Palabos Tutorial

Release 1.0r0

Jonas Latt

Jun 12, 2020

CONTENTS

Introductory tutorials:

CHAPTER

ONE

INTRODUCTORY COMMENTS

The following tutorial provides an overview of the C++ programmer interface of the Palabos library. If instead you are interested in the Python interface, have a look at the Palabos-Python tutorial from DSFD 2010 (PDF). As for the Java interface, no tutorial is available at this moment. The user's guide provides however installation instructions, and you can then have a look at the provided example applications.

CHAPTER

TWO

TUTORIAL 1: FIRST STEPS WITH PALABOS

2.1 Tutorial 1.1: The first code

To get started under Linux or Mac OS X you need to have the C++ frontend of GCC (`g++`) installed. Under Mac OS X, a convenient way to get GCC is to install xcode. If any of the examples fails to compile under Mac OS X, edit the Makefile, and add the option `-DPLB_MAC_OS_X` to the entry `compileFlags = ...`.

Note: Windows programmers

Under Windows, you can get started with Palabos in two different ways: either you install the Code::Blocks (code-blocks) programming environment (choose the download that is packaged with the MinGW library), or you run Linux in a virtual machine. The first approach is easier and sufficient for the purposes of this tutorial. The second approach is however more convenient in the long run, because currently, not all features of Palabos (example: the Python and the Java bindings) are available under Windows.

If you choose to work with Code::Blocks, the following command-line instructions do not apply. Instead, use the Code::Blocks IDE and create a project for each code of the tutorial.

Download the tarball of the most recent version of Palabos, for example `palabos-1.0r0.tgz`, and unpack it with the command `tar xvfz palabos-1.0r0.tgz`. To use Palabos, you do not need to proceed with an explicit compilation and installation of the library. Instead, the library is compiled on-demand when end-user applications are created. This approach reflects the fact that many Palabos users are also developers, and end-user development is coupled with development of the source core. Furthermore, the approach makes it easy to recompile the code on-demand with or without debug flags, or for sequential/parallel execution. Finally, it makes it simple to use Palabos on a machine on which you have no administrator rights.

Change into the directory of the tutorial, `palabos/examples/tutorial/section_1`, and type `make` to compile the library and create the executable of the first tutorial code, `tutorial_1_1.cpp`. Finally, the application is executed by typing `./tutorial_1_1` at the command line.

This application simulates the time evolution of an initial-value problem, without boundary conditions. The boundaries are chosen to be periodic, i.e. flow particles leaving the domain on one boundary re-enter the domain on the opposite boundary. The initial condition has a zero velocity and constant density. On a squared sub-domain of the field, the density is initialized at a slightly higher value, to create a perturbation which then generates a flow pattern.

This setup is chosen mostly to illustrate programming concepts in Palabos, and not to propose an interesting hydrodynamic problem. It should in particular be pointed out that the initial condition does not represent the state of an incompressible fluid, because the zero velocity is incompatible with a non-zero density. After sufficient iterations, the flow automatically converges to a situation which is compatible with the physics of an incompressible, or slightly compressible, flow.

The code `tutorial_1_1.cpp` is listed below:

```
/* Code 1.1 in the Palabos tutorial
 */

#include "palabos2D.h"
#include "palabos2D.hh"
#include <iostream>
#include <iomanip>

using namespace plb;
using namespace std;


typedef double T;
#define DESCRIPTOR plb::descriptors::D2Q9Descriptor


// Initialize the lattice at zero velocity and constant density, except
//   for a slight density excess on a square sub-domain.
void defineInitialDensityAtCenter(MultiBlockLattice2D<T,DESCRIPTOR>& lattice)
{
    // The lattice is of size nx-by-ny
    const plint nx = lattice.getNx();
    const plint ny = lattice.getNy();

    // Create a Box2D which describes the location of cells with a slightly
    //   higher density.
    plint centralSquareRadius = nx/6;
    plint centerX = nx/3;
    plint centerY = ny/4;
    Box2D centralSquare (
            centerX - centralSquareRadius, centerX + centralSquareRadius,
            centerY - centralSquareRadius, centerY + centralSquareRadius );

    // All cells have initially density rho ...
    T rho0 = 1.;
    // .. except for those in the box "centralSquare" which have density
    //    rho+deltaRho
    T deltaRho = 1.e-4;
    Array<T,2> u0(0,0);

    // Initialize constant density everywhere.
    initializeAtEquilibrium (
           lattice, lattice.getBoundingBox(), rho0, u0 );

    // And slightly higher density in the central box.
    initializeAtEquilibrium (
           lattice, centralSquare, rho0 + deltaRho, u0 );

    lattice.initialize();
}

int main(int argc, char* argv[]) {
    plbInit(&argc, &argv);
    global::directories().setOutputDir("./tmp/");

    const plint maxIter = 1000; // Iterate during 1000 steps.
    const plint nx = 600;       // Choice of lattice dimensions.
    const plint ny = 600;
```

```
59      const T omega = 1.;         // Choice of the relaxation parameter
60
61      MultiBlockLattice2D<T, DESCRIPTOR> lattice (
62              nx, ny, new BGKdynamics<T,DESCRIPTOR>(omega) );
63
64      lattice.periodicity().toggleAll(true); // Use periodic boundaries.
65
66      defineInitialDensityAtCenter(lattice);
67
68      // Main loop over time iterations.
69      for (plint iT=0; iT<maxIter; ++iT) {
70          if (iT%40==0) {  // Write an image every 40th time step.
71              pcout << "Writing GIF file at iT=" << iT << endl;
72              // Instantiate an image writer with the color map "leeloo".
73              ImageWriter<T> imageWriter("leeloo");
74              // Write a GIF file with colors rescaled to the range of values
75              //   in the matrix
76              imageWriter.writeScaledGif (
77                      createFileName("u", iT, 6),
78                      *computeVelocityNorm(lattice) );
79          }
80          // Execute lattice Boltzmann iteration.
81          lattice.collideAndStream();
82      }
83  }
```

