Technische Universität München
Institut für Informatik
Prof. Dr. Hans-Joachim Bungartz
Benjamin Uekermann, Martin Schreiber, Christoph Kowitz

Summer Term 2013
Worksheet 3

Praktikum Wissenschaftliches Rechnen
Computational Fluid Dynamics

Worksheet 3

Arbitrary Geometries with Navier-Stokes Equations or the Lattice-Boltzmann method

Deadline: June 3rd, 10 AM

In the third part of this Lab Course, either the Navier-Stokes solver *or* the Lattice-Boltzmann solver will be extended, such that it can work on arbitrary domain geometries and apply various boundary conditions.

What we have in mind is that the program reads all physical and numerical parameters, together with the boundary conditions, from a single input file `problem.dat`. The program should then be started by the command "'`sim problem`"'. Changes of parameters or boundary conditions should no longer require modifications in the source code, nor recompilation of the program.

Exercises

You have to choose *one* solver, in which you will implement the arbitrary geometries. You can either extend the Navier–Stokes solver from Worksheet 1 or the LBM solver from worksheet 2.

1. Implement the modifications to the program described in the Navier–Stokes or the Lattice–Boltzmann part (new boundary conditions and complex geometries). Note that the part on the Navier–Stokes solver contains some useful information for the Lattice–Boltzmann solver as well. Demonstrate, that the program works properly by providing solutions of the examples shown in the last section.

2. Implement the possibility to perform computations of the problems that are defined by setting the boundary pressure. Check your program with the plane shear flow and the flow over a step.

3. With the help of ParaView, visualize the Karman vortex street and the flow over a step. Use the features provided by ParaView excessively to analyze the flows.

1 Arbitrary Geometries for the Navier–Stokes Equations

1.1 Other Boundary Conditions

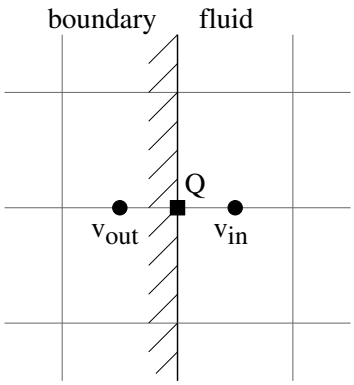
In this section, we shall treat again the boundary conditions already discussed in Worksheet 1. We shall deal with the question how reasonable boundary conditions for the velocity can be implemented. In particular, we will introduce three new types of boundary conditions: *free-slip*, *inflow*, and *outflow* conditions.

Free-slip Conditions:

Free-slip conditions model the case when the fluid can flow freely parallel to the domain boundary, but cannot cross the boundary. As a consequence, the velocity component normal to the wall should be zero, as well as the normal derivative of the velocity component parallel to the wall. In our rectangular domain that has been discretized with the help of a staggered grid, the discrete velocity components normal to the wall lie directly at the boundary. Thus, we may set (just as for the no-slip condition):

$$\begin{array}{llll} u_{0,j} = 0, & u_{imax,j} = 0, & & j = 1, \ldots, jmax; \\ v_{i,0} = 0, & v_{i,jmax} = 0, & & i = 1, \ldots, imax \end{array} \tag{1.1}$$

(assuming free-slip conditions at all four boundaries).



The normal derivative $\partial v/\partial n$ of the tangential velocity at a boundary point Q may be discretized by the expression $(v_i - v_a)/\delta x$, so that the requirement $\partial v/\partial n = 0$ leads to the condition

$$v_a = v_i.$$

We thus obtain the further boundary conditions

$$\begin{array}{llll} v_{0,j} = v_{1,j}, & v_{imax+1,j} = v_{imax,j}, & & j = 1, \ldots, jmax; \\ u_{i,0} = u_{i,1}, & u_{i,jmax+1} = u_{i,jmax}, & & i = 1, \ldots, imax \end{array} \tag{1.2}$$

(again, for all four boundaries).

