LBM C++ Documentation

atru

March 25, 2019

Contents

List of Figures

List of Tables

1. Geometry

1.1 Introduction

The `Geometry` class provides functionality for creating model geometry or reading one from a file. Geometry is created as a $(L_x + 1) \times (L_y + 1)$ two-dimensional boolean array of 0s and 1s. 0s represent the solid nodes/grid points and 1s the fluid nodes. L_x is the domain length in horizontal direction and L_y is the domain length in the vertical direction. The number of grid points created is the lengths + 1 as the points themselves have no length.

The class is a template with L_x and L_y as its template parameters. The default constructor creates a $(L_x + 1) \times (L_y + 1)$ array as soon as a Geometry object is declared. At this point the user can manually add geometric elements to the object or can read the geometry from a file.

Currently available objects are rectangles, squares, ellipses, and circles as well as their regular and staggered arrays. The staggered arrays have a constraint that the even row is shorter then the odd one. Creating the opposite is possible through creation of two different regular arrays and will be shown in the Examples section. Anything can be imported from a file as long as the file is a $(L_x + 1) \times (L_y + 1)$ sized collection of 0 and non-zero values (the latter will be interpreted as boolean 1 i.e. `true`).

Since `Geometry` is a template all the implementation is directly included in the header file `Geometry.h`.

The class is a template and is instantiated as `Geometry<Nx+1, Ny+1> geometry_object_name`. Here `Nx` and `Ny` are the number of x and y points in the domain respectively (i.e. domain dimension in x or y - 1).

1.2 Commands

1.2.1 Walls

```
void add_walls(size_t dH=1, std::string where ="y")
```

Creates two side walls in either vertical (default, `"y"`) or horizontal direction (`"x"`). `dH` is the wall thickness expressed in solid nodes. Default is 1 solid node. Verifies if `dH` is larger than the domain size. The walls are creted starting with first and last index in the second direction and span the entire specified direction (Fig. 1.1).

1.2.2 Objects

`void add_rectangle(size_t` Δx`, size_t` Δy`, size_t` x_c`, size_t` y_c`)`

Creates a single rectangle or square with side lengths $\Delta x \times \Delta y$ and center coordinates (x_c, y_c) (Fig. 1.2).

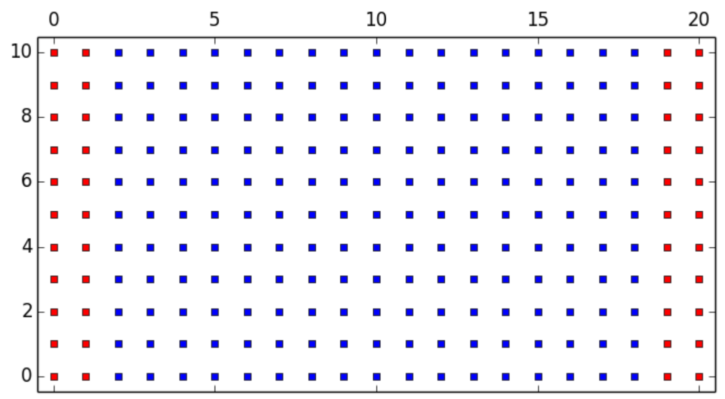

Figure 1.1: Geometry with walls (red) and a fluid domain (blue). Domain dimensions are 20x10 units which corresponds to 11×21 nodes. Walls are `dH=2` solid nodes deep and span the domain in y-direction (`where="y"`).

`void add_ellipse(size_t a, size_t b, size_t` x_c`, size_t` y_c`)`

Creates a single ellipse or circle with horizontal and vertical axis lenghts $a \times b$ and center coordinates (x_c, y_c) (Fig. 1.2).

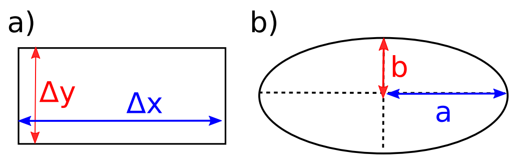

Figure 1.2: Length conventions used in defining a) Rectangles or squares and b) Ellispses or circles.

1.2.3 Arrays of objects

`void add_array(std::vector<size_t> obj_params, std::vector<size_t> arr_params, std::string object, size_t alpha=0)`

Creates an array of single type of objects `object`.

- `obj_params` - vector of object parameters; Following Figure 1.2 - Δx and Δy dimensions for a rectangle/square; $2a$ and $2b$ for ellipse/circle.
- `arr_params` - vector of array parameters - lower and upper x limits, lower and upper y limits, x and y spacing between object edges.
- `object` - object type; `"rectangle"`/`"square"` or `"ellipse"`/`"circle"`
- `alpha` - staggering angle in degrees; 0 (default) - no staggering; $(0-90)$ - custom staggering with even rows shorter than odd rows.

Array parameters for both not staggered and staggered cases are shown in Figure 1.3. Arrays are formed by computing the number of objects that fit within specified bounds according to following formulas for number of rows (number of objects in y direction):

$$N_{row} = \frac{y_f - y_0 - \Delta y - \delta y}{\Delta y + \delta y} + 2 \tag{1.1}$$

and columns (number of objects in x direction):

$$N_{col} = \frac{x_f - x_0 - \Delta x - \delta x}{\Delta x + \delta x + 2} \tag{1.2}$$

where x_0, x_f, y_0, y_f are lower and upper bounds on object centers in x and y respectively, $\Delta x, \Delta y$ are object lengths in x and y, and $\delta x, \delta y$ are distances between object sides in x and y. The same formulas are applied for staggered arrays except that the bounds for the staggered-even rows are recalculated based on the staggering angle. The initial center position x_{s0} of the even row is assumed to be between x_0 and x_1 of the odd row,

$$x_{s0} = x_0 + \delta x_s = x_0 + \frac{\delta x + \Delta x}{2} \tag{1.3}$$

The final position is similarly computed as $x_{sf} = x_f - \delta x_s$. The y bounds, y_{s0} and y_{sf} are calculated using the staggering angle α based on the law of sines,

$$\Delta x_\alpha = \frac{\sin{(\alpha)}}{\sin{(180 - 2\alpha)}} (\delta x + \Delta x) \tag{1.4}$$

The new vertical distance between object centers, δy_s is calculated using Pythagorean theorem,

$$\delta y_s = \sqrt{\Delta x_\alpha^2 - \delta x_s^2} \tag{1.5}$$

Corrected by subtracting Δy it is then used in Eq. 1.1 with previously specified bounds. The number of objects in a row, i.e. N_{col} in Eq. 1.2 is computed separately for odd and even rows using respective bounds.

In both staggered and not cases only the x_0, y_0 center positions are evaluated exactly, the x_f, y_f positions will vary. They represent an upper bound that cannot be crossed and the actual final object position will depend on number of objects that fit within the specified bounds. In staggered arrays the δy parameter is also not user-defined, as it is seen it is computed based on the staggering angle. The option is still left in the parameter list due to program design choices. The y_f value in the staggered arrays is determined based on the original y_f since it is larger than the corrected one.