2.1.1 Global definitions and includes

Line 4-7 The include file `palabos2D.h` gives access to all declarations in the Palabos project. Additionally, the file `palabos2D.hh` is included to guarantee access to the full template code, for example to instantiate simulations with different data types than the standard data type `double`. The distinction between generic and non-generic code is explained in the next tutorial.

Line 11-12 These two declarations guarantee an automatic access to the names in the Palabos code, which is contained in the namespace `plb`, and to the C++ standard library, which is contained in the namespace `std`.

Line 14-15 As specified by these two directives, the simulation will be executed with double precision floating point numbers, and on the two-dimensional D2Q9 lattice.

2.1.2 Creating the initial condition

Line 19 The data type at the heart of Palabos is the `MultiBlockLatticeXD`, where `X` stands for 2 or 3, for 2D or 3D simulations. As will be shown shortly, the word `Multi` stands for the fact that in the internal implementation, the lattice is often decomposed into several smaller lattices. On the interface however, and this abstraction mechansim can be confidently used, in a first time at least, to ignore details of the behind-the-scene action.

Line 27-32 The domain is initialized with a slightly exceeding density on a rectangular sub-domain of the full domain. This domain is defined through a geometric object of type `Box2D`.

Line 42 Initialize all lattice cells at an equilibrium distribution with constant density and zero velocity.

Line 46 Then, redefine the initial values for the cell inside the box `centralSquare` to have a density exceeding the other ones by `deltaRho`.

Line 49 The method `initialize` is called after initialization operations, to prepare the lattice for simulations.

2.1.3 Running the simulation

Line 53 The function `plbInit` must mandatorily be invoked in the very beginning of the program, to guarantee consistent results between sequential and parallel program runs.

Line 54 Specify the directory where output files will be placed.

Line 61-62 During creation of the multi block, the default dynamics (in this example BGK), also called *background dynamics*, is specified. This object defines the nature of the collision to be executed by all cells, unless a new individual dynamics is attributed to them in the simulation setup.

Line 64 Periodic boundaries are handled in a special way in Palabos. They can only be imposed on the outer bound of a block-lattice, through a call to the method `periodicity()`.

Line 73 An image writer object provides a convenient way to write GIF images from simulated data, in 2D or in 3D. In the latter case, the image represents for example a slice through the computational domain. Such images are mostly used to monitor the evolution of the simulation, and to get a qualitative idea of the state of the simulation, before proceeding to a detailed evaluation of the data. To check the images, change into the directory `tmp` during or after the simulation, and visualize them for example with help of the command `display`. To get an animation from subsequent GIF images, use the command `convert` (`convert -delay 5 u*.gif animation.gif`), and visualize the animated gif through a command like `animate animation.gif`.

Line 81 The method `collideAndStream` executes a collision and a streaming step on each cell of the lattice. The two steps can also be separated by calling first `lattice.collide()` and then `lattice.stream()`. The synchronous execution of collision and streaming is however more efficient, because it requires only a single traversal of the data space to execute a full lattice Boltzmann iteration. The argument `true` to the method `collideAndStream` or to the method `stream` is used to implement periodicity for the domain boundaries. Reversely, the argument `false` is used for non-periodic boundaries which then implement, by default, a half-way bounce-back condition. The effect of this condition is to simulate no-slip walls located half a cell spacing beyond the outer lattice sites.

