Openlb Software Lab - Exercise Protocol

Jousef Murad
jousef.m@googlemail.com

May 2018

Contents

1 Introduction

In the last three decades, the **Lattice Boltzmann Method** (LBM) became a promising alternative for the simulation of fluid flows. Flows through porous media, multi-phase and multi-component flows with and without heat transfer as well as flows around complex geometries have been investigated by several people.

With LBM the computational domain is discretized by an equidistant mesh, on which a discrete set of velocity distribution functions is solved numerically. This set of velocity distribution functions corresponds to discrete lattice velocities, which are used to recover the macroscopic moments in terms of a Hermite Polynomial expansion.[1]

The Lattice Boltzmann Method along with the **Bhatnagar–Gross–Krook approximation (BGK)** model as a collision operator has reached a big success in fluid simulations. It basically considers all the possible outcomes of two-particle collisions for any choice of intermolecular forces.[2] Using this model has many advantages such as application of boundary condition in complex geometries, reduction of running time and simple parallelism in computations. Unfortunately this model suffers from numerical instabilities and limitations of usage for high Reynolds numbers. These problems cause that in some cases the so called **MRT (Multiple-Relaxation-Time)** model is used. In this study only the standard BGK model for the Lid Driven Cavity as well as the Channel flow around different objects has been used. The numerical instablilty will be addressed in the first exercise by increasing the Reynolds number and respectively changing parameter to make sure the simulations will not diverge.

In the following report several cases are going to be investigated including a **Lid Driven Cavity** and **Channel Flow around a Cylinder**. Changes in the source code will be performed and the effects are investigated in the post-processor software **Paraview** and conclusions will be derived from the terminal output window such as the pressure drop. The main purpose of this study is to get a deeper knowledge and to grasp the main concepts of the LBM. The report will include the following points:

1. Mathematical problem formulations of the cases
2. Description of the test cases
3. Plots and tables of computed solutions and other relevant values

2 The Lattice Boltzmann Method

In this section a small overview of the Lattice Boltzmann Method (LBM) and the fundamental concepts will be pointed out.

In the context of fluid dynamics, one can either describe the length scales as **microscopic**, **mesoscopic** or **macroscopic** as depicted in the figure below.

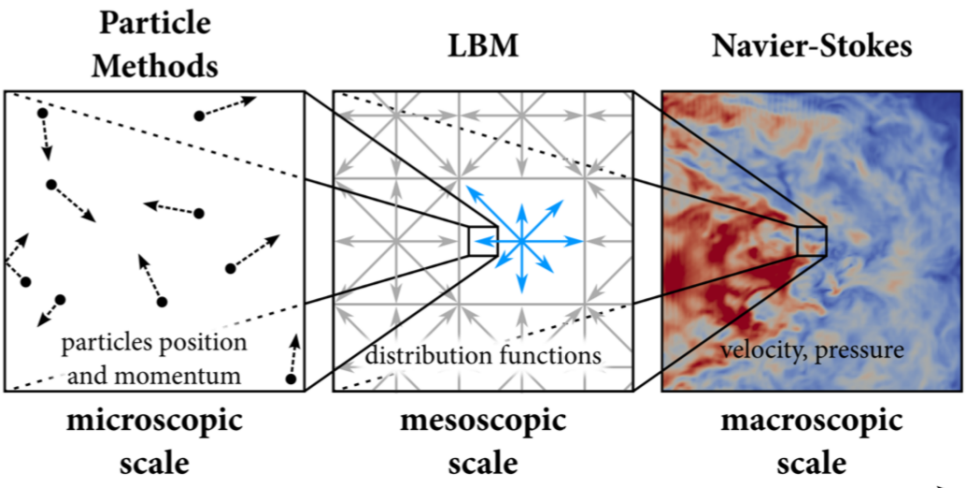

Figure 1: Hierarchy of length scales [5]

Microscopic systems are therefore governed by Newton's dynamics, while the Navier-Stokes Equation is the governing equation for a fluid continuum where in between there is the so called and already mentioned **mesoscopic** description which does not track individual fluid particles/molecules. We speak about tracking so called *distributions* or *representative collections* of molecules. The LBM is exactly based on this principle which is covered by the **kinetic theory.** [3]

In the kinetic theory the most important variable is the particle **distribution function** $f(\mathbf{x}, \xi, t)$. It can be interpreted as some kind of density ρ which takes into account the microscopic *particle velocity* ξ. The difference between the density $\rho(\mathbf{x}, t)$ and the distribution function $f(\mathbf{x}, \xi, t)$ is that the normal density represents the density of mass in **physical space** where the function takes into account the **threedimensional physical** and **velocity** space.

On top of that the distribution function f is connected to the density ρ and fluid velocity u which are macroscopic variables from its **moments** which are defined as integrals of the function f and their weight ξ over the complete velocity space. A detailed formulation of the macroscopic **mass density**, **momentum density**, **total energy density**, **internal energy density** and **relative velocity** can be found in [3]

Before writing the whole **Boltzmann equation** the total derivative for the function f has to be defined since it it a function of the position $\mathbf{x}$, particle velocity ξ and the time $\mathbf{t}$ the total derivative with respect to the time $\mathbf{t}$ must be

$$\frac{df}{dt} = \left(\frac{\partial f}{\partial t}\right)\frac{dt}{dt} + \left(\frac{\partial f}{\partial x_\beta}\right)\frac{dx_\beta}{dt} + \left(\frac{\partial f}{\partial \xi_\beta}\right)\frac{\xi_\beta}{dt} \tag{1}$$

Rearranging the terms and writing $\frac{df}{dt}$ as $\Omega(f)$ for the total differential, we get to the well known Boltzmann equation

$$\frac{\partial f}{\partial t} + \xi_\beta \frac{\partial f}{\partial x_\beta} + \frac{F_\beta}{\rho} \frac{\partial f}{\partial \xi_\beta} = \Omega(f) \tag{2}$$

which can be seen as an "advection equation where the first two terms describe the advection of the distribution functions with the velocity ξ of its particles. The third term represents forces affecting this velocity. The right-hand side is the source term which usually represents the local redistribution of f due to collisions and is called the **collision operator**."[3]

The collision operator used in these studies is the so called **BGK collision operator** which was named after the inventors Bhatnagar, Gross and Krook.

$$\Omega(f) = -\frac{1}{\tau}(f - f^{eq}) \tag{3}$$

The expression given above captures the relaxation of the distribution function towards the equilibrium distribution. τ is a time constant which is known as the **relaxation time** describing the **speed of equilibration**.

With the BGK-Boltzmann collision operator, the Boltzmann equation now reads

$$\frac{\partial f}{\partial t} + \xi_\beta \frac{\partial f}{\partial x_\beta} + \frac{F_\beta}{\rho} \frac{\partial f}{\partial \xi_\beta} = -\frac{1}{\tau}(f - f^{eq}) \tag{4}$$

Discretising the Boltzmann equation in velocity space, physical space and time we get the **Lattice Boltzmann equation**

$$f_i(\mathbf{x} + c_i \Delta t, t + \Delta t) = f_i(\mathbf{x}, t) + \Omega_i(\mathbf{x}, t) \tag{5}$$

"The equation states that particles denoted by $f_i(\mathbf{x}, t)$ move with the velocity c_i to a neighbouring point $\mathbf{x} + c_i \Delta t$ at the following time step $t + \Delta t$ but particles are simultaneously affected by the collision operator Ω_i which models particle collision by redistributing particles among the populations f_i at each side." [3]

3 Exercise 1 - Initialisation of Lid Driven Cavity

In the first exercise a two dimensional lid-driven cavity in a square domain has been investigated and a numerical benchmark for convergence and accuracy analysis was performed.

The two-dimensional flow that has been investigated can be described mathematically in terms of the stream function equation

$$\Psi_{xx} + \Psi_{yy} + \omega = 0 \tag{6}$$

and vorticity equation - with the advective terms expressed in conservative form:

$$\omega_{xx} + \omega_{yy} - Re[(\Psi_y \omega)_x - (\Psi_x \omega)_y] = Re\omega_t \tag{7}$$

The boundary conditions depicted in figure 2 below consist of the zero slip condition at the nonporous walls meaning that Ψ and its normal derivatives vanish at all the boundaries which provides no direction for ω at the walls.[4]

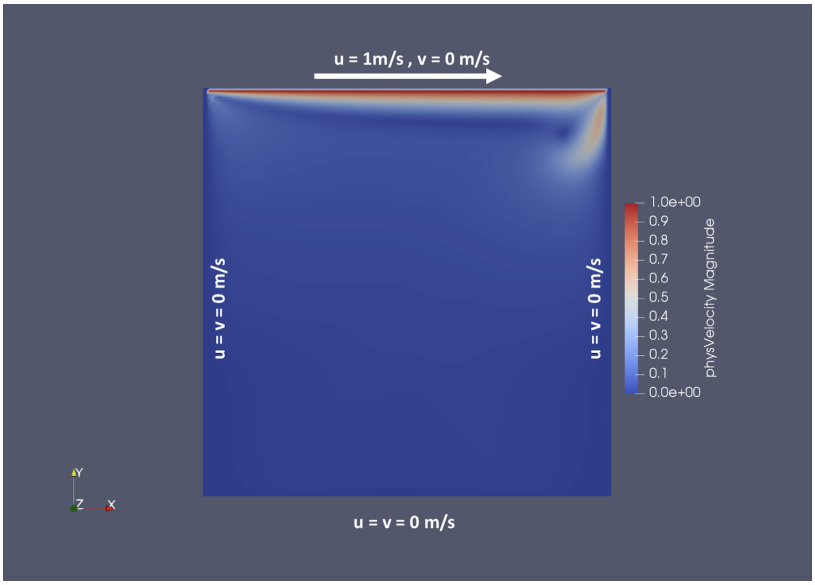

Figure 2: First Exercise - Lid Driven Cavity

The flow was assumed to be steady-state with the Dirichlet boundary conditions

- $u_{lid} = \mathbf{1\ m/s}$
- $u_{sides} = \mathbf{0\ m/s}$

the kinematic viscosity of **0.001** m^2/s, the Reynolds number **1,000** and a wall length of **1 m**.