Once the number of objects is computed, the objects are created as 0 entries in the domain by calling respective single object creation functions. The `object` parameter choice determines which types of objects are created. If the choice is `rectangle` or `square` the object will be either of the two which will depend on the user object parameter choice as they are both created using the same function, `add_rectangle`. The same holds for ellipses and circles.

The object adding functions check requested bounds with assertion statements and terminate the program with out of bounds input. The `add_array` function checks for proper `object` type which is also case sensitive, and throws an exception if an unknown type is requested.

The domain is represented as a `size_t` array which implies that any fractional input is rounded. In case of objects this will round (and display a warning) rectangles/squares with odd lengths.

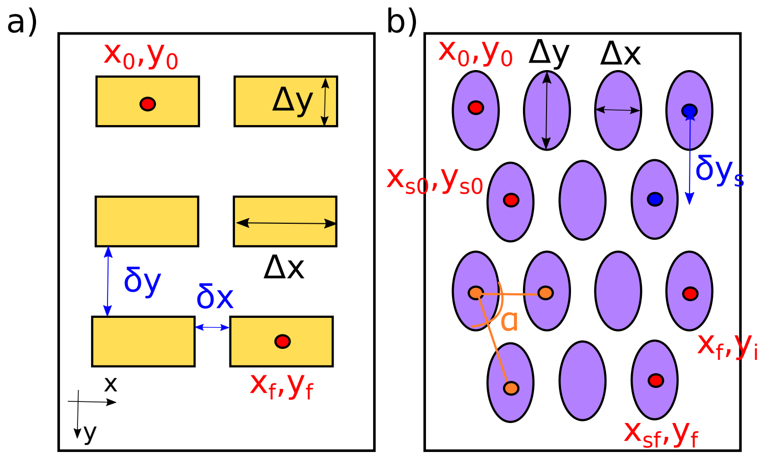

Figure 1.3: Length conventions used in defining a) regular arrays of rectangles or squares and b) staggered arrays of ellispses or circles.

Circular objects will be represented approximately, with results improving with array resolution. Staggered arrays will also be inexact - the actual staggering angle and corresponding vertical distance are calculated and shown to the user after the array is created. Again, increasing the number of grid points improves the quality of geometric features.

It is recommended that extensions - additions of new types of objects or arrays - follow the current logics of the implementation.

1.2.4 Reading from file and writing to file

`void read(std::string fin)`

Reads the geometry from file. The file should be a collection of 0s (solid) and non-zeros (fluid) values. If non-zeros are larger than 1 or not integers they will be converted to boolean 1. The file should tightly reflect the previously defined geometry dimensions - number of rows and columns should correspond to the declared Geometry object dimensions. There is no bounds checking at this step and using wrong number of entries will cause crashes or/and undefined behavior. Throws an exception if the file cannot be open.

`void read(std::string fin)`

Writes existing geometry to file. The resulting file is a collection of 0s (solid) and 1s (fluid) separated by spaces and is the size of the domain. There is a check if writing completed successfully.

1.3 Additional programs

- `geom_vis.py` - visualizes given geometry from file. Description provided in the program or by running it with `-h` option.

- `comp_geom.py` - simple program for computing parameters for given staggered array. Input are parameters in the program in the same way they are used in the geometry creation. Ouput are calculated staggered array paramters written to `stdout`. This program uses the same rounding logics as the main program thus the resulting parameters should match.

1.4 Examples

1.4.1 Single objects with walls

Single square

The following program creates a single 4×4 square with center coordinates $(10, 20)$. The domain is 20×50 units size and has a pair of walls in y-direction deep 1 solid node. The geometry is saved to file and visualized by running `./geom_vis.py -f "single_square.txt"`. The image of the geometry is shown in Figure 1.4.

```
void ex_single_square()
{
    Geometry<20,50> geom;
    geom.add_rectangle(4,4,10,20);
    geom.add_walls();
    geom.write("single_square.txt");
}
```

Single circle

The program creates a single circle with radius 10 and center coordinates $(20, 20)$. The domain is 40×50 units size and has a pair of walls in y-direction deep 1 solid node. The geometry is saved to file and visualized by running `./geom_vis.py -f "single_circle.txt"`. The image of the geometry is shown in Figure 1.4.

```
void ex_single_circle()
{
    Geometry<40,50> geom;
    geom.add_ellipse(10,10,20,20);
    geom.add_walls();
    geom.write("single_circle.txt");
}
```

Rectangular step

The program creates a single 20×10 rectangular step center coordinates $(20, 7)$ so that the bottom of the step starts right above the bottom wall. The domain is 60×30 units size and has a pair of walls in x-direction deep 2 solid nodes. The geometry is saved to file and visualized by running `./geom_vis.py -f "step.txt"`. The image of the geometry is shown in Figure 1.4.

```
void ex_step()
{
    Geometry<60,30> geom;
    geom.add_rectangle(20,10,20,7);
    geom.add_walls(2,"x");
    geom.write("step.txt");
}
```

1.4.2 Arrays of objects

Regular array of squares

The following program greates a regular (not staggered) array of squares (side length 4) in a 62×100 unit size domain without walls. The center coordinates of first object are $(x_0, y_0) = (4, 10)$ and the limit on coordinate center is $(x_f, y_f) = (58, 90)$. The distances $\delta x, \delta y$ are the same in both direction and measure 5 units.

```
void ex_reg_squares()
{
    Geometry<62,100> geom;
    std::vector<size_t> square = {4,4};
    std::vector<size_t> array = {4, 58, 10, 90, 5, 5};
    geom.add_array(square, array, "rectangle");
    geom.write("reg_array_sq.txt");
}
```

Note that the square array is invoked with an option "rectangle" as it will result in an array of squares due to the input parameters as specified in `square` vector. Option can also be `"square"` both leading to the same outcome. α has a default value of 0 and is thus omitted here. At the end the geometric domain is written to a file that can be visualized with `geom_vis.py`. The resulting domain is shown in Figure 1.5.