This tutorial shows that the central structure in which the data of a simulation is stored, is the `MultiBlockLatticeXD`. It was chosen this way, because the interface of the `MultiBlockLatticeXD` mimicks regular data arrays with which researchers and engineers are usually familiar from the implementation of simpler problems in languages like Matlab or Fortran. Another advantage of the "regular-block interface" is the ease with which individual lattice cells are identified and treated, for example for the implementation of boundary conditions.

If you are unfamiliar with object-oriented programming, it is likely that at this point you start feeling nervous about this form of data encapsulation and the apparent loss of control over the details of the program workflow. If you stay focused, you will however progressively understand that the behind-the-scene action is simple enough and by no means hidden from the understanding of the programmer. Instead, object-oriented mechanisms make it easier to keep control over a structured and well understood course of action. The principles and mechanisms of the `MultiBlockLatticeXD` are presented and trained in the Tutorial 2. In particular, Tutorial 2.1 shows that the `MultiBlockLatticeXD` could in practice be replaced by an `AtomicBlockLatticeXD`, which stands for a simple regular array, and you should feel free to go ahead and do so. Be aware though that there are in practice only disadvantages in making this substitution, as one loses for example the possibility to parallelize the program. On the other hand, it has however been observed that the use of the simple `AtomicBlockLatticeXD` can help overcome psychological barriers by providing a sense of control in a first stage of gaining familiarity with Palabos.

2.2 Tutorial 1.2: Initializing the lattice

In the previous lesson, the initial condition was created by assigning a constant value of density and velocity to domains of rectangular shape. A more detailed initialization of lattice cells is obtained by writing functions in which a different value of density and velocity is attributed to each cell. This is illustrated in the tutorial code 1.2, `tutorial_1_2.cpp`. To compile this code, edit the file `Makefile`, and replace `tutorial_1_1` by `tutorial_1_2` in the corresponding line.

The relevant parts of the code `tutorial_1_2.cpp` are listed below:

```
1  const plint maxIter = 1000; // Iterate during 1000 steps.
2  const plint nx = 600;       // Choice of lattice dimensions.
3  const plint ny = 600;
4  const T omega = 1.;         // Choice of the relaxation parameter
5
6  T rho0 = 1.; // All cells have initially density rho ...
7  // .. except for those inside the disk which have density
8  //    rho+deltaRho
9  T deltaRho = 1.e-4;
10 Array<T,2> u0(0,0);
11
12 void initializeConstRho(plint iX, plint iY, T& rho, Array<T,2>& u) {
13     u = u0;
14     rho = rho0 + deltaRho;
15 }
16
17 void initializeRhoOnDisk(plint iX, plint iY, T& rho, Array<T,2>& u) {
18     plint radius = nx/6;
19     plint centerX = nx/3;
20     plint centerY = ny/4;
21     u = u0;
22     if( (iX-centerX)*(iX-centerX) + (iY-centerY)*(iY-centerY) < radius*radius) {
23         rho = rho0 + deltaRho;
24     }
25     else {
26         rho = rho0;
27     }
28 }
29
30 // Initialize the lattice at zero velocity and constant density, except
31 //   for a slight density excess on a circular sub-domain.
32 void defineInitialDensityAtCenter(MultiBlockLattice2D<T,DESCRIPTOR>& lattice)
33 {
34     // Initialize constant density everywhere.
35     initializeAtEquilibrium (
36             lattice, lattice.getBoundingBox(), rho0, u0 );
37
38     // And slightly higher density in the central box.
39     initializeAtEquilibrium (
40             lattice, lattice.getBoundingBox(), initializeRhoOnDisk );
41
42     lattice.initialize();
43 }
```

Line 12 The function `initializeConstRho` can be used to obtain exactly the same effect as in the previous lesson: assign a constant density to each cell of a sub-domain.

Line 17 The function `initializeRhoOnDisk` on the other hand attributes a different density only to cells contained inside a disk. The initial condition is therefore more natural, having a circular instead of a rectangular shape.

Line 39 This time, the function `initializeAtEquilibrium` is called with the function `initializeRhoOnDisk` as an argument, instead of a constant value of density and velocity. More generally, the function `initializeAtEquilibrium` behaves like an algorithm of the C++ standard template library. It's argument can be a functional in the general sense, *i.e.* either a classical function or an object which behaves like a function by overloading the function call operator. The advantage of using an object instead of a function is that objects can store internal data and can therefore have a customized behavior.