3.1 Preperation for Parallel Processing

In order to make use of the parallel processing it is necessary to adapt the *config.mk* file inside the OpenLB folder and uncomment the following lines:

$$\begin{aligned} \mathbf{CXX} &:= \mathbf{mpic++} \\ \mathbf{PARALLEL\_MODE} &:= \mathbf{MPI} \end{aligned} \tag{8}$$

3.2 Parallel Run for Re = 1,000

In order to run the test case with a Reynolds number of 1000 it is necessary to switch into the **cavity2d/parallel** folder and to clean the folder first to delete old dependencies. After that the **make** command is used in order to build the executable.

To run the parallel processing of the case the following command has been typed into the terminal:

$$\textbf{mpirun -np 2 cavity2d} \tag{9}$$

After a successful run the images and Paraview files can be viewed.
Both post-processing items can be found in:

$$\begin{aligned}\textbf{tmp/imageData/}\\ \textbf{tmp/vtkData/}\end{aligned} \tag{10}$$

3.3 Visualization in Paraview

For the post-processing visualization the Open Source software Paraview has been used. The **.pvd** which is located in **tmp/vtkData/** has to be chosen and loaded into Paraview. A click on Apply activates the mode and in order to visualize the velocity the **Surface** and **physVelocity** mode will be used. Clicking **Play** will show an animation of the simulation.

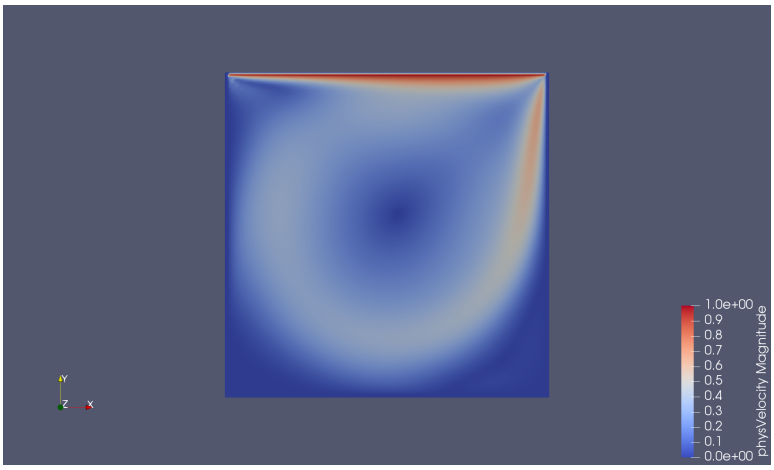

Figure 3: Results for a simulation with Re = 1000 - Last Timestep

3.4 Run for Re = 100

In order to change the Reynolds number the **xml** file has to be changed. For that the **cavity2d.xml** which is located in the root directory of the example has to be edited. The kinematic viscosity will be changed to **0.01** in order to get a Reynolds Number of 100.

$$\langle\ \textbf{PhysViscosity}\rangle\textbf{0.01}\langle\textbf{/PhysViscosity}\rangle \tag{11}$$

The following relation applies for the kinematic viscosity physViscosity:

$$physViscosity = \frac{charPhysLength \cdot charPhysVelocity}{Re} \tag{12}$$

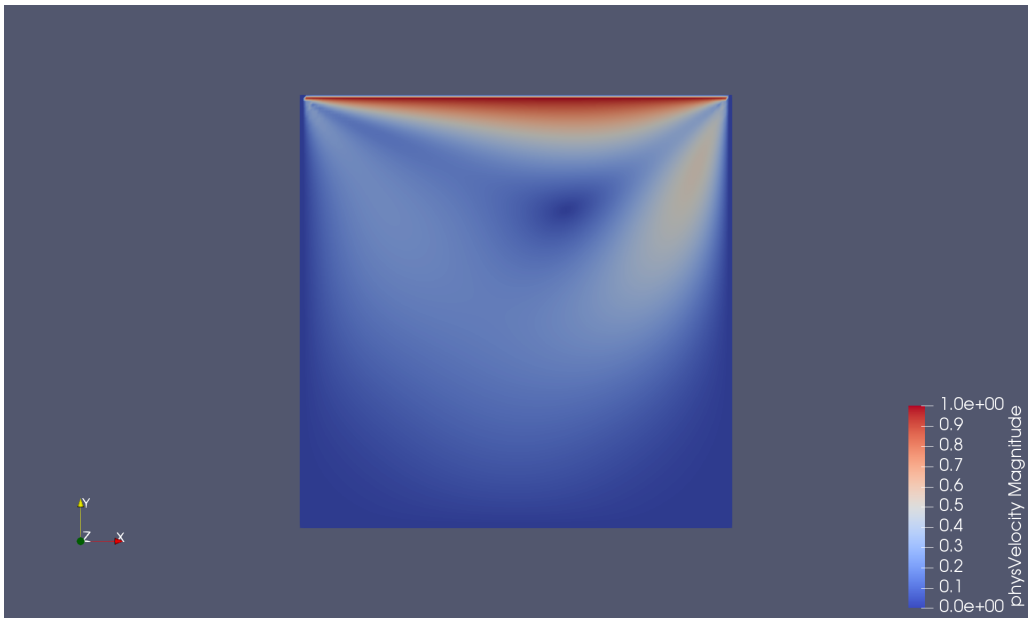

Figure 4: Results for a simulation with Re = 100 - Last Timestep

3.5 Run for Re = 10,000

The last exercise of this chapter was to increase the Reynolds Number to **10,000** and to adapt the kinematic viscosity accordingly. For that the old executables were deleted and a new parallel run has been initiated by using the following order of commands:

$$\begin{aligned} &\textbf{make clean} \\ &\textbf{make} \\ &\textbf{mpirun -np 2 cavity2d} \end{aligned} \tag{13}$$

The problem in this case was that the simulation **diverged** which could also be seen in the terminal as the energy and the average density were increasing exorbitantly and in addition to that the run terminated autonomously. To fix this issue a new discretisation parameter called **physDeltaX** (denoted with **N**) had to be calculated for this Reynolds number of **10,000** and a **relaxation time** τ of **0.51**. The following formula has been used to determine the new **N**:

$$N > Re\frac{\tau - 0.5}{3 \cdot u_{lb}} \tag{14}$$

where u_{lb} is the so called **charLatticeVelocity** and is usually smaller or equal to **0.1**. There are two possibilities to choose this parameter.

- For the fastest and stable simulation $u_{lb} \leq 0.1$
- For a more accurate (less compressible) simulation $u_{lb} = 0.2$

In the end **charLatticeVelocity** was assumed to be **0.1** which will lead to the following form of equation (14)

$$N > Re\frac{0.51 - 0.5}{3 \cdot 0.1} \approx 333 \tag{15}$$

This means that at least a resolution of 334 had to be chosen in order to overcome the diverging simulation.

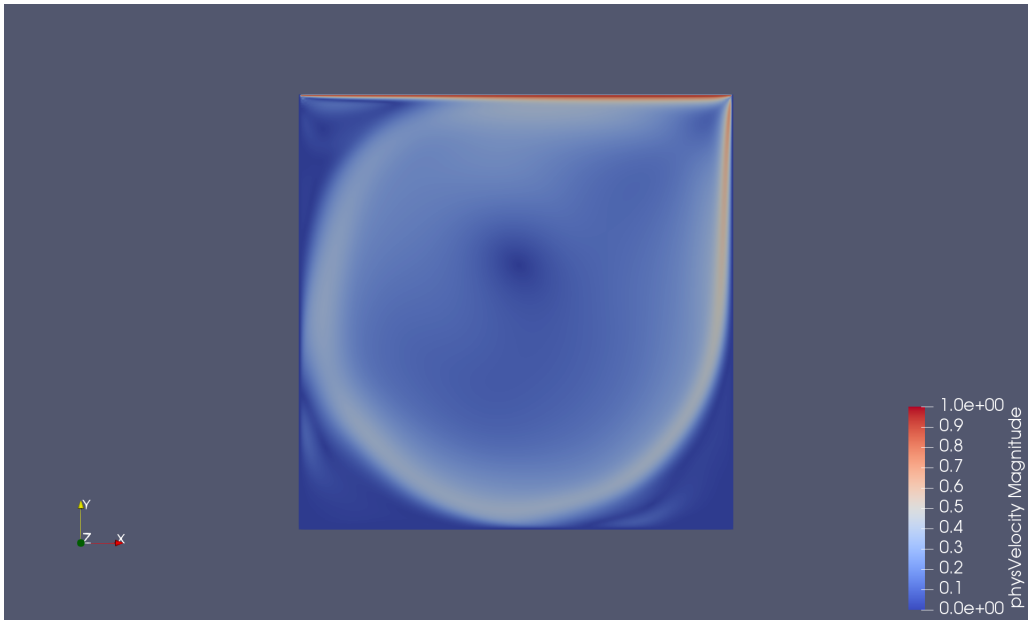

Figure 5: Results for a simulation with Re = 10,000 - Last Timestep

4 Exercise 2 - Geometry for Channel Flow around a Cylinder

The second exercise was dealing with the flow around a cylinder which is commonly as the so called **Krmn vortex street**. For this purpose a domain with the height of **0.41 m + 1 cell** and length of **2.2 m**. The obstacle (a cylinder) is positioned at **x = 0.2 m** and **y = 0.2 m** with a radius of **0.05 m**. The steady flow has the following properties:

- Re = 20
- u = 0.2 m/s

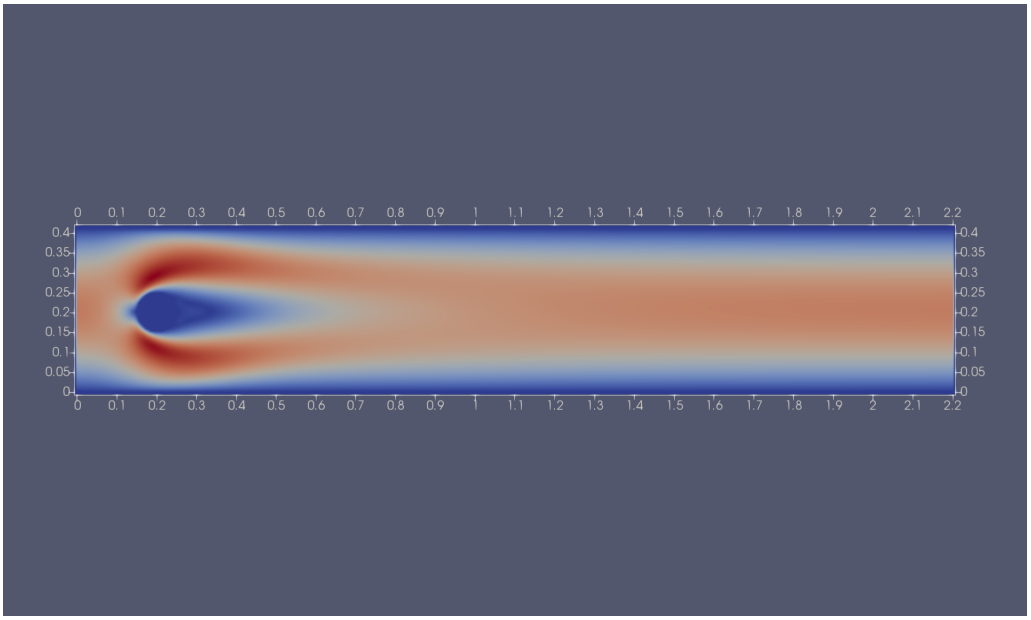

Figure 6: Second exercise - Flow Around A Cylinder

Above the project has been described mathematically with its dimensions. Additionally expressions for the lift and drag coefficient as well as the pressure drop shall be given in order to fully describe the problem and the parameters that are calculated and printed in the terminal.

We start with the equations for the lift forces and the corresponding lift coefficient:

$$F_D = \int_S (\rho v \frac{\partial v_t}{\partial n} n_y - P n_x) dS \tag{16}$$

$$c_d = \frac{2F_w}{\rho \overline{U}^2 D} \tag{17}$$

Accordingly the equations for the drag forces and the corresponding drag coefficient will be given:

$$F_L = -\int_S (\rho v \frac{\partial v_t}{\partial n} n_x - P n_y) dS \tag{18}$$

$$c_d = \frac{2F_a}{\rho \overline{U}^2 D} \tag{19}$$

with the circle S, the normal vector n on S with the x and y component n_x & n_y, the tangential vector v_t on S and so called tangent vector which is defined as $t = (n_y, -n_x)$.

4.1 Investigation of Pressure Drop

The first task was to run a simulation with the default setup and again using the **make** command and start the parallel execution with **mpirun -np 2 ./cylinder2d**.

The terminal will output show the pressure drop accordingly.

```
[getResults] pressure1=0.13189; pressure2=0.0135893; pressureDrop=0.118301
```

Listing 1: Pressure Drop from the terminal

The terminal excerpt shows the pressure close to the inlet/outlet as well as the pressure difference between both boundaries.

In order to familiarize with the functionalities of the *openlb* code the radius of the cylinder was changed.

```
const T radiusCylinder = 0.05;
```

Listing 2: Standard dimensions of the cylinder

```
const T radiusCylinder = 0.1;
```

Listing 3: Changed dimensions of the cylinder

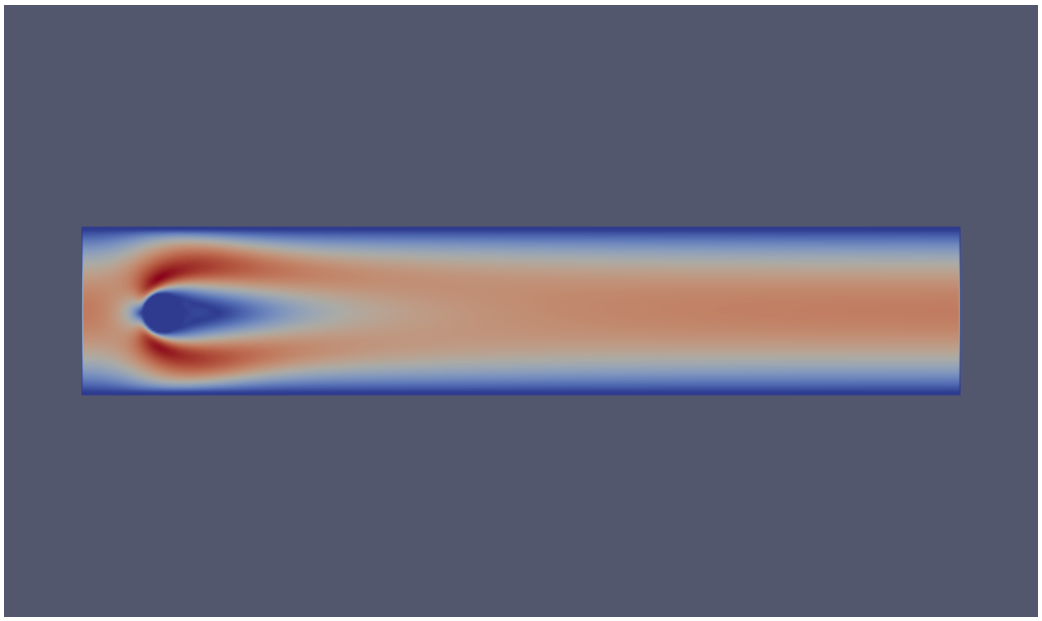

Figure 7: Standard case with a radius of 0.05

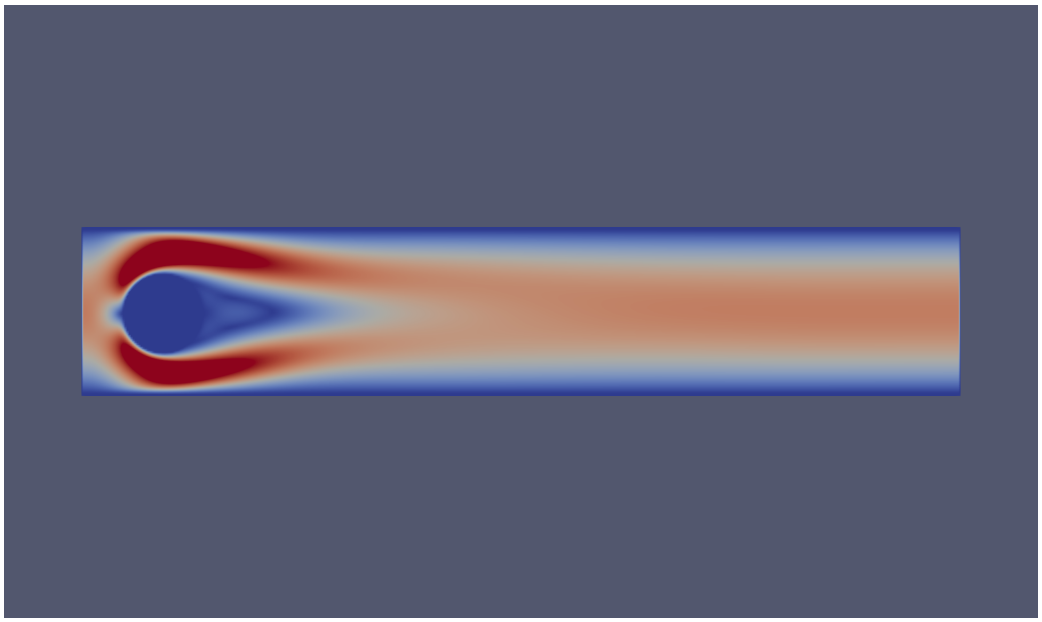

Figure 8: Adapted case with a radius of 0.1

```
[getResults] pressure1=0.215786; pressure2=-0.0176812; pressureDrop
    =0.233467
```

Listing 4: Pressure Drop for a radius of 0.1

It can be seen that the pressure drop doubles when the size of the cylinder is doubled. To verify this observation another run with a radius of 0.15 was performed.