Staggered array of rectangles

This program creates two staggered arrays of rectangles differing by both object dimensions and array parameters. The vertical distance between the object sides is specified though not needed - it is recalculated as part of staggering implementation. The program also demonstrates interchangability of `"rectangle"` and `"square"` input options. The array is shown in Figure 1.5.

```
void ex_stag_rectancles()
{
    // Geometry object
    Geometry<56,100> geom;
    // First array
    std::vector<size_t> rect_1 = {2,4};
    std::vector<size_t> array_1 = {4, 58, 15, 50, 10, 5};
    geom.add_array(rect_1, array_1, "rectangle", 60);
    // Second array
    std::vector<size_t> rect_2 = {4,2};
    std::vector<size_t> array_2 = {8, 52, 60, 96, 3, 5};
    geom.add_array(rect_2, array_2, "square", 75);
    // Write to file
    geom.write("stag_array_rec.txt");
}
```

Regular array of ellipses

This program creates a regular array of ellipses with walls in x direction as shown in Figure 1.5.

```
void ex_reg_ellipses()
{
    Geometry<200,100> geom;
    std::vector<size_t> ellipse = {16,8};
    std::vector<size_t> array = {40, 180, 16, 91, 20, 10};
    geom.add_array(ellipse, array, "ellipse");
    geom.add_walls(1,"x");
    geom.write("reg_array_ell.txt");
}
```

Staggered array of circles

This program creates a staggered array of circles with walls in y direction as shown in Figure 1.5.

```
void ex_stag_circ()
{
    Geometry<248,200> geom;
    std::vector<size_t> circle = {16,16};
    std::vector<size_t> array = {16, 232, 40, 180, 20, 5};
    geom.add_array(circle, array, "circle", 60);
    geom.add_walls();
    geom.write("stag_array_circ.txt");
}
```

Reading from file

This program first reads the geometry created by 1.4.2 and then writes it to a new file. The outputs should match and can be compared using `diff` - result should be no output.

```
void ex_read_write()
{
    Geometry<56,100> geom;
    geom.read("stag_array_rec.txt");
    geom.write("geom_stag_test.txt");
}
```

1.5 Tests

All tests performed with `Geometry` class are available in the `tests/geometry_tests/` directory. The file `geom_test_core.cpp` contains non-template implementation and the header `geom_test_core.h` all implementation that uses template parameters. Test functions are called from `geom_test_main.cpp`. The files also include example code shown in Section 1.4 and the directory contains the python scripts for computing parameters and visualization.

It is recommended that new geometry is added and tested in the same way or even as part of this particular package. To compile, run `g++ -std=c++11 -o geom_test geom_test_main.cpp geom_test_core.cpp`. To run, simply enter `./geom_test`.

1.6 To Do

- Object filling and trimming for ellipses and circles.
- Explanation of wrappers in the testing suite.
- Improve the documentation
- Add qualifiers, especially for strings

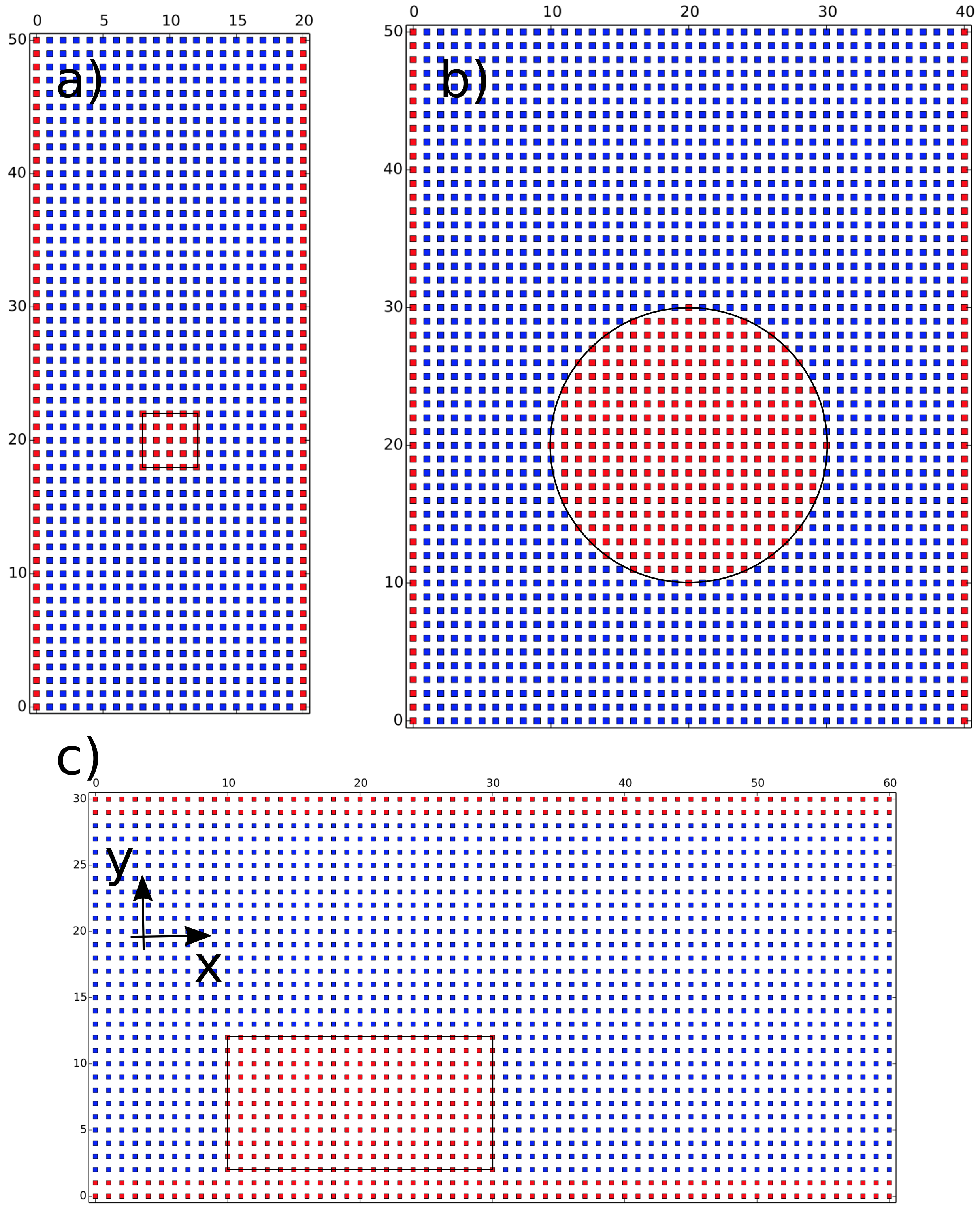

Figure 1.4: Geometric domains for single object examples. The objects are additionally highlighted with a contour approximately representing their ideal extents; a) Single square, b) Single circle, c) Rectangular step