Outflow Conditions:

For outflow boundary conditions, the normal derivatives of both velocity components are set to zero at the boundary, which means that the total velocity does not change in the direction normal to the boundary. This can be realized in the discrete grid by setting the velocity values at the boundary equal to their neighboring velocities inside the region, i.e.,

$$\begin{array}{llll} u_{0,j} = u_{1,j}, & u_{imax,j} = u_{imax-1,j}, & & \\ v_{0,j} = v_{1,j}, & v_{imax+1,j} = v_{imax,j}, & j = 1, \ldots, jmax; & \\ & & & \\ u_{i,0} = u_{i,1}, & u_{i,jmax+1} = u_{i,jmax}, & & \\ v_{i,0} = v_{i,1}, & v_{i,jmax} = v_{i,jmax-1}, & i = 1, \ldots, imax. & \end{array} \tag{1.3}$$

Inflow Conditions:

For inflow boundary conditions, the velocities on an inflow boundary are explicitly given. However, since these velocities take diverse values in different examples and are not always constant at the whole boundary, we cannot read these values from the input file in a simple way. Instead, we will set these boundary conditions, according to the considered problem, in a function `spec_boundary_val` that will be called each time directly after the function `boundary_val`.

Remark:

The boundary conditions for p, F and G have the same form as for no-slip conditions in Worksheet 1 (formulas (16) and (17)), regardless of whether we adopt free-slip, outflow, or inflow conditions.

Implementation:

The program now needs to be extended in a way that allows us to specify the boundary condition for each of the four sides of our domain. Therefore, we shall introduce integer parameters `wl`, `wr`, `wt` and `wb` for the left, right, upper, and lower boundary, respectively. The initial values of these parameters should be read from the input file by the function `read_parameters`.

The parameters `wl, wr, wt` and `wb` can take the values

- 1 for no-slip conditions,
- 2 for free-slip conditions, or
- 3 for outflow conditions.

For inflow conditions, using the options `wl,wr,wt` and `wb` is not sufficient, as the velocities have to be set to specific values determined by the respective problem (for example, corresponding to a parabolic or constant inflow profile). Instead, the distribution of velocities at such boundaries should be produced by a separate procedure `spec_boundary_val`. To decide which boundaries are inflow boundaries, the procedure `spec_boundary_val` uses the newly introduced variable `char* problem` that specifies the considered problem scenario.

In addition, this new procedure `spec_boundary_val` allows us to specify any kind of special boundary conditions. These can even act only on a specific part of a boundary and override the settings given by the parameters `wl, wr, wt` and `wb`.

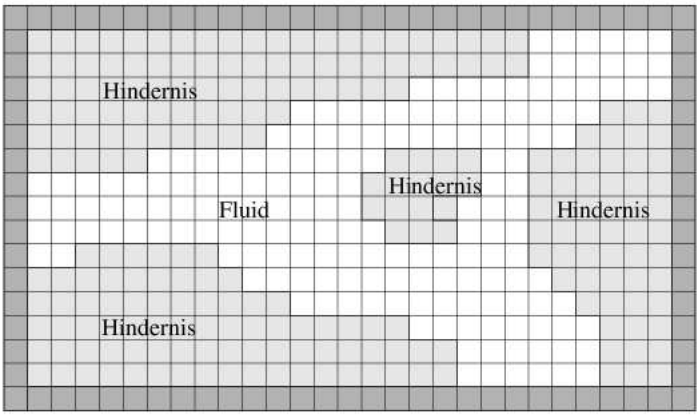


Fig. 1: Embedding of an arbitrary domain into a rectangular domain.

Assignments:

1. Supplement the function `read_parameters` such that it reads initial values for the variable `problem` and for the boundary value options `wl,wr,wt` and `wb`.

2. Extend the function `boundary_values` in such a way that, for each initial distribution of boundary value parameters `wl, wr, wt` and `wb`, with the help of `if` or `switch` command, the correct boundary values for u and v are set according to the formulas above.

3. Write (in `boundary.c`) the procedure `void spec_boundary_val (char *problem, int imax, int jmax, double **U, double **V)`, where, depending on the considered problem (given by the variable `problem`), additional boundary conditions are implemented. This includes any kind of inflow conditions, and also special settings, as they are for example present in the driven cavity example (setting $u = 1$ at the upper boundary).