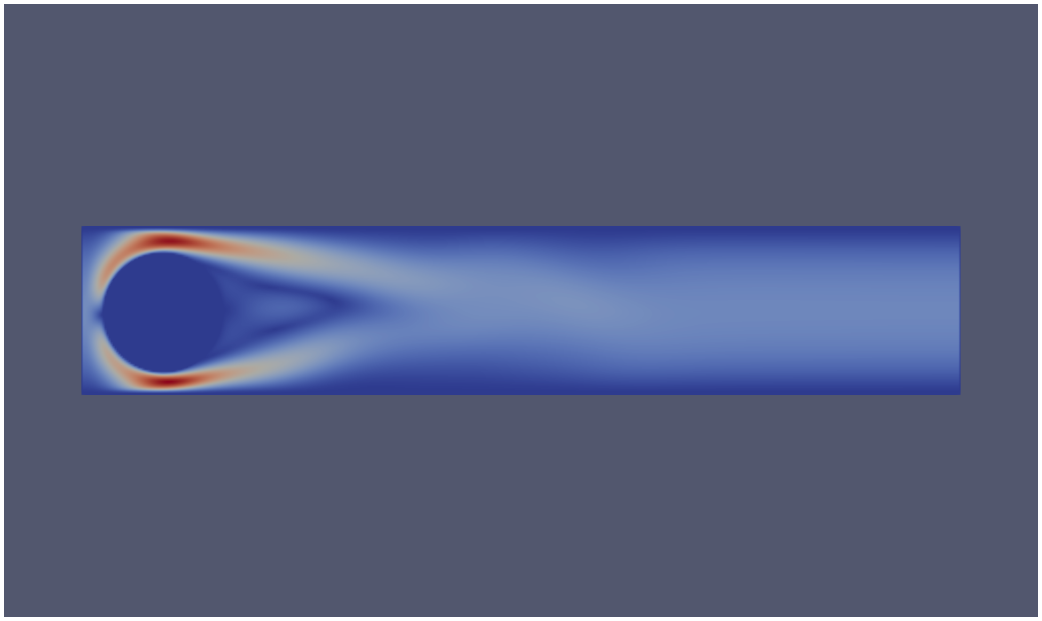

Figure 9: Adapted case with a radius of 0.15

```
[getResults] pressure1=0.829365; pressure2=-0.115429; pressureDrop
    =0.944794
```

Listing 5: Pressure Drop for a radius of 0.15

```
[getResults] pressure1=0.444596; pressure2=0.0279164; pressureDrop
    =0.416679
```

Listing 6: Pressure Drop for a radius of 0.125

```
[getResults] pressure1=4.49595; pressure2=-0.0674475; pressureDrop=4.5634
```

Listing 7: Pressure Drop for a radius of 0.175

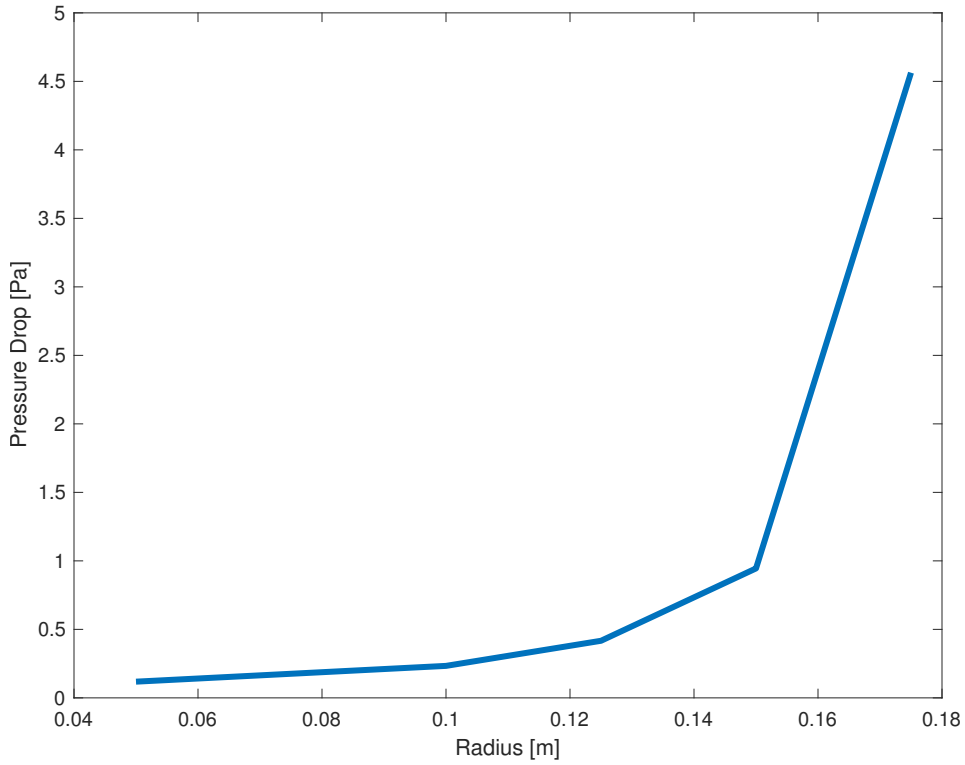

Figure 10: Relation between radius of obstacle and pressure drop

As can be seen from the plot above the relation between the two parameter that were investigated is approximately **parabolic**.

4.2 Exchange Object

In the second task the object inside the fluid domain had to be changed. For that focus had to be put at the functors for the circle as well as the cuboid. Special emphasis has been placed on the following lines:

```
Vector<T,2> center(centerCylinderX, centerCylinderY) ;
IndicatorCircle2D<T> circle(circle, radiusCylinder) ;

superGeometry.rename(1, 5, circle) ;
```

Listing 8: Indicator Functor for the circle

In order to reset the compilation we again use **make clean** here and exchange the **IndicatorCircle2D** in our **cylinder2d.cpp** with the **IndicatorCuboid2D**. For that the following changes in the code have been performed:

```
const T xlength = 0.1;
const T ylength = 0.1;
const T theta = 0;
```

Listing 9: Parameter for the cuboid

```
IndicatorCuboid2D<T> cube(xlength, ylength, center, theta);
```

Listing 10: Cuboid Functor

```
superGeometry.rename(1, 5, cube);
```

Listing 11: Set Material Number for Cuboid

Another point that needed to be taken care of was the commenting of redundant code to avoid compiler errors but which will not be mentioned in detail here and should be straightforward.

After all the fixes we get a fluid domain with an obstacle depicted in figure 13.

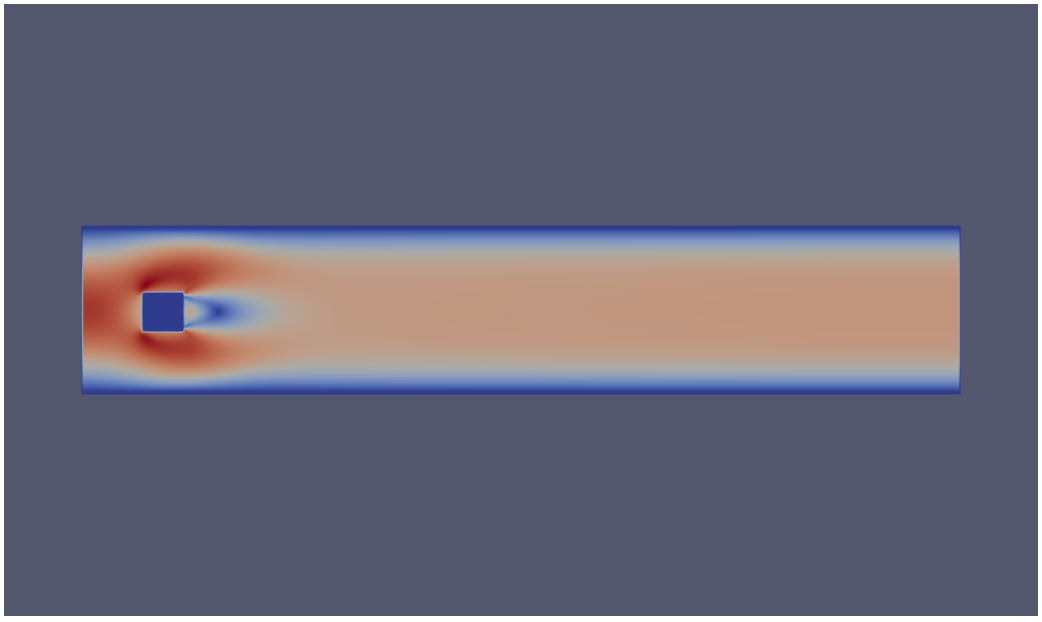

Figure 11: Exchanged obstacle - Cuboid

4.3 Bonus: Add second object to flow field

As a bonus task a second object was inserted into the fluid domain which required adding a new material number as well.

```
const T centerCylinderX1 = 0.2+.5;
const T centerCylinderY1 = 0.2+.5*L/2.;
const T radiusCylinder1 = 0.05;
```

Listing 12: Parameter Set for second object (Circle)

```
Vector<T,2> center1( centerCylinderX1,centerCylinderY1 );
IndicatorCircle2D<T> circle1( center1, radiusCylinder1 );
```