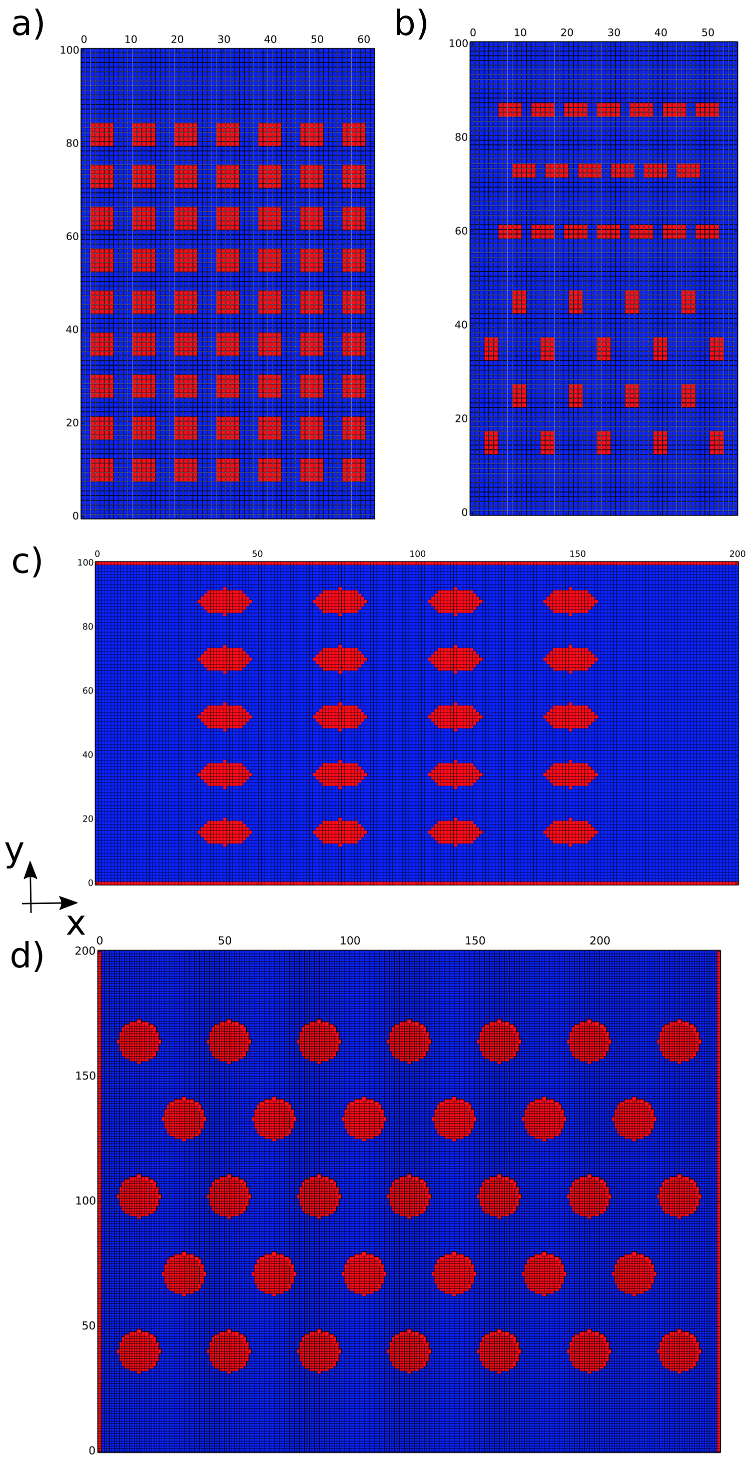

Figure 1.5: Geometric domains for arrays of objects examples: a) Regular array of squares, b) Staggered array of rectangles, c) Regular array of ellipses, d) Staggered array of circles

2. Fluid

2.1 Introduction

The `Fluid` class provides interface for setting up and manipulating quantities directly related to fluids. The class is a general class that stores fluid properties and density distribution. The class also stores macroscopic properties - density and x and y velocity components. The macroscopic properties are initialized and computed when needed or requested by the user but storage for them is allocated upon creation of a `Fluid` object.

Other simulation components and forces attributed to presence of different phases of the same fluid and/or other fluids will be introduced through separate interfaces with proper examples of usage.

The class is a template and is instantiated as
`Fluid<Nx+1, Ny+1, Nd> fluid_object_name(`*`parameters`*`)`. Here `Nx` and `Ny` are the number of x and y points in the domain respectively (i.e. domain dimension in x or y - 1).

2.2 Commands

2.2.1 Constructors

There are currently three constructors - one default and two custom ones. Default implements a set of fluid parameters typical of Shan and Chen LBM simulations. The parameters set by the user are the relaxation time, τ and the lattice speed, c_s [**?**, **?**]. User can also provide a name for the fluid. The second custom constructor uses default simulation parameters and a custom fluid name.

```
Fluid()
```

Default constructor - initilizes fluid's name to `fluid`, lattice speed to $\frac{1}{3}$, and relaxation time to 1.0.

```
Fluid(const std::string &name)
```

initializes fluid's name, while other properties are initialized using the default constructor.

```
Fluid(const std::string &name, double cs, double tau)
```

initializes fluid's name, lattice speed `cs`, and relaxation time `tau` to user-defined values.

2.2.2 Initialization of density and density distribution

```
void simple_ini(const Geometry<Lx,Ly>& geom, double val)
```

Initializes density to `val` on the fluid nodes and to 0 on the solid nodes. The position of solid nodes is determined from the geometric object `geom`. The initialization takes place *without* computing the density - it initializes directly the density distribution function.

2.2.3 Macroscopic property computation and retrieval

`grid_2D get_rho()`, `grid_2D get_ux()`, `grid_2D void get_uy()`

Computes from the distribution function and returns the density and x and y velocity components. `grid_2D` is a `typedef` for `std::array<std::array<double, Lx+1>, Ly+1>` where `Lx` and `Ly` are lengths of the domain. The `typedef` is currently private and local to the class `Fluid`.

2.2.4 Other properties retrieval

`grid_full get_f()`

Get the distribution function. `grid_full` is a `typedef` for `std::array<std::array<std::array<double, Nd>, Lx+1>, Ly+1>`.

2.2.5 Output

`void print_properties() const`

Prints fluid properties - name, speed of sound, relaxation time, viscosity, inverse relaxation time, and domain dimensions.

`void write_rho(const std::string fname)`, `void write_ux(const std::string fname)`, `void write_uy(const std::string fname)`

Computes the density and velocity components x and y, respectively, and writes them to a file `fname`. Checks if writing succeeded and throws an error if that was not the case.

`void write_f(const std::string fname) const`

Writes the density distribution function to a set of files with common name `fname`. Each file is `fname_i` where *i* is the number of lattice direction i.e. each file represents the density distribution within one lattice direction, resulting in `Nd` files.

3. LBM

3.1 Introduction

The class `LBM` provides interface for Shan and Chen LBM simulation. The class is a template and is instantiated as
`LBM<Nx+1, Ny+1, Nd> lbm_object_name(`*`Fluid<Nx+1, Ny+1, Nd> fluid`*`)`. Here `Nx` and `Ny` are the number of x and y points in the domain respectively (i.e. domain dimension in x or y - 1). `fluid` is a `Fluid` class object as described in Fluid.

