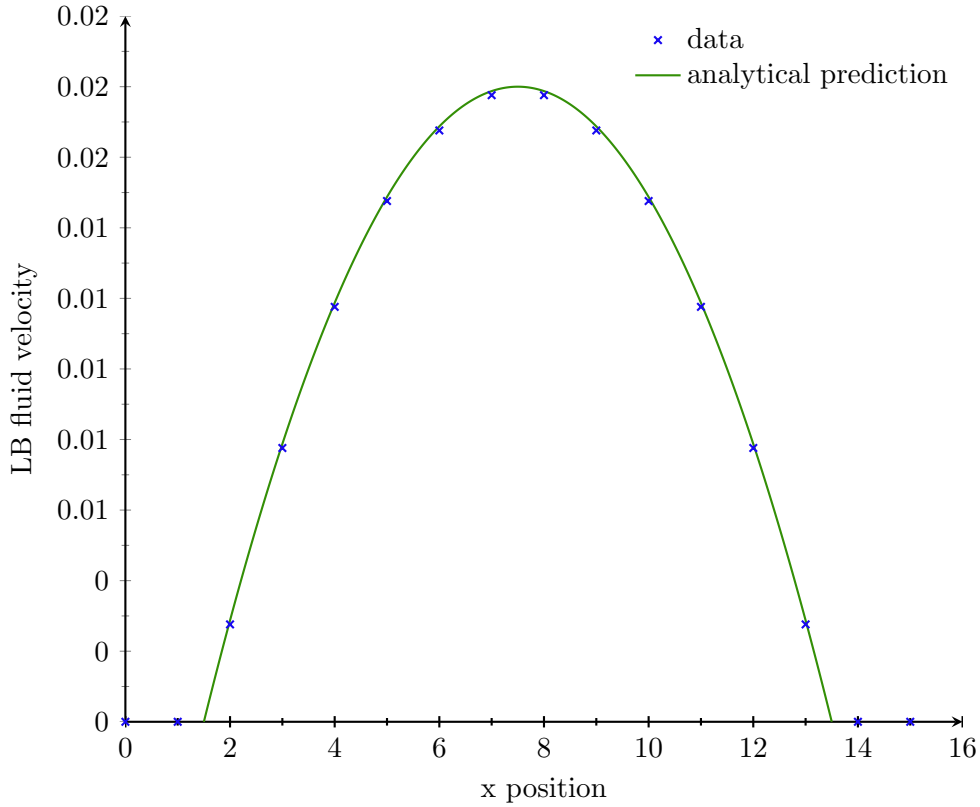


Figure 4: Poisseuille Flow in a slit Geometry.

References

[1] S Succi. *The lattice Boltzmann equation for fluid dynamics and beyond*. Clarendon Press, Oxford, 2001.

[2] B. Dünweg and A. J. C. Ladd. *Advanced Computer Simulation Approaches for Soft Matter Sciences III*, chapter II, pages 89–166. Springer, 2009.

[3] B. Dünweg, U. Schiller, and A.J.C. Ladd. Statistical mechanics of the fluctuating lattice-boltzmann equation. *Phys. Rev. E*, 76:36704, 2007.

[4] P. G. de Gennes. *Scaling Concepts in Polymer Physics*. Cornell University Press, Ithaca, NY, 1979.

[5] M. Doi. *Introduction do Polymer Physics*. Clarendon Press, Oxford, 1996.

[6] Michael Rubinstein and Ralph H. Colby. *Polymer Physics*. Oxford University Press, Oxford, UK, 2003.

[7] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation*. Academic Press, San Diego, second edition, 2002.