Listing 13: New Functor for Circle

```
superGeometry.rename(1, 5, circle1);
```

Listing 14: Set Material Number for Circle

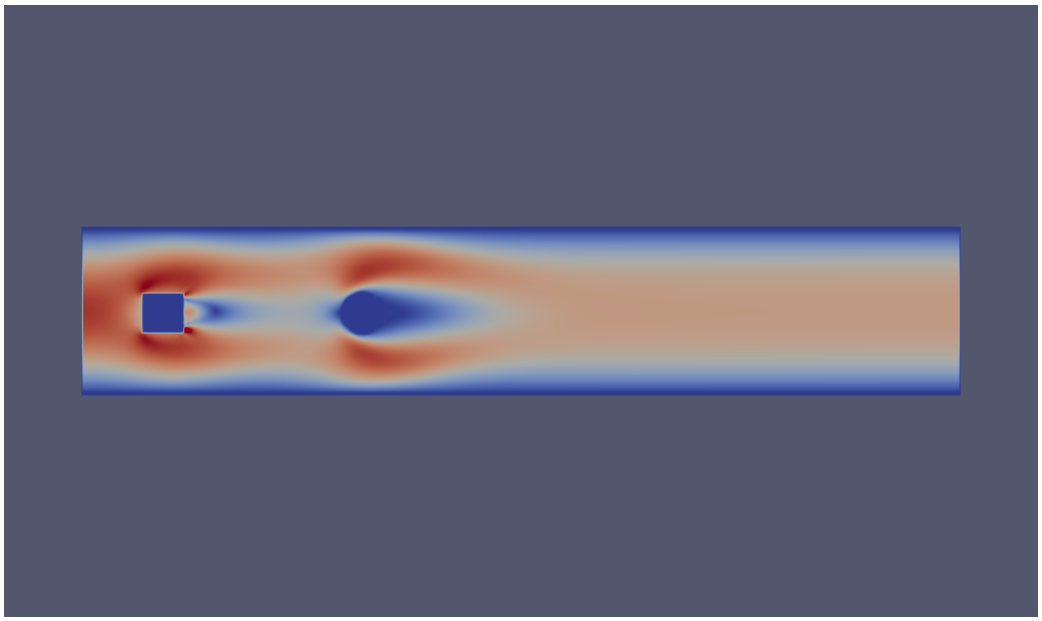

Figure 12: Two obstacle in the fluid domain

The problem with this setup was that no mass was conserved and after a certain point the post-processing results did not make much sense. The reason for that will be covered in the next exercise.

5 Exercise 3

The third exercise dealt again with the flow around the obstacles but now the important part was to create boundary conditions for the objects as no physical results could be obtained in exercise 2.

5.1 Definition of the Boundary Condition

In this exercise there is a **NoDynamics** condition for every material number **5** which means that there is interaction with the fluid but no mass conservation is given. Therefore a zero-velocity boundary is set inside the **circle**

```
sLattice.defineDynamics (superGeometry, 5, &instances::getNoDynamics<T,
    DESCRIPTOR>() );
```

Listing 15: Set NoDynamics

```
offBc.addZeroVelocityBoundary(superGeometry, 5, circle) ;
```

Listing 16: Zero-Velocity Boundary for circle

The next aim was to set these boundary conditions for the second obstacle in the fluid domain and to compare the results. The important thing to notice is that the object has to be **redefined inside this function**!

```
 ////////////////////////////////////////
 // Material=5 -->bouzidi
 ////////////////////////////////////////

 Vector<T,2> center1( centerCylinderX1,centerCylinderY1 );
 IndicatorCircle2D<T> circle1( center1, radiusCylinder1 );

sLattice.defineDynamics( superGeometry, 5,&instances::getNoDynamics<T,
    DESCRIPTOR>() );

 offBc.addZeroVelocityBoundary( superGeometry, 5, circle1 );

 ////////////////////////////////////////
 // Material=6 -->bouzidi
 ////////////////////////////////////////

 Vector<T,2> center( centerCylinderX,centerCylinderY );
 IndicatorCuboid2D<T> cube(xlength, ylength, center, theta);

sLattice.defineDynamics( superGeometry, 6,&instances::getNoDynamics<T,
    DESCRIPTOR>() );

 offBc.addZeroVelocityBoundary( superGeometry, 6, cube );
```

Listing 17: Modified C++ code section for boundary conditions

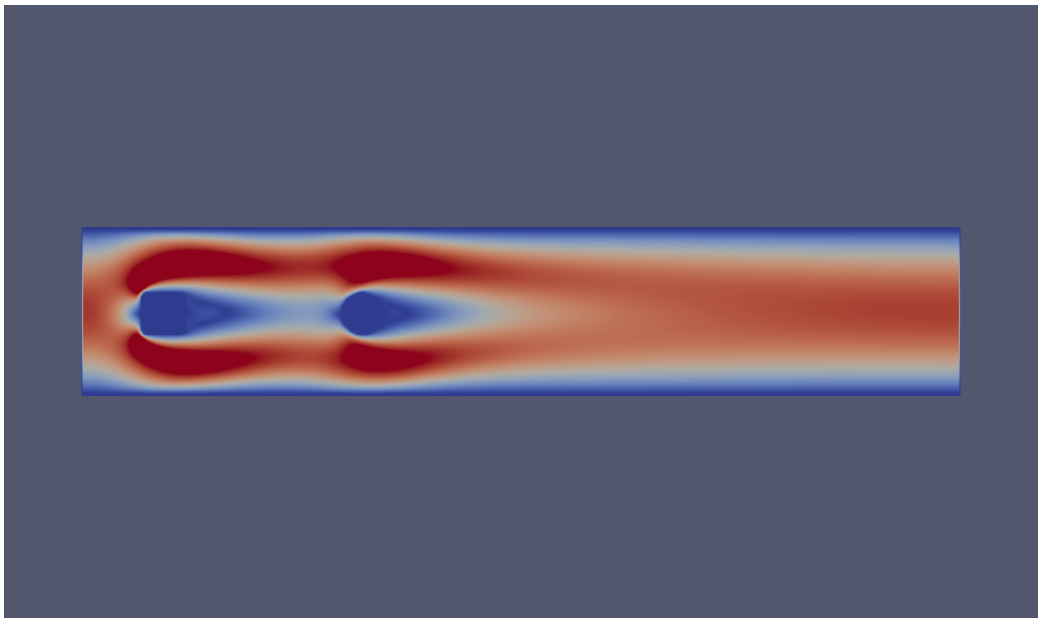

Figure 13: Fixed simulation

5.2 Smooth Start-Up

The following task included to delete the smooth start up and see how that would affect the flow/flow domain. To adapt the settings the **setBoundaryValues** section with **maxVelocity** has been changed. The last part of the code snippet given below, namely **frac[0]** had to be removed.

```
1 T maxVelocity = converter.getCharLatticeVelocity()*3./2.*frac[0]
```

Listing 18: Removal of smooth start up

Another run showed what effect the deleted part **frac[0]** had on the domain. As shown in figure 14 one can see that the *"unramped"* velocity acts as a shockwave inducing an enormous velocity gradient along the fluid domain.

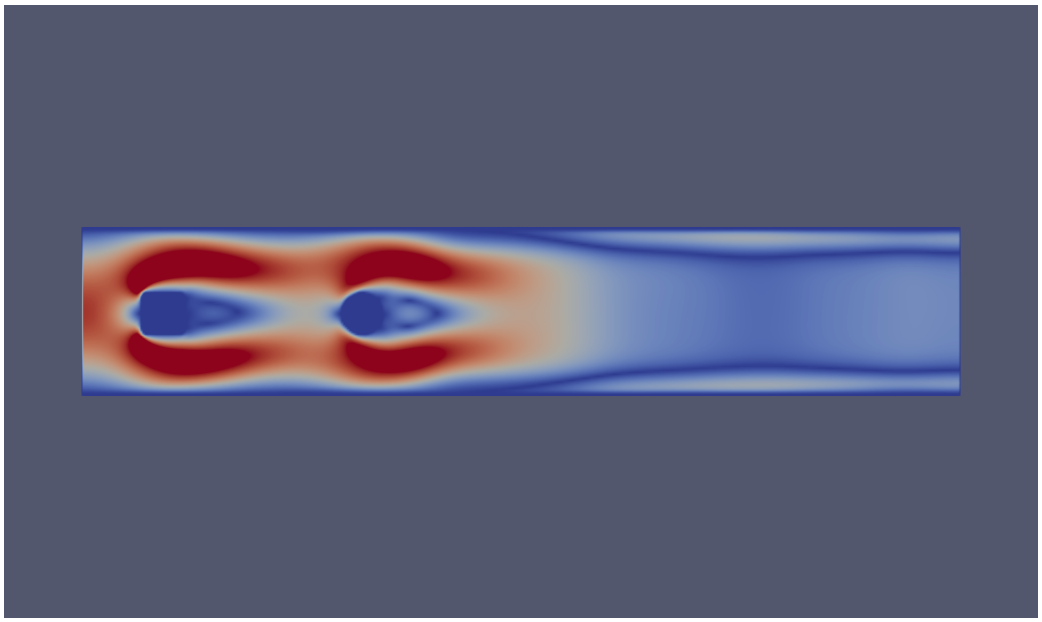

Figure 14: Shockwave propagation inside the fluid domain

It was clear that this gives very inaccurate results thus the smooth start up has been added again and changes have been made in the ramping. The percentage of the start up or more precisely the **slope of the ramp** was changed to 10% and 80% respectively and the post-processing results were compared. Please note that the post-processing results for the last time step for a start up of 80% and 40% look completely identical.

```
1 int iTmaxStart = converter.getLatticeTime( maxPhysT*0.4) ; // 0.8 & 0.1
```