3.2 Commands

3.2.1 Constructors

The default constructor for this class is explicitly deleted. There is only one constructor - that takes a `Fluid` object and there will be another one, for taking two `Fluid` objects for two-phase flow simulations.

```
LBM(Fluid<Nx+1, Ny+1, Nd> fluid)
```

Creates a reference to object `fluid` and initializes equilibrium distribution function to 0.0. Equilibrium distribution function is a private member of this class as it is not used anywhere outside of it. The current inefficiency of this code is that it also creates a second fluid object (with `new`) and a reference to it. This will eventually be refactored. Idea was to have the same interface for both single and multiphase flow.

3.2.2 LBM Simulation

```
void f_equilibrium()
```

Computes the equilibrium distribution function for a single fluid. In the future it will have two phase/two components capabilities.

```
void collide(Fluid<Lx,Ly,Nd> &fluid)
```

Performs the collision operation on the `fluid` object's density distribution function.

```
void add_volume_force(const Geometry<Lx,Ly>& geom,
   Fluid<Lx,Ly,Nd> &fluid, const std::array<double, Nd> &force) const
```

Adds a volume force to the fluid density distribution function. `force` is an stl array that specifies force directions with respect to lattice directions. Lattice numbering used in this program is shown in Figure 3.1. `geom` is the Geometry object and is needed for distinguishing between solid and fluid nodes.

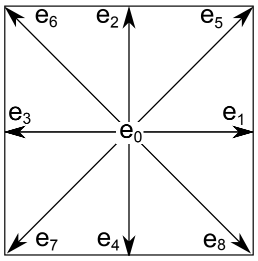

Figure 3.1: Lattice structure i.e. numbering convention used in this program.

`void stream(const Geometry<Lx,Ly>& geom, Fluid<Lx,Ly,Nd> &fluid) const`

Performs a stream operation on the density distribution function stored in `fluid` object. `geom` object is used to distinguish solid from fluid nodes. Implements periodic and bounce back boundary conditions following *cite*.

3.2.3 Getters

`grid_full get_feq()`

Retrieve the equilibrium distribution function. `grid_full` is a `typedef` for `std::array<std::array<std::array<d` `Nd>, Lx+1>, Ly+1>`.

4. Single phase flow benchmarks

4.1 Introduction

This chapter provides examples for common and benchmark single component and single phase flow scenarios. These cases where also used for testing and verifications and can be found in the testing directory.

4.2 Fluid in periodic domain

This example models a fluid in a fully periodic domain with no solid objects. The density of the fluid is supposed to be constant throughout the simulation, i.e. the final density as written in the files after the simulation loop should equal, on every node, to initial density `rho_0`. As there are no external forces acting on the fluid and hence no flow, both velocity components should be equal to 0.0 on every node.

The program creates and initializes the objects and runs the simulation for `max_steps` time. After the simulation loop it computes and saves the macroscopic variables to files. The loop represents a simple, basic LBM simulation in the order followed by this program. The `disp_every` parameter determines how often the program outputs progress status.

```
#include <iostream>
#include <array>
#include "../../geometry.h"
#include "../../fluid.h"
#include "../../lbm.h"

using std::cout;
using std::endl;
using std::string;

int main()
{
  // Additional parameters
  int max_steps = 1000;
  int disp_every = 200;
  double rho_0 = 1.0;

  // Output file names
  string file_rho("./empty/rho.txt"),
       file_ux("./empty/ux.txt"), file_uy("./empty/uy.txt");

  // Simulation objects
  Geometry<5,10> geom;
  Fluid<5,10,9> fluid("water");
  LBM<5,10,9> lbm(fluid);
  // Simulation
  fluid.simple_ini(geom, rho_0);
  int step = 1;
  while (step <= max_steps){
    lbm.f_equilibrium();
    lbm.collide(fluid);
    lbm.stream(geom, fluid);
    if (!(step%disp_every))
      cout << "---- Iteration " << step << std::endl;
    step++;
  }

  // Save final data
  fluid.write_rho(file_rho);
  fluid.write_ux(file_ux);
  fluid.write_uy(file_uy);
}
```

4.3 Laminar flow through a channel

Fluid is confined in a 2D channel with wall boundaries on the sides and periodic boundaries in the flow direction. Fluid is subject to uniform volume force causing it to move in the positive y-direction. Within a number of steps a stable, equilibrium velocity profile develops. The x-dependence of this profile should match the analytical solution closely, especially on fine grids. The flow (i.e. y) direction can be very small since boundaries are periodic.

```
#include <iostream>
#include <ctime>
#include <array>
#include "../../geometry.h"
#include "../../fluid.h"
#include "../../lbm.h"
#include "benchmarks.h"

using std::cout;
using std::endl;
using std::string;

int main()
{
  // Parameters
  constexpr size_t Nx = 50, Ny = 10, Nd = 9;
  int max_steps = 10000;
  int disp_every = 500;
  int save_every = 500;
  double rho_0 = 1.0;

  // Simulation time measurment
  std::time_t sim_start, sim_end;

  // Output file names
  string file_rho("./laminar/rho.txt"),
      file_ux("./laminar/ux.txt"), file_uy("./laminar/uy.txt");
  string file_rho_temp("./laminar/rho"),
       file_ux_temp("./laminar/ux"), file_uy_temp("./laminar/uy");

  // Simulation objects
  Geometry<Nx,Ny> geom;
  Fluid<Nx,Ny,Nd> fluid("water");
  LBM<Nx,Ny,Nd> lbm(fluid);

  // Force - flow in positive y direction
  // (notice the sign change due to dPdL)
  const double dPdL = 1e-5;
  std::array<double,9> force = {0, 0, -1, 0, 1, -1, -1, 1, 1};
  std::for_each(force.begin(), force.end(), [&dPdL](double &f){ f*=(-dPdL*(1.0/6.0))
      ; });

  // Geometry setup and initialization
  geom.add_walls();
  fluid.simple_ini(geom, rho_0);
  geom.write("./laminar/geom_walls.txt");
  fluid.write_f("./laminar/f_walls");

  // Simulation
  int step = 1;
  sim_start = std::time(nullptr);
  while (step <= max_steps){
    lbm.f_equilibrium();
    lbm.collide(fluid);
    lbm.add_volume_force(geom, fluid, force);
    lbm.stream(geom, fluid);
    if (!(step%disp_every))
      cout << "---- Iteration " << step << std::endl;
    if (!(step%save_every))
      fluid.save_state(file_rho_temp, file_ux_temp, file_uy_temp, step);
    step++;
  }
  sim_end = std::time(nullptr);
  std::cout << "Simulation time: " << std::difftime(sim_end, sim_start) << std::endl
      ;

  // Save final data
  fluid.write_rho(file_rho);
  fluid.write_ux(file_ux);
  fluid.write_uy(file_uy);
}
```

The force is introduced using the `std::for_each` on previously defined force array of directions. The sign of the force changes in the process due to negative value of `dPdL`. The sign of `dPdL` originates from the fact that "pressure drop" i.e. difference between inlet and outlet pressure will indeed be negative for flow to occur. In addition to components already described in 4.2, macroscopic variables in this example are saved at regular intervals during the simulation. Simulation time is also measured and displayed. Note that saving and displaying during the simulation, slows the simulation down - a more efficient example will be shown in the next section.