1.2 Treatment Of General Geometries

Up to now, we have described the simulation of flows in rectangular domains. A simple extension will enable us to approximately treat flows in arbitrary two-dimensional geometries. For this, we embed the flow region Ω in a rectangle $\mathcal{R}$ of the smallest possible size, which we shall cover with a grid, as described in the previous worksheets. The cells of $\mathcal{R}$ are then classified into *fluid cells* (that lie completely or mostly in Ω) and *obstacle cells* (which lie completely or mostly in the obstacle region, $\mathcal{H} := \mathcal{R} \setminus \Omega$). The fluid problem is then to be solved only in the fluid cells. Obstacle cells which share an edge (*boundary edge*) with at least one fluid cell are denoted as *boundary cells*. They play a similar role like the cells of the artificial cell layers at the outer boundaries. By the way, the cells of the outer boundary layers are also denoted as obstacle cells. A domain Ω with an arbitrary curved boundary is thus approximated by a domain $\tilde{\Omega}$ whose boundary is specified by the set of boundary edges lying on grid lines (see Figure 1).

Boundary Conditions:

In fluid cells adjacent to obstacle cells, we need the following boundary values in order to compute F and G according to (9) and (10) from Worksheet 1:

- values of the normal velocity components at the boundary edges and

- values of the tangential velocity components on edges between two boundary cells.

We will only implement no-slip conditions at the internal boundary edges. In the example shown in Figure 2, the boundary edges are marked with a square, whereas the edges between two boundary cells are marked with a circle.

Furthermore, to compute the pressure at the centers of these fluid cells, the F and G values at the boundary edges are needed, as well as the pressure values at the centers of the adjacent boundary cells (see the cross in Figure 2).

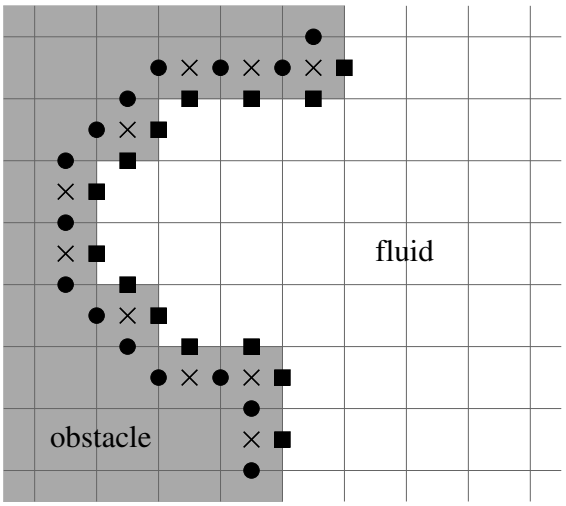


Fig.2: Required boundary values

To classify the fluid cells, we use an integer array `int **Flag`, which can be initialized as follows:

- `C_F` for a fluid cell and
- `C_B` for an obstacle cell.

The macros `C_B` and `C_F` denote integer constants that can be chosen arbitrarily.

To implement the initialization of boundary values, the set of boundary cells must be further classified: in the cell flag array we will store which of the four neighboring cells are fluid cells. Thus, the macro value `B_O` designates e.g. a boundary cell whose right (eastern, "Ost") neighbor belongs to $\tilde{\Omega}$. Similar, `B_SW` may denote a boundary cell whose lower (southern) and left (western) neighbors belong to $\tilde{\Omega}$. We assume that only boundary cells with one neighboring fluid cell [1] (*edge cells*) or two neighboring fluid cells sharing a corner (*corner cells*) may occur. Boundary cells with two opposite or even three or four neighboring fluid cells are excluded (forbidden boundary cells), because these no longer allow uniquely defined boundary values. The accuracy of representing the geometry is thus limited by the double cell length.

In accordance to the formulas (14), (15), (16) and (17) of Worksheet 1, we set for a northern edge cell (i, j) having the flag `B_N`

$$\begin{array}{lll} v_{i,j} = 0, & u_{i-1,j} = -u_{i-1,j+1}, & u_{i,j} = -u_{i,j+1}, \\ G_{i,j} = v_{i,j}, & p_{i,j} = p_{i,j+1} & \end{array} \tag{1.4}$$

or, for a western edge cell (i, j) with flag `B_W`

[1] We consider cells to be *neighboring* if they share a common edge, not only a corner.