Listing 19: Change of Ramping

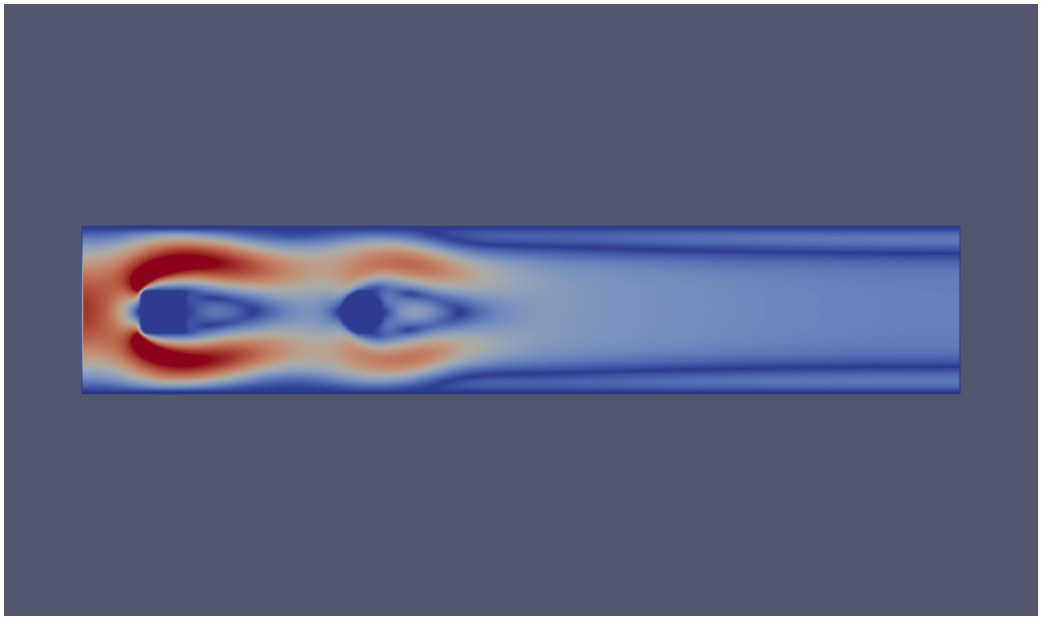

Figure 15: Startup 10% - Last Time Step

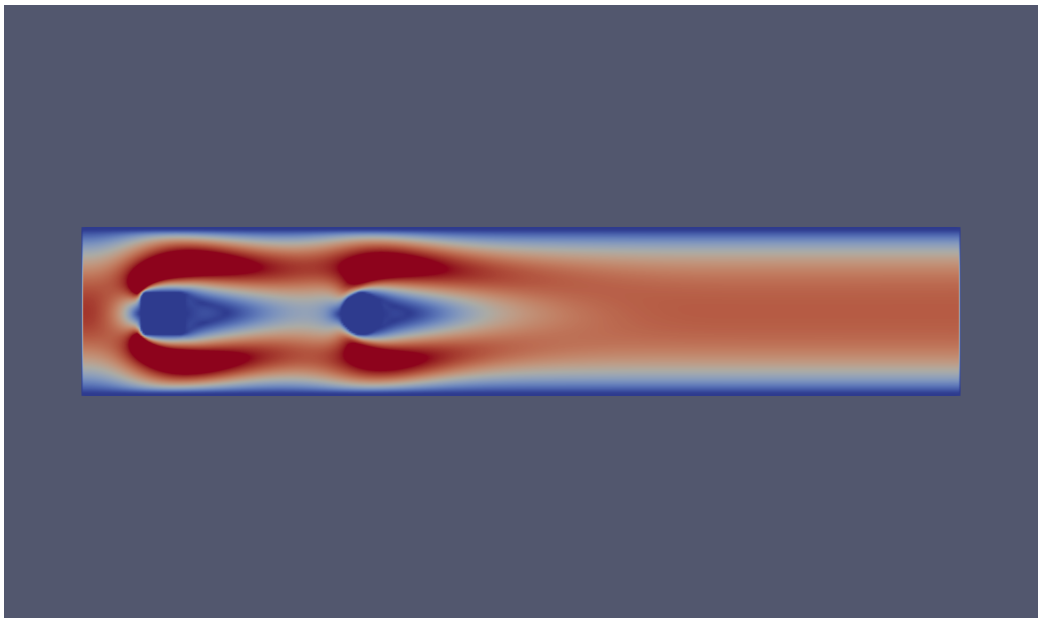

Figure 16: Startup 80% & 40% - Last Time Step

Comparing all three runs one can say that the *"most stable"* flow configuration could be achieved with a start up of 80% meaning that we have a very slow ramping of our velocity. In the other two runs with 10% and 40% fluctuations of the velocity field could be observed which were exaggerated in the 10% start up. One can conclude that the lower the percentage of the start up is the closer we move to the shockwave region we observed the exercise before.

5.3 Inlet - Pressure Boundary Condition

Henceforth the velocity boundary condition at the inlet has been exchanged with a pressure boundary condition.

```
1 sBoundaryCondition.addPressureBoundary( superGeometry, 3, omega );
```

Listing 20: Pressure Boundary Condition Definition

To define a ramping a functor in **setBoundaryValues** has been defined in order to keep the information of the density of the corresponging pressure in lattice units. Here the *smooth ramping* acts as a **perturbation** which is needed to **inject dynamics into the system**.

```
1 AnalyticalConst2D<T,T> rho(converter.getLatticeDensityFromPhysPressure(.1*
    frac[0]));
```

Listing 21: Definition of Pressure Functor with ramping

The last step before running the simulation was to define the pressure boundary at the inflow.

```
1 sLattice.defineRho(superGeometry, 3, rho) ;
```

Listing 22: Pressure Boundary definition at the inflow

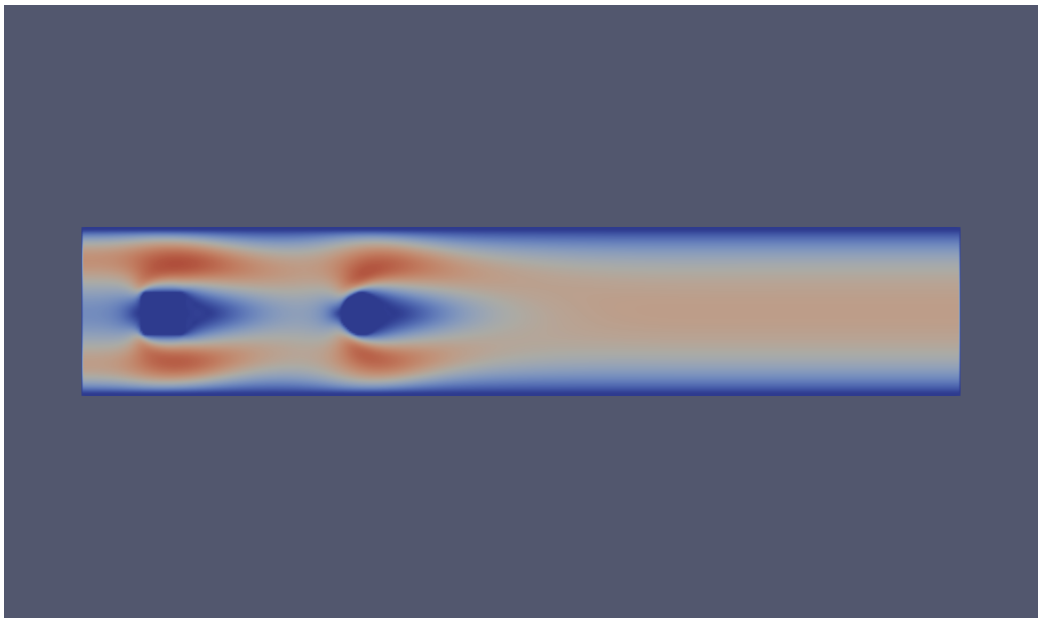

Figure 17: Pressure boundary - Ramping from 0 to 0.1 Pa - Last Time Step

5.4 Bonus: Increase of Inflow Velocity

In the bonus exercise and last section of this task the increase of the inflow velocity has been investigated. Changes in different options have been performed in order to see the effects on the velocity field.

The first step was to increase **maxVelocity** by **2**,**3** and **4** and compare the outcomes of the changes which can be found below.