Figure 4.1 shows the development of y velocity profile in x direction while Figure 4.3 compares the developed profile with analytical solution. Figure **??** shows the velocity magnitude surface plot and streamlines.

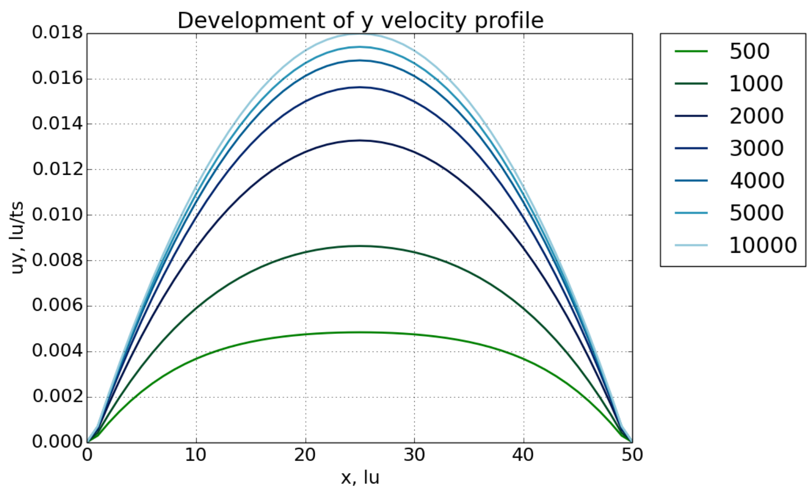

Figure 4.1: y velocity profile in the x direction at various steps of the simulation.

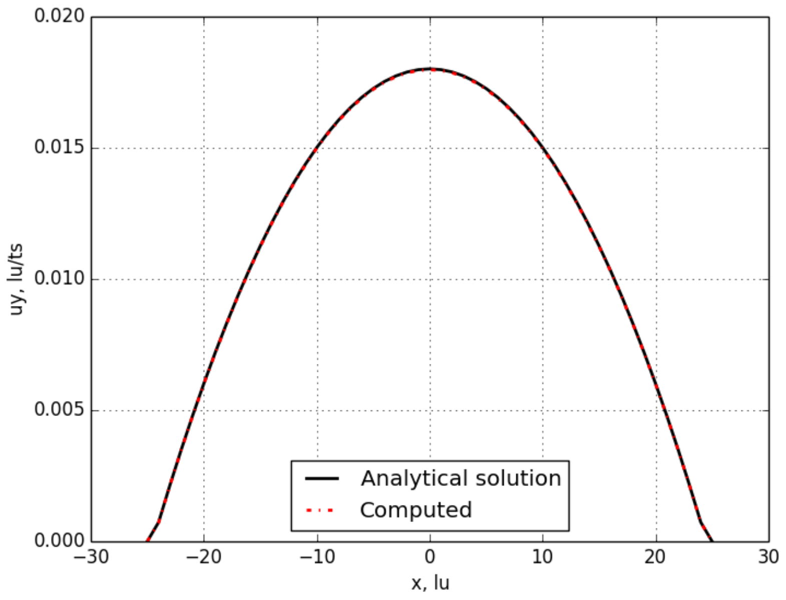

Figure 4.2: Comparison between simulation and analytical solution.

4.4 Flow through an array of squares

In this program (shown in 4.1) fluid flows through an array of objects - squares arranged in a non-staggered (rectangular) way. There is no direct solution for this kind of flow, the solution was compared to one obtained with previously verified code in MATLAB.

The only real difference between this code and one in Section 4.3 is that the geometry is more complex. Again, the flow is in the positive y direction, there are walls on the sides and the domain

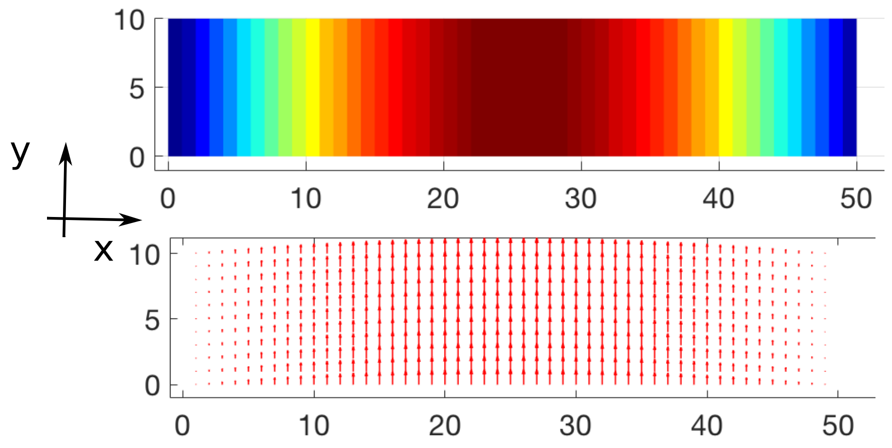

Figure 4.3: Surface plot of velocity magnitude and velocities streamlines.

is periodic in y. The important thing is to make sure the flow develops fully which is confirmed by tracking the velocity magnitude or/and the Reynolds number. Figure **??** shows the mean and maximum velocity magnitudes with simulation step. Values were computed across the whole domain, excluding 0.0s. Figure **??** shows the Reynolds number computed at the center of the domain. Both figures indicate that the equilibrated flow is reached at around step 400.

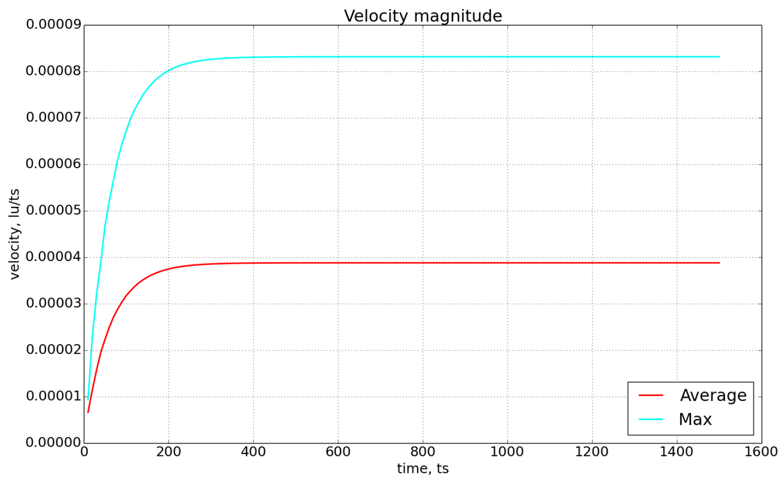

Figure 4.4: Development of mean and maximum velocity magnitude (excludes zero values).