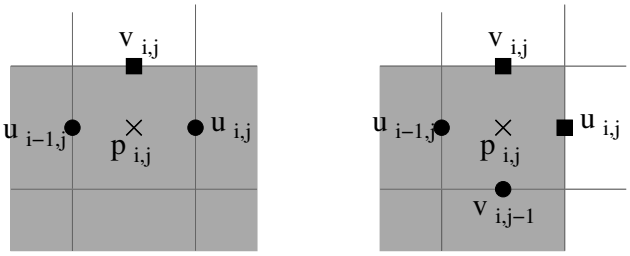


Fig. 3: Setting boundary values at obstacle cells (left: edge cell `B_N`, right: corner cell `B_NO`

$$\begin{array}{lll} u_{i-1,j} = 0, & v_{i,j-1} = -v_{i-1,j-1}, & v_{i,j} = -v_{i-1,j}, \\ F_{i-1,j} = u_{i-1,j} & p_{i,j} = p_{i-1,j}. & \end{array} \tag{1.5}$$

The boundary values for `B_S` and `B_O` cells are set analogously.

For corner cells, we apply the condition to the normal velocity component at the two edges, and the condition for the tangent velocity component at the remaining two edges. Thus, e.g. for a cell (i, j) with flag `B_NO`

$$\begin{array}{lll} u_{i,j} = 0, & u_{i-1,j} = -u_{i-1,j+1}, & F_{i,j} = u_{i,j}, \\ v_{i,j} = 0, & v_{i,j-1} = -v_{i+1,j-1}, & G_{i,j} = v_{i,j}, \\ & p_{i,j} = (p_{i,j+1} + p_{i+1,j})/2. & \end{array} \tag{1.6}$$

The solution of the pressure equation (Worksheet 1, formula (11)) is constrained to the fluid cells, whereas the computation of the F and G values according to (9) and (10), Worksheet 1, as well as the correction to the velocity value according to (7), (8), Worksheet 1, takes place only on edges between two fluid cells.

As a summary, we shall list once more all cell and edge definitions:

- fluid cell: cell lying completely or mostly in Ω
- obstacle cell: cell lying completely or mostly in $\mathcal{R} \setminus \Omega$
- boundary cell: obstacle cell bordering on one or more fluid cells
- edge cell: boundary cell bordering on exactly one fluid cell
- corner cell: boundary cell bordering on two fluid cells sharing a corner
- forbidden boundary cell: boundary cell that is neither corner, nor edge cell
- boundary edge: edge between a fluid cell and a boundary cell

Assignments:

The integer array `int **Flag` of dimension `[0,imax+1]x[0,jmax+1]` is now needed as an additional data structure. Its entries are set depending on the parameter `problem`. It is convenient to use a binary representation for the possible states, assigning one bit to each cell and its four neighbors, e.g.

center	east	west	south	north

Each bit can be set to either 1, when the corresponding cell is a fluid cell, or 0, when it is an obstacle cell. The cells in the interior of the obstacle would thus carry the flag 00000, boundary cells (including forbidden ones), a flag between 00001 and 01111, fluid cells bordering on obstacle cells, a flag between 10000 and 11110, and cells in the interior of the fluid region receive a flag value

of 11111. The flag `B_SO`, for example, corresponds to a bit coding of 01010 i.e. the decimal value of 10. Using the bit operators available in the C programming language, these distinctive cases can be formulated in a quite elegant fashion.

The function `int **imatrix(int nrl, int nrh, int ncl, int nch)` from `helper.c` will generate an array of integers, which is suitable to hold the flag field describing the fluid region. Based on this data structure, you should implement the following new function (in the file `init.c`):

- `void init_flag(problem,imax,jmax,Flag)`
  The array `Flag` is initialized with the flags `C_F` for fluid cells and `C_B` for obstacle cells as specified by the parameter `problem`. This must be followed by a loop over all cells where the boundary cells are marked with the appropriate flags `B_xy` depending on the direction, in which neighboring fluid cells lie.

The function `void init_flag` has to be called in the initialization part of the main program. In addition, the following functions must be adjusted:

- In `boundary_values` and `calculate_fg`, boundary values must be assigned to the boundary cells defined by the flag array, as described above.
- In `SOR`, apart from the boundary values for the pressure at the boundary stripe, also the boundary values in the interior boundary cells should be set before each iteration step.
- In `SOR`, the iteration and the residual computation needs to be limited to the fluid cells, whereas F and G values in `calculate_fg` and velocities in `calculate_uv` are only calculated on edges separating two fluid cells. Besides, the normalization of the residual in `SOR` should be produced by dividing by the current number of fluid cells, and not by $imax \cdot jmax$.