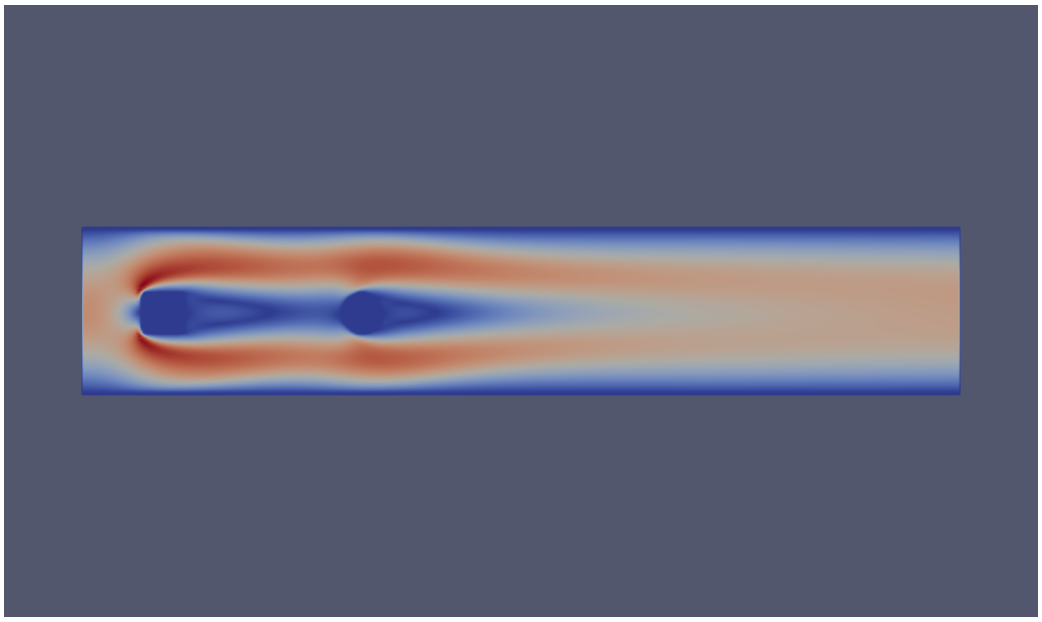

Figure 18: maxVelocity multiplied by a factor of 2

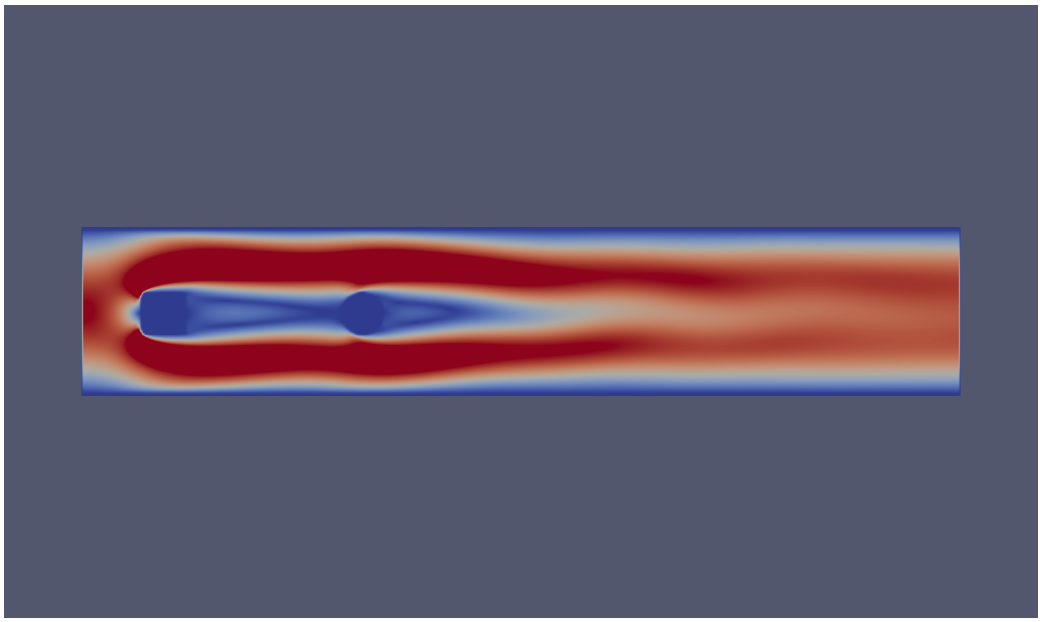

Figure 19: maxVelocity multiplied by a factor of 3

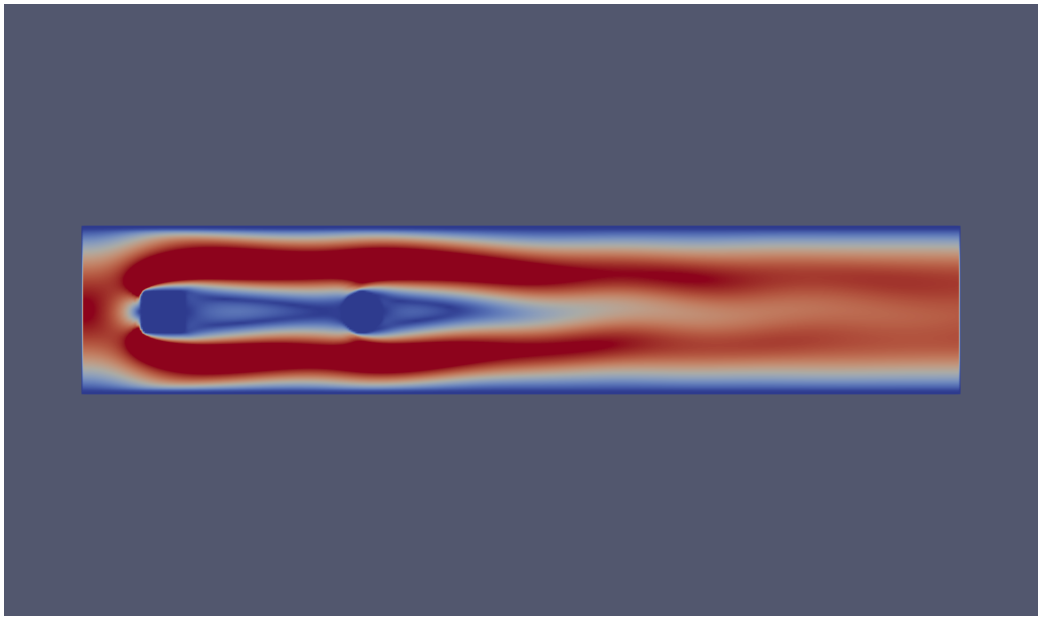

Figure 20: maxVelocity multiplied by a factor of 4

After these simulations where **maxVelocity** has been changed another approach was tested in the next step. The old value of **3./2.** was inserted into the expression for the velocity and changes have been made for the characteristical velocity **charU**_ in order to reach a Reynolds number of 40,60 and 80 respectively. The results of this approach are depicted below.

6 Exercise 4

The last exercise dealt with the validation of the **Lid Driven Davity** simulation. The question was how to get the velocity in x-direction along a line that was defined in Paraview in the first step and programatically inside the **openlb** code in the second step.

Using Paraview one uses the Filter **Data Analysis** → **Plot Over Line** in order to get the velocity values along the defined path. First the direction of the line had to be prescribed and the number of data points for the resolution was set to 100 in the properties.

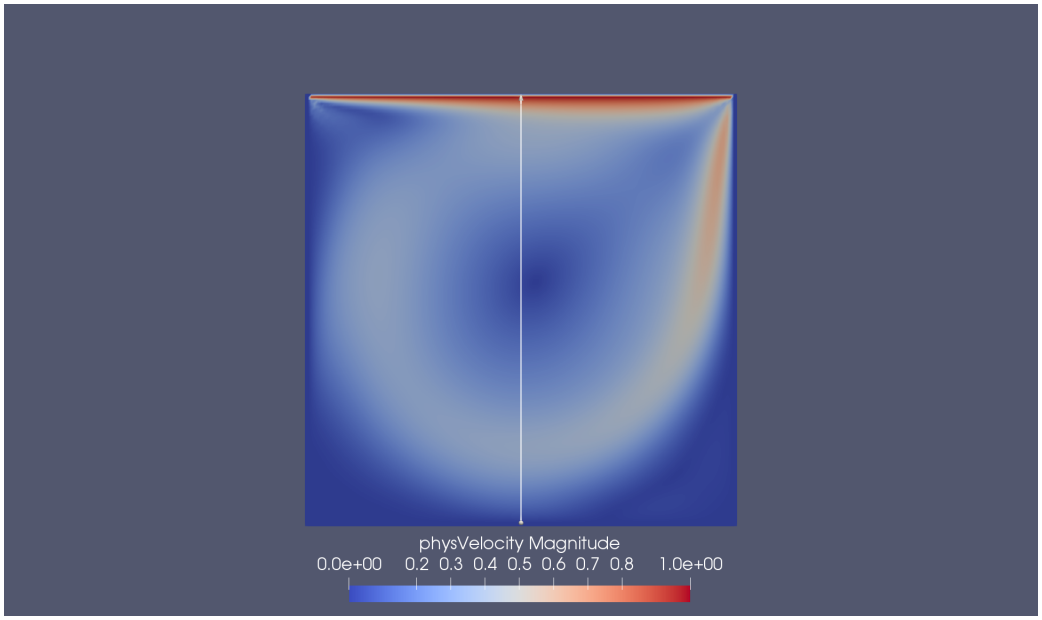

Figure 21: Plot Over Line Filter for Lid Driven Cavity

In order to only get the x-direction of the velocity the corresponding velocity has to be chosen inside the filters properties. The results were saved using **File** → **Save Data...**.