Velocity magnitude across the domain is shown in Figure 4.6 and streamlines are shown in Fig. 4.7 and 4.8.

The efficient version of the code is shown in 4.2. For longer programs or way to determined equilibrium some saving would be advised, but here the data is saved only at the end. The geometry and equilibrium distribution function is also saved at the beginning for comparison with MATLAB.

This program is compiled with `-O3` compiler optimizations and compared to MATLAB code with the same components in the main loop (which is the only part timed). The main loop in program

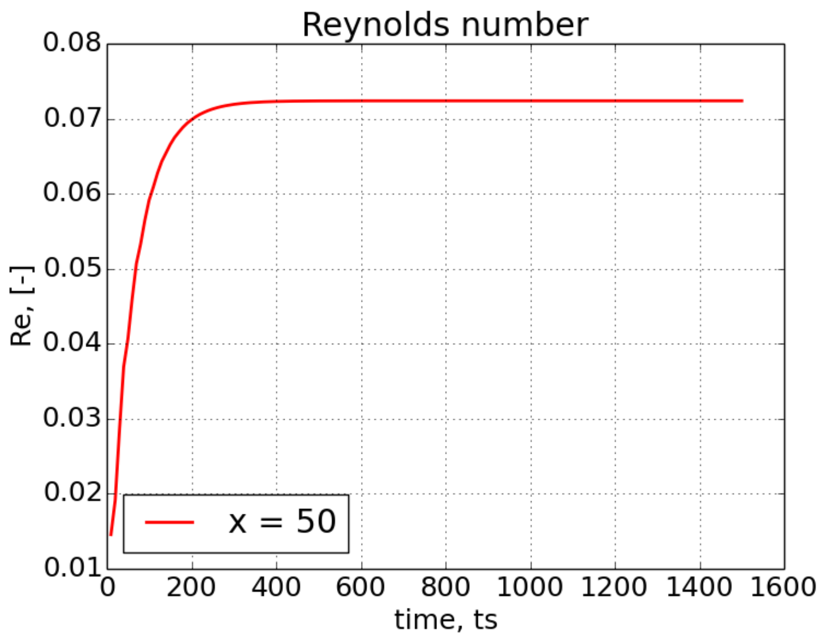

Figure 4.5: Development of Reynolds number at x = 50.

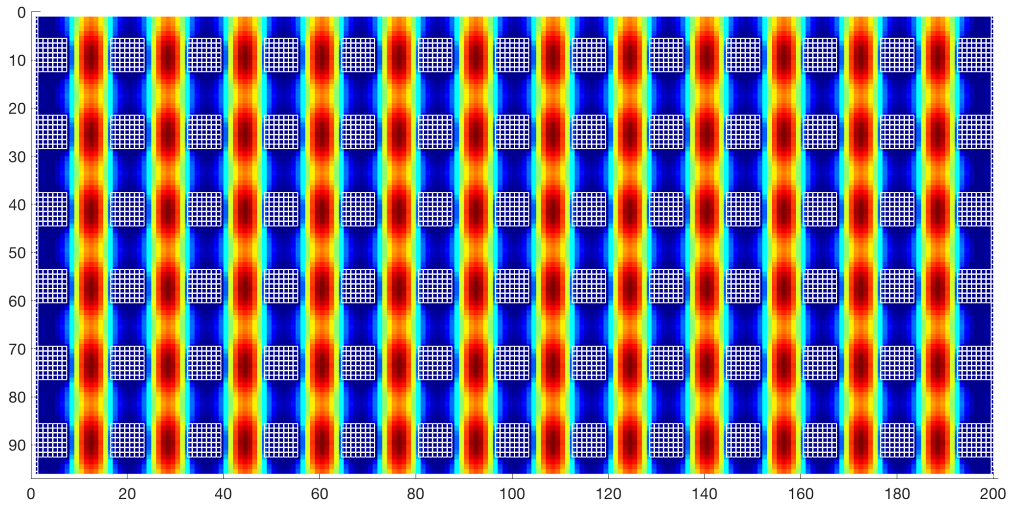

Figure 4.6: Surface plot of velocity magnitude. Indexes are shifted by 1 w.r.t the simulation. Horizontal direction is x and vertical is y.

above takes 2.82s on average while MATLAB loop takes 12.7s. There were 11 measurements each, MATLAB version used was 2018b. The first MATLAB result was slightly higher and discarded (this tends to happen with MATLAB - first run is slower). Nothing was run while the programs

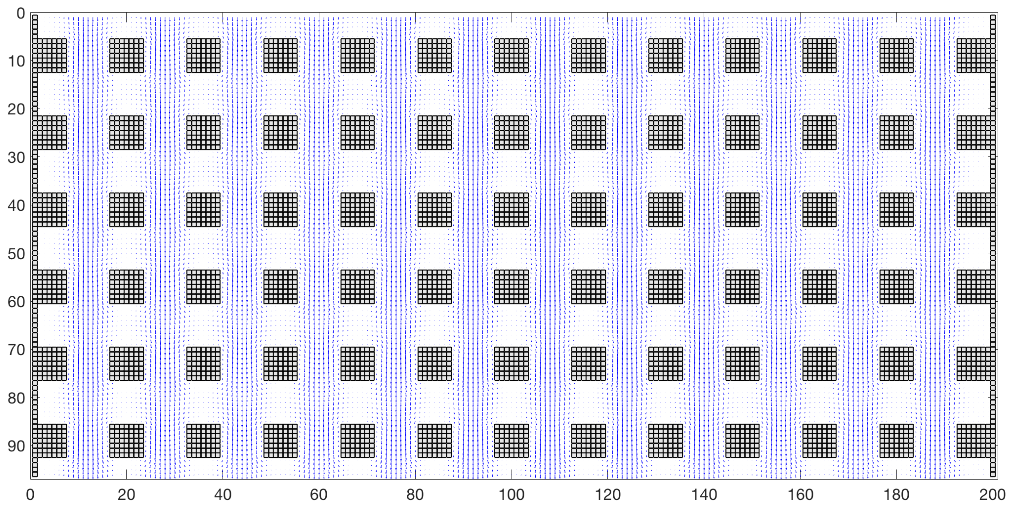

Figure 4.7: Velocity streamlines. Indexes are shifted by 1 w.r.t the simulation. Horizontal direction is x and vertical is y.