In such a case, the function `initializeRhoOnDisk` could for example be customized with respect to the center and radius of the disk. The usage of such a function object is illustrated in the tutorial code 1.4.

2.3 Tutorial 1.3: Compilation options

So far, we have compiled the programs using the default compiler options. Edit the file `Makefile` to see other available options

2.3.1 Debug mode

Debug mode is activated in Palabos by setting the flag `debug=true`. In this mode, the program executes additional error checking, which can help in locating the source of the error. Furthermore, debug information is put into the object file, after which you can use a debugger like `gdb`.

Debug mode is activated by default in all Palabos examples. As a general rule, we recommend that you keep this flag activated even in production mode. It decreases the overall performance of Palabos by a few percents only, and provides helpful insights whenever your program crashes.

2.3.2 Parallel mode

All codes presented in this tutorial work in serial and parallel. Be aware, though, that most examples perform lots of output operations, such as, the generation of GIF images. They are therefore unlikely to run significantly faster in parallel, unless you comment out the output operations. Compilation for parallel execution is achieved by selecting the parallel compiler in the line `parallelCXX=...` of the Makefile, and by setting the flag `MPIparallel=true`. The program can the be executed through a call to `mpirun`, `mpiexec`, or something similar.

Parallel compilation is activated by default in all Palabos examples: most computers nowadays have multi-core CPUs which you can exploit by running the Palabos applications in parallel.

2.4 Tutorial 1.4: Data analysis

In this lesson, a few approaches to analyzing the results of a simulation are reviewed. The corresponding code is found in the file `tutorial_1_4.cpp`

Let us look at the function `main` in this code:

```
1  int main(int argc, char* argv[]) {
2      plbInit(&argc, &argv);
3      global::directories().setOutputDir("./tmp/");
4
5      MultiBlockLattice2D<T, DESCRIPTOR> lattice (
6             nx, ny, new BGKdynamics<T,DESCRIPTOR>(omega) );
7
8      lattice.periodicity().toggleAll(true); // Set periodic boundaries.
9
10     defineInitialDensityAtCenter(lattice);
11
12     // First part: loop over time iterations.
13     for (plint iT=0; iT<maxIter; ++iT) {
14         lattice.collideAndStream();
15     }
16
```

```
17      // Second part: Data analysis.
18      Array<T,2> velocity;
19      lattice.get(nx/2, ny/2).computeVelocity(velocity);
20      pcout << "Velocity in the middle of the lattice: ("
21            << velocity[0] << "," << velocity[1] << ")" << endl;
22
23      pcout << "Velocity norm along a horizontal line: " << endl;
24      Box2D line(0, 100, ny/2, ny/2);
25      pcout << setprecision(3) << *computeVelocityNorm(*extractSubDomain(lattice,␣
    ↪line)) << endl;
26
27      plb_ofstream ofile("profile.dat");
28      ofile << setprecision(3) << *computeVelocityNorm(*extractSubDomain(lattice,␣
    ↪line)) << endl;
29
30      pcout << "Average density in the domain: " << computeAverageDensity(lattice) <<␣
    ↪endl;
31      pcout << "Average energy in the domain: " << computeAverageEnergy(lattice) <<␣
    ↪endl;
32  }
```

Line 19 The method `get` delivers access to a lattice cell. A cell is an object with which one can not only access the actual variables defined on the cell, but also perform useful computations, such as, evaluate macroscopic variables. In this example, the velocity is computed on a cell in the middle of the domain.

Line 25 With the method `extractSubDomain`, a rectangular sub domain of the full lattice is extracted. In the present example, the velocity norm is computed on the extracted domain and printed to the terminal.

Line 27 In the same way, C++ streams are used to write the data into a file instead of the terminal. Make sure to use the data type `plb_ofstream` instead of `ofstream` to guarantee working conditions in parallel programs.

Line 30-31 Many functions are predefined for computing average values, like the average density or the average kinetic energy in the present example.

The data written to the file `profile.dat` is conveniently post-processed with a mathematics processing tool. For example, type the two following lines at the command prompt of the program Octave to plot a curve of the velocity profile:

```
load profile.dat
plot(profile)
```

2.5 Tutorial 1.5: Boundary conditions

For the first time, we implement a simulation which represents a well-known physical situation: a Poiseuille flow. This flow evolves in a channel with no-slip walls, and the flow velocity is everywhere parallel to the channel walls. The analytical solution of this flow is represented by a parabolic profile for the velocity component parallel to the channel walls. In the simulation, the analytical solution is used to implement Dirichlet boundary condition for the velocity on the domain inlet and outlet. As an initial condition, a zero velocity field is used, and the simulation is started to converge towards the Poiseuille solution in each point of the domain.