1.3 Boundary Conditions For Pressure

In numerous cases, the driving force for the flow is the one of the pressure. An example is provided by air bubbles passing through a straw. Here, one produces the pressure difference between the air in the mouth and that in the environment. This results in an air flow directed against this gradient. Quite a lot of technologically important flows are produced by pressure differences.

Besides, for the numerical simulation, taking the pressure difference as the boundary condition is a great simplification. Actually, one can as well simulate each pressure-driven flow by imposing the velocity boundary conditions, and obtain the same results as by stipulating the pressure difference. However, the velocity boundary values are in most cases unknown. Often also the pressure difference must be calculated, the one that should be applied to produce a predetermined volume flow in a device.

In case we want to apply the pressure difference Δp to the considered domain, we can implement the former in our simulation by imposing Dirichlet boundary conditions for the pressure at the corresponding boundaries.

Since no points with a definite pressure value lie at the physical boundary of the domain, this value must be defined by interpolation (like for the velocities parallel to the wall). If we denote the pressure at the right boundary by p_w, then one can formulate the following relationship between the pressure p_{imax+1} at the boundary outer edge, the last value p_{imax} in the pressure array and p_w:

$$p_w = \frac{p_{imax} + p_{imax+1}}{2} \tag{1.7}$$

This equation, being solved with respect to p_{imax+1}, must be inserted into the system of equations for the pressure values. It replaces the old condition for pressure at the boundary, where the pressure at the outer edge was set equal to that in neighboring cells lying inside the domain.

If we specify a fixed value for the pressure at a boundary (Dirichlet condition), the corresponding velocity should obey to a Neumann boundary condition (equivalent to our *outflow* condition). To decide whether certain velocity values have to be computed at a boundary, we have already introduced the flag field `**Flag`. Hence, we also use the flag field to implement the pressure boundary conditions. Within the framework of this Lab Course, it will be sufficient to apply the pressure boundary values just for two sides of the considered domain, left and right. Therefore, we set one bit in the flag field for the case when the pressure value is set at the (boundary) cell lying at the right boundary and another bit corresponding to the left boundary.

The decision in which cells the appropriate bit for the pressure boundary conditions will be set is contained in the function `init_flag`, which also holds the other bits. The pressure values for both boundaries as well as the pressure difference (in case the pressure always remains constant at one boundary) should be read by the parameter file.

Assignments:

1. Make sure the initialization of appropriate bits is produced in `init_flag` at boundary cells.
2. Read the pressure difference in `read_parameters`.
3. Modify the `SOR` function in such a way that instead of the Neumann boundary conditions for the pressure the correct Dirichlet boundary conditions will be set.

Hint:

The file `helper.c` contains a function `read_pgm(filename)` that reads in an ASCII-encoded pgm-file. With that you can paint your own geometries easily and also impose the boundary conditions of an arbitrary domain. For details on the function, see the documentation in the code; for details on the pgm-format, see for example http://de.wikipedia.org/wiki/Portable_Graymap.

1.4 Example Problems

With the help of these modifications of the flow program, we are now able to compute a whole bunch of new examples. We only need to set up a new input file for each example. Besides, the problem-specific obstacle cells should be implemented in the function `init_flag` as well as the inflow conditions and further special boundary conditions in the function `spec_boundary_val`.

a) The Karman vortex street:

The flow in a channel hits a tilted plate. At the left boundary, the fluid inflow has a constant velocity profile ($u = 1.0$, $v = 0.0$), while at the upper and lower boundaries no-slip conditions are imposed.

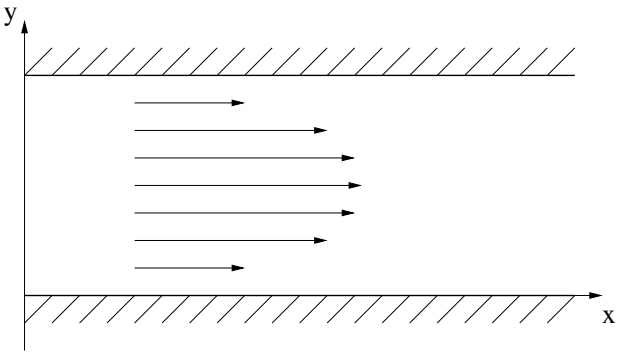
The plate occupies one fifth of the channel width and is three cells thick. The distance from the left boundary is assumed to be the same as the distance to the lower and upper boundaries. Make sure that there are no forbidden obstacle cells.