The other option was to change the code:

```
SuperLatticePhysPressure2D<T,DESCRIPTOR> pressure( sLattice, converter );
```

Listing 23: Access to velocity information on the lattice

```
AnalyticalFfromSuperF2D<T> interpolation( pressure, true, 1 );
```

Listing 24: Interpolation functor for velocityField

```
for ( int nY = 0; nY <= 100; ++nY ) {
      T postition[2] = {0.5, y_coord[nY]/100.0};
      T velocity[2] = {T(), T()};
```

Listing 25: 100 data points evenly distributed between 0 and 1 (height)

```
interpolation( velocity, postition );
```

Listing 26: Interpolate of the velocityField

In order to use the Gnuplot interface a static Gnuplot functor has to be defined.

```
1 static Gnuplot<T> gplot( "centerVelocityX" );
```

Listing 27: Gnuplot interface to create plots

To define the data that needs to be plotted the following line has to be written.

```
1 gplot.setData(dataXaxis, {dataYaxis1, dataYaxis2}, {"nameY1", "nameY2"}) ;
```

Listing 28: Define data to be plotted

Writing the plot in **png** or **PDF** format can be achieved by using

```
1 gplot.writePNG();
2 gplot.writePDF();
```

Listing 29: Output PNG or PDF

6.1 Comparing Numerical and Experimental Results

In order to compare the results compilation and re-running the simulation with the standard configuration of **Re = 1000** was necessary.

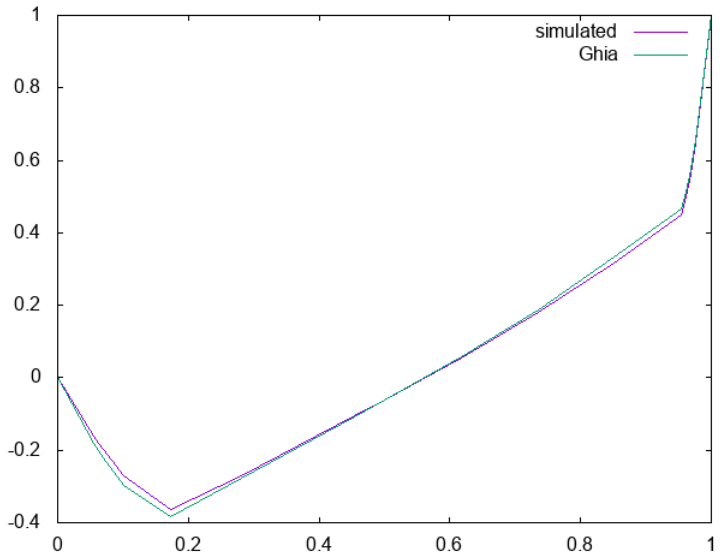

Figure 22: Re = 1000 - Comparison simulation and Data from Ghia

One can see that there is a very good agreement between both data curves. In the next task the Reynolds number was changed and the data were compared again.

6.2 Comparing Results with a change in Reynolds Number

As mentioned above the **cavity2d.cpp** was edited. One of the main tasks here was to familiarize with the following line

```
1 if ( iT == converter.getLatticeTime( maxPhysT ) || converged)
```

Listing 30: Functor to get information

This code snippet expresses an output for x-velocity along y-position at the last time step. Alternatively one can define another **or-condition** to see at the beginning of the simulation if everything is initialized properly.

```
1 if ( iT == converter.getLatticeTime( maxPhysT ) || converged || iT == 0)
```

Listing 31: Modified functor to get information

When changing the Reynolds number to **100** with the **xml** file one can see that the results are completely off.

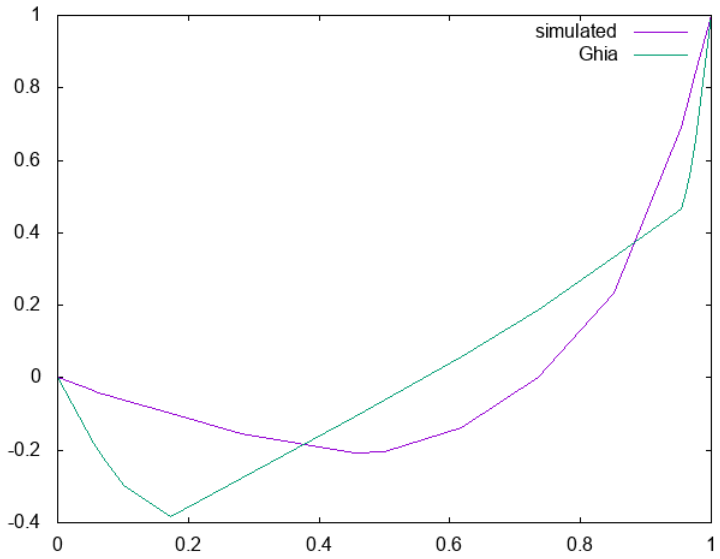

Figure 23: Re = 100 - Comparison simulation and Data from Ghia

Reason for that is that the wrong comparison is called inside the C++ code.

```
1 Vector<T,17> comparison = vel_ghia_RE1000;
```

Listing 32: Comparison call

To fix this a small changed was introduced.

```
1 Vector<T,17> comparison = vel_ghia_RE100;
```

Listing 33: Comparison call for Re = 100

The results now looked as expected and are in agreement with the Data from Ghia.

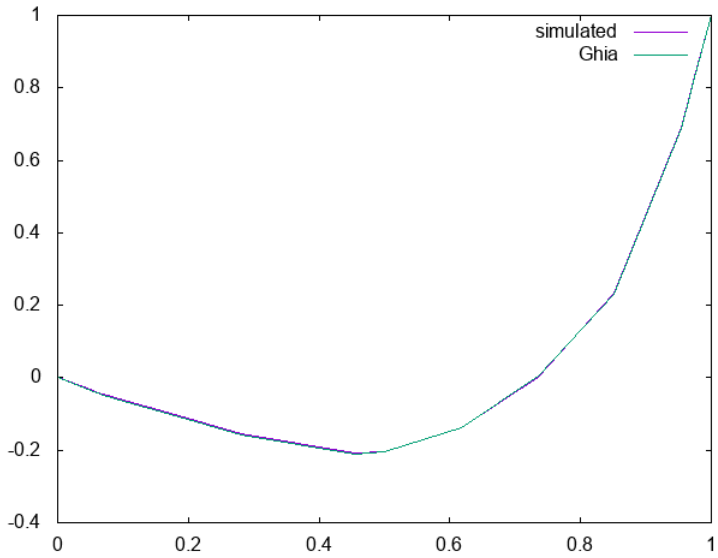

Figure 24: Re = 100 - Comparison of fixed simulation and Data from Ghia

6.3 Change of Functors

A very convenient and flexible way of getting the velocity and position data is to print them in the terminal. This can be achieved by adding the following lines into the C++ code.

```
clout << "position: " << postition[1]
          << "; velocity: " << velocity[0]
          << std::endl;
```

Listing 34: Terminal Console Output code snippet

```
[getResults] position: 0; velocity: 0.034063
[getResults] position: 0.01; velocity: 0.0342167
[getResults] position: 0.02; velocity: 0.0342029
[getResults] position: 0.03; velocity: 0.0341303
[getResults] position: 0.04; velocity: 0.0339682
[getResults] position: 0.05; velocity: 0.0336821
[getResults] position: 0.06; velocity: 0.0332526
[getResults] position: 0.07; velocity: 0.0326504
[getResults] position: 0.08; velocity: 0.0318091
[getResults] position: 0.09; velocity: 0.0307203
[getResults] position: 0.1; velocity: 0.0293616
[getResults] position: 0.11; velocity: 0.0276933
[getResults] position: 0.12; velocity: 0.025673
[getResults] position: 0.13; velocity: 0.0233348
[getResults] position: 0.14; velocity: 0.0206856
[getResults] position: 0.15; velocity: 0.0177108
[getResults] position: 0.16; velocity: 0.0144711
[getResults] position: 0.17; velocity: 0.0110245
[getResults] position: 0.18; velocity: 0.00741822
[getResults] position: 0.19; velocity: 0.00370146
[getResults] position: 0.2; velocity: -5.2324e-05
[getResults] position: 0.21; velocity: -0.003796
[getResults] position: 0.22; velocity: -0.00748
[getResults] position: 0.23; velocity: -0.0110676
[getResults] position: 0.24; velocity: -0.0145417
```

Figure 25: Terminal Output for the positions and corresponding velocity

The number of points has been changed in order to have the same resolution as in Paraview.

```
for ( int nY = 0; nY <= 17; ++nY ) {
      T postition[2] = {0.5, y_coord[nY]/128.0};
      T velocity[2] = {T(), T()};
```

Listing 35: 17 data points evenly distributed between 0 and 1 (height)

Another change has been made in the **position** vector in the **forloop** where the expression **y_coord[nY]** has been substituted with **T postition[2] = 0.5, nY/100.0;**. This was necessary to be free from the definition already existing in the code, namely

```
Vector <int,17> y_coord
```

which contained predefined points along the domain. With the change it was possible to incrementally go smaller steps along the defined line and get more data points, thus a less coarse curve to compare datas with.

References

[1] P. Nathen. "On the Stability and Accuracy of Lattice Boltzmann Schemes for the Simulation of Isotropic Turbulent Flows". In: *Communications in Computational Physics* (2018).

[2] T. Krüger et al. "The Lattice Boltzmann Method - Principles and Practice". In: *Springer Verlag* (2017).

[3] PD Dr. G. Thäter, Dr. M. J. Krause, and Fabian Klemens. *Introduction to Modelling Fluid Flow and Lattice Boltzman Methods.*

[4] U. Ghia, K. N. Ghia, and C. T. Shin. "High-Re solutions for incompressible flow using the Navier-Stokes equations and a multigrid method". In: *Journal of Computational Physics, 48:387-411* (1982).

[5] M. Schäfer and S. Turek. "Benchmark computations of laminar flow around a cylinder". In: *Vieweg+ Teubner Verlag* (1996).