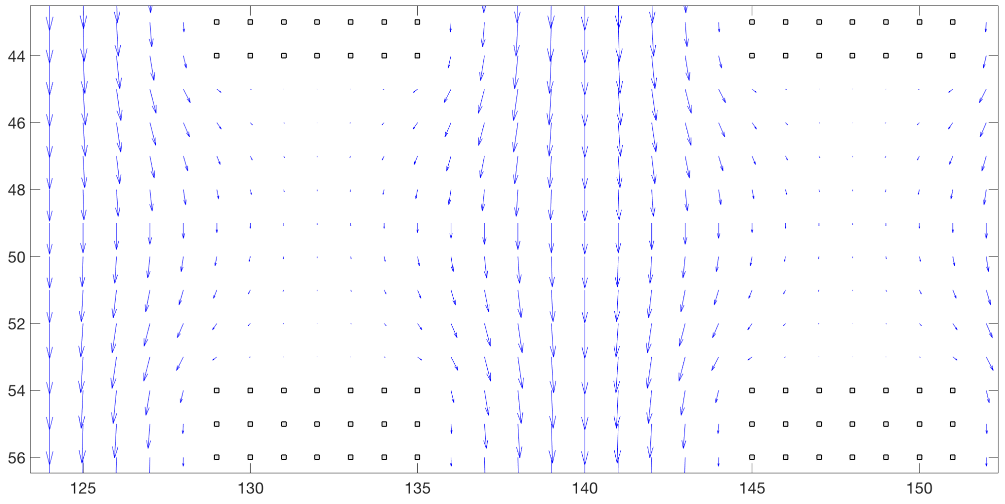

Figure 4.8: Zoomed velocity streamlines. Indexes are shifted by 1 w.r.t the simulation. Horizontal direction is x and vertical is y.

were timed (except for a number of background processes - but that still makes the two results comparable). C++ code is therefore 4.5 times faster than MATLAB equivalent. This is particularly interesting given that previously developed parallel two phase flow LBM in MATLAB was about 7-8 times faster on 12 cores than on one, which is not that far from the C++/MATLAB comparison here.

```cpp
#include <iostream>
#include <ctime>
#include <array>
#include "../../geometry.h"
#include "../../fluid.h"
#include "../../lbm.h"

using std::cout;
using std::endl;
using std::string;

int main()
{
  // Parameters
  constexpr size_t Nx = 199, Ny = 95, Nd = 9;
  int max_steps = 1500;
  int disp_every = 100;
  int save_every = 10;
  double rho_0 = 1.0;

  // Simulation time measurment
  std::time_t sim_start, sim_end;

  // Output file names
  string file_rho("./arrays/rectangular/rho.txt"),
       file_ux("./arrays/rectangular/ux.txt"), file_uy("./arrays/rectangular/uy.txt");
  string file_rho_temp("./arrays/rectangular/rho"),
       file_ux_temp("./arrays/rectangular/ux"), file_uy_temp("./arrays/rectangular/uy");

  // Simulation objects
  Geometry<Nx,Ny> geom;
  Fluid<Nx,Ny,Nd> fluid("water");
  LBM<Nx,Ny,Nd> lbm(fluid);

  // Force - flow in positive y direction
  const double dPdL = 1e-6;
  std::array<double,9> force = {0, 0, -1, 0, 1, -1, -1, 1, 1};
  std::for_each(force.begin(), force.end(), [&dPdL](double &f){ f*=(-dPdL*(1.0/6.0)); });

  // Geometry setup and initialization
  geom.add_walls();
  std::vector<size_t> square = {6,6};
  std::vector<size_t> array = {3, Nx-2, 8, Ny-5, 10, 10};
  geom.add_array(square, array, "square");
  geom.write("./arrays/rectangular/geom_squares.txt");
  fluid.simple_ini(geom, rho_0);

  // Simulation
  int step = 1;
  sim_start = std::time(nullptr);
  while (step <= max_steps){
    lbm.f_equilibrium();
    lbm.collide(fluid);
    lbm.add_volume_force(geom, fluid, force);
    lbm.stream(geom, fluid);
    if (!(step%disp_every))
      cout << "---- Iteration " << step << std::endl;
    if (!(step%save_every))
      fluid.save_state(file_rho_temp, file_ux_temp, file_uy_temp, step);
    step++;
  }
  sim_end = std::time(nullptr);
  std::cout << "Simulation time: " << std::difftime(sim_end, sim_start) << std::endl;

  // Save final data
  fluid.write_rho(file_rho);
  fluid.write_ux(file_ux);
  fluid.write_uy(file_uy);
}
```

Listing 4.1: Code for fluid flow through an array of squares.

```
#include <iostream>
#include <ctime>
#include <array>
#include "../../geometry.h"
#include "../../fluid.h"
#include "../../lbm.h"

using std::cout;
using std::endl;
using std::string;

int main()
{
  // Parameters
  constexpr size_t Nx = 199, Ny = 95, Nd = 9;
  int max_steps = 1500;
  double rho_0 = 1.0;

  // Simulation time measurment
  std::time_t sim_start, sim_end;

  // Output file names
  string file_rho("./arrays/rectangular/rho.txt"),
       file_ux("./arrays/rectangular/ux.txt"), file_uy("./arrays/rectangular/uy.txt");

  // Simulation objects
  Geometry<Nx,Ny> geom;
  Fluid<Nx,Ny,Nd> fluid("water");
  LBM<Nx,Ny,Nd> lbm(fluid);

  // Force - flow in positive y direction
  const double dPdL = 1e-6;
  std::array<double,9> force = {0, 0, -1, 0, 1, -1, -1, 1, 1};
  std::for_each(force.begin(), force.end(), [&dPdL](double &f){ f*=(-dPdL*(1.0/6.0)); });

  // Geometry setup and initialization
  geom.add_walls();
  std::vector<size_t> square = {6,6};
  std::vector<size_t> array = {3, Nx-2, 8, Ny-5, 10, 10};
  geom.add_array(square, array, "square");
  geom.write("./arrays/rectangular/geom_squares.txt");
  fluid.simple_ini(geom, rho_0);
  fluid.write_f("./arrays/rectangular/f_ini_rarr.txt");

  // Simulation
  int step = 1;
  sim_start = std::time(nullptr);
  while (step <= max_steps){
    lbm.f_equilibrium();
    lbm.collide(fluid);
    lbm.add_volume_force(geom, fluid, force);
    lbm.stream(geom, fluid);
    step++;
  }
  sim_end = std::time(nullptr);
  std::cout << "Simulation time: " << std::difftime(sim_end, sim_start) << std::endl;

  // Save final data
  fluid.write_rho(file_rho);
  fluid.write_ux(file_ux);
  fluid.write_uy(file_uy);
}
```

Listing 4.2: Efficient version of array code.

5. Compilation instructions

5.1 Compilation

To compile current version of the program make sure that header paths are correctly specified in the main program, then run,

```
g++ -std=c++11 -O3 -o exe_name test.cpp
```

where `test.cpp` is the name of the main file and `exe_name` the name of the executable. Run as `./exe_name`. The header path does not need to be absolute (it is not in the current examples). The compilation is valid for clang and linux. The version of the compiler used has to support C++11 standard. This standard has to be used otherwise the program will fail (to compile). `-O3` is the optimization level used by all the benchmarks in Part 4. It speeds things up considerably.