Use the following values for the parameters in case you use the Navier-Stokes solver:

imax = 100	jmax = 20	xlength = 10	ylength = 2
dt = 0.05	t_end = 20	tau = 0.5	dt_value = 2.0
eps = 0.001	omg = 1.7	alpha = 0.9	itermax = 500
GX = 0.0	GY = 0.0	Re = 10000	
UI = 1.0	VI = 0.0	PI = 0.0	
wl = 1.0	wr = 3.0	wt =1.0	wb =1.0

Use the tools provided by paraView to characterize the flow around the obstacle.

b) Plane shear flow:



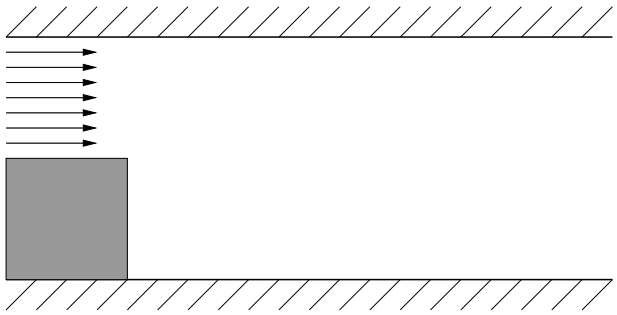
The plane shear flow is a favorite example problem to test the numerical methods and their functionality, since it is possible to give an analytic solution to this problem:

$$\begin{aligned} u(y) &= -\frac{1}{2} Re \cdot \frac{\Delta p}{\Delta x} \cdot y \cdot (y - h) \\ v(y) &= 0 \end{aligned}$$

You can use, for example, the following parameters for Navier–Stokes with no-slip boundaries:

imax = 100	jmax = 20	xlength = 10	ylength = 2
dt = 0.05	t_end = 30	tau = 0.5	dt_value = 5.0
eps = 0.001	omg = 1.7	alpha = 0.9	itermax = 500
GX = 0.0	GY = 0.0	Re = 10	Δp = 4.0
UI = 1.0	VI = 0.0	PI = 0.0	
wl = 3	wr = 3	wt = 1	wb = 1

c) Flow over a step:



The fluid flows through a channel widening on one side. No-slip conditions are imposed at the upper and lower walls.

The obstacle domain is represented by a square filling up half of the channel height. Use the following values for the problem parameters:

imax = 100	jmax = 20	xlength = 10	ylength = 2
dt = 0.05	t_end = 500	tau = 0.5	dt_value = 10.0
eps = 0.001	omg = 1.7	alpha = 0.9	itermax = 500
GX = 0.0	GY = 0.0	Re = 100	
UI = 1.0	VI = 0.0	PI = 0.0	
wl = 3	wr = 3	wt = 1	wb = 1

While initializing the velocity values in `init_uvp`, in the lower half-part of the channel the value u might be set to 0 instead of 1.

Please bring your computer again or prepare one of the computers in the lab to speed up the review process.

2 Arbitrary Geometries for the Lattice–Boltzmann Method

2.1 New Boundary Conditions

As for the Navier–Stokes equation other boundaries as the *no-slip* boundaries can be imposed in the Lattice-Boltzmann method. Due to the nature of the LBM, these can also be computed mostly locally without any global communication between cells. All boundary conditions here are handled similarly as in the previous worksheet by traversing over all non-fluid cells and reconstructing the distributions functions pointing into the fluid domain.

Free–Slip Boundaries

The free-slip boundaries are closely related to the the no-slip boundary, as it can be seen in Fig. 2.1. Note that the implementation of these boundaries requires more effort and communication, since the particle distribution functions are not copied into the ones opposing them in the boundaries but they are mirrored into the boundary cell. By doing so the velocity tangential to the boundary is preserved. Care has to be taken at corners, since there the "mirrored" particle distributions will pass the boundary.

Outflow Conditions

In order to create an open boundary, through which the fluid can exit the domain freely, the distribution function coming in from the outside has to represent the region behind the domain wall. To preserve the velocity $\vec{v}$ of the fluid at the boundary, the incoming particle distribution functions are reconstructed by

$$f_i^*(\vec{x}_B, t) = f_{\mathrm{inv}(i)}^{\mathrm{eq}}(\rho_{\mathrm{ref}}, \vec{v}) + f_i^{\mathrm{eq}}(\rho_{\mathrm{ref}}, \vec{v}) - f_{\mathrm{inv}(i)}^*(\vec{x}_B + \vec{c}_i \mathrm{d}t, t), \tag{2.1}$$

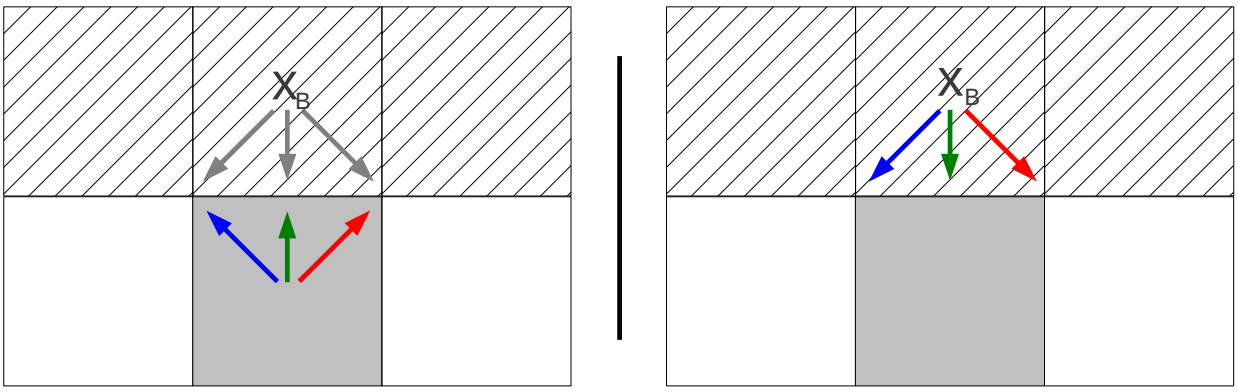


Figure 2.1: Free-slip boundaries for the Lattice-Boltzmann method.

with $\rho_{\text{ref}} = 1$ being the reference density and $\vec{v} = \vec{v}(\vec{x}_{\text{B}} + \vec{c}_i dt, t)$ the velocity of the fluid in the cell the pdf will stream into. Thus a force is created, which is acting in the opposite direction of the density gradient.

Inflow Conditions

For inflow conditions, the particle distribution functions pointing from the boundary cells into the domain are reconstructed using the equilibrium distribution function computed from the reference density ρ_{ref} and the inflow velocity $\vec{v}_{\text{in}}$.

$$f_i^*(\vec{x}_B, t) = f_i^{\text{eq}}(\rho_{\text{ref}}, \vec{v}_{\text{in}}) \tag{2.2}$$

By using this approach a certain velocity profile is enforced. Another way of creating an inflow of fluid would be to create a boundary similar to the outflow boundary, but instead of setting the standard reference density there, a higher density $\rho_{\text{in}} = \rho_{\text{ref}} + \Delta\rho$ is imposed. Due to the resulting gradient in density and thus pressure, an inflow of fluid results. Note that, the pressure gradient is chosen in a limit, such that the resulting velocity of the fluid is still far below the Mach number.

2.2 Arbitrary Geometries

As in the previous worksheet the `*flagField` is used to distinguish fluid cells and the different types of boundary cells. Contrary to the arbitrary geometries in the Navier-Stokes case, no special treatment of the cells concerning the different directions of the boundary is done, since every boundary cell is checking its neighbors for its state. New flags `FREE_SLIP`, `INFLOW`, `OUTFLOW` and `PRESSURE_IN` have to be introduced and assigned to the respective cells. No distinction between domain boundaries and obstacle boundaries is required, so that only one initialization of the `*flagField` containing the respective boundary/ cell types is required. Care has to be taken for the design of obstacles since particle distribution functions are also propagated diagonally. In order to create a non-penetrable object at least two layers of obstacle cells have to be chosen.

2.3 Programming Assignments

1. Change the current LBM code, so that is able to handle domains of different sizes in x, y and z. Thus the variable `xlength` has to be substituted by `xlength[3]` at the appropriate positions in the code. Do not forget to change the signatures of functions as well (`int xlength` to `int *xlength`).

2. Add the handling of the boundary cells of type `FREE_SLIP`, `INFLOW`, `OUTFLOW` and `PRESSURE_IN` in the function `void treatBoundary()`. The input parameter `wallVelocity` is to be substituted by an array of parameters, which are required to set the inflow velocity and the artificial input pressure for a pressure inlet. Note that `readParameters()` has to be updated accordingly to be able to steer the simulation without the need for recompilation.

3. Adjust `initialiseFields()`, so that the `*flagField` is set with the appropriate flags according to the desired geometry. The signature of `initialiseFields()` is extended by `*char problem`, which contains the name of a chosen scenario. In `initialiseFields()` the `*flagField` is then set according to a given scenario. Note that this scenario should be created independent from the used domain size `*xlength` to allow a change of resolution of the simulation. Furthermore `read_pgm` *can* be used to initialize the `*flagField` to create scenarios more easily.

2.4 Example Problems

With the help of these modifications of the flow program, we are now able to compute a whole bunch of new examples. We only need to set up a new input file for each example.

a) A tilted plate:

The flow in a channel hits a tilted plate. At the left boundary, the fluid inflow has a constant velocity profile, while at the upper, lower and side boundaries no-slip conditions are imposed.

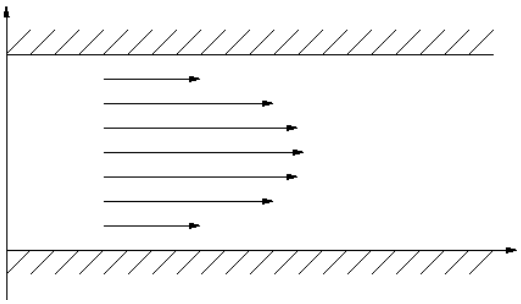
The plate occupies three fifth of the channel width in x-direction and is three cells thick. The distance from the inflow boundary is assumed to be the same as the distance to the lower and upper boundaries. In y-direction the obstacle is attached to one of the walls and it has a length of two thirds of the width in y-direction. The normal direction of the plate is $(1, 0, -1)$. Make sure that there are no forbidden obstacle cells.

Try to set up this kind of domain in 3D with no-slip boundaries on all of the channel boundaries and a no-slip obstacle. The xy faces are the in or outflow respectively.

xlength=30	$\vec{v}_{\text{in}} = 0.025 \cdot \vec{e}_z$
ylength=30	$\tau = 0.6$
zlength=70	t=1000

Use the tools provided by paraView to characterize the flow around the obstacle.

b) Plane shear flow:



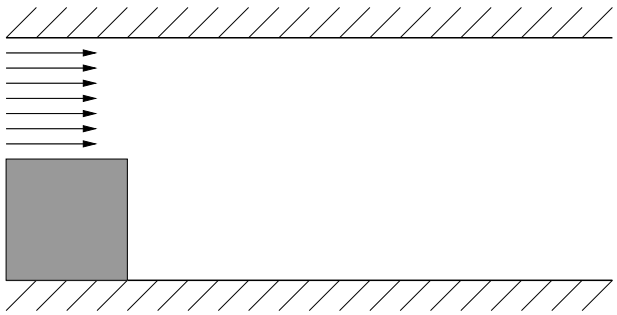
The plane shear flow is a favorite example problem to test the numerical methods and their functionality, since it is possible to give an analytic solution to this problem:

$$v_z(y) \quad = \quad -\frac{1}{2} Re \cdot \frac{\Delta p}{\Delta x} \cdot y \cdot (y - h)$$

For the LBM solver set up a 3D domain with no-slip boundaries for the xz faces, free-slip boundaries for the yz faces and pressure in and outlet for the xy faces. Again a comparable velocity profile should appear.

xlength=8	ylength=8	zlength=20
$\rho_{\text{ref}} = 1.0$	$\rho_{\text{in}} = 1.01$	$\tau = 2.0$
$t = 1000$		

c) Flow over a step:



The fluid flows through a channel widening on one side. No-slip conditions are imposed at the channel walls and on the object.
following values for the problem parameters: The obstacle domain is represented by a square filling up half of the channel height. Use the

For the LBM simulation the geometry is extended to 3D by extending the step over the whole third dimension. You might use these parameters:

xlength=30	ylength=30	zlength=50
$\vec{v}_{\text{in}} = 0.05$	$\tau = 0.6$	$t = 3000$