Note that the simulation has initially a discontinuity between the zero velocity in the initial condition and the finite velocity in the boundary condition. If the simulation becomes numerically unstable when you play with the parameters, try either increasing the lattice resolution or decreasing the velocity in lattice units.

This time, the full code is reprinted in the tutorial:

```cpp
#include "palabos2D.h"
#include "palabos2D.hh"

#include <vector>
#include <iostream>
#include <iomanip>

/* Code 1.5 in the Palabos tutorial
 */

using namespace plb;
using namespace std;

typedef double T;
#define DESCRIPTOR plb::descriptors::D2Q9Descriptor


/// Velocity on the parabolic Poiseuille profile
T poiseuilleVelocity(plint iY, IncomprFlowParam<T> const& parameters) {
    T y = (T)iY / parameters.getResolution();
    return 4.*parameters.getLatticeU() * (y-y*y);
}

/// A functional, used to initialize the velocity for the boundary conditions
template<typename T>
class PoiseuilleVelocity {
public:
    PoiseuilleVelocity(IncomprFlowParam<T> parameters_)
        : parameters(parameters_)
    { }
    /// This version of the operator returns the velocity only,
    ///    to instantiate the boundary condition.
    void operator()(plint iX, plint iY, Array<T,2>& u) const {
        u[0] = poiseuilleVelocity(iY, parameters);
        u[1] = T();
    }
    /// This version of the operator returns also a constant value for
    ///    the density, to create the initial condition.
    void operator()(plint iX, plint iY, T& rho, Array<T,2>& u) const {
        u[0] = poiseuilleVelocity(iY, parameters);
        u[1] = T();
        rho  = (T)1;
    }
private:
    IncomprFlowParam<T> parameters;
};

void channelSetup (
        MultiBlockLattice2D<T,DESCRIPTOR>& lattice,
        IncomprFlowParam<T> const& parameters,
        OnLatticeBoundaryCondition2D<T,DESCRIPTOR>& boundaryCondition )
{
    // Create Velocity boundary conditions.
    boundaryCondition.setVelocityConditionOnBlockBoundaries(lattice);

    // Specify the boundary velocity.
    setBoundaryVelocity (
            lattice, lattice.getBoundingBox(),
```

```
            PoiseuilleVelocity<T>(parameters) );

    // Create the initial condition.
    initializeAtEquilibrium (
           lattice, lattice.getBoundingBox(), PoiseuilleVelocity<T>(parameters) );

    lattice.initialize();
}

void writeGifs(MultiBlockLattice2D<T,DESCRIPTOR>& lattice, plint iter)
{
    const plint imSize = 600;
    ImageWriter<T> imageWriter("leeloo");
    imageWriter.writeScaledGif(createFileName("u", iter, 6),
                               *computeVelocityNorm(lattice),
                               imSize, imSize );
}

int main(int argc, char* argv[]) {
    plbInit(&argc, &argv);

    global::directories().setOutputDir("./tmp/");

    // Use the class IncomprFlowParam to convert from
    //   dimensionless variables to lattice units, in the
    //   context of incompressible flows.
    IncomprFlowParam<T> parameters(
            (T) 1e-2,  // Reference velocity (the maximum velocity
                       //   in the Poiseuille profile) in lattice units.
            (T) 100.,  // Reynolds number
            100,       // Resolution of the reference length (channel height).
            2.,        // Channel length in dimensionless variables
            1.         // Channel height in dimensionless variables
    );
    const T imSave   = (T)0.1;  // Time intervals at which to save GIF
                                //   images, in dimensionless time units.
    const T maxT     = (T)3.1;  // Total simulation time, in dimensionless
                                //   time units.

    writeLogFile(parameters, "Poiseuille flow");

    MultiBlockLattice2D<T, DESCRIPTOR> lattice (
             parameters.getNx(), parameters.getNy(),
             new BGKdynamics<T,DESCRIPTOR>(parameters.getOmega()) );

    OnLatticeBoundaryCondition2D<T,DESCRIPTOR>*
        boundaryCondition = createLocalBoundaryCondition2D<T,DESCRIPTOR>();

    channelSetup(lattice, parameters, *boundaryCondition);

    // Main loop over time iterations.
    for (plint iT=0; iT*parameters.getDeltaT()<maxT; ++iT) {
        if (iT%parameters.nStep(imSave)==0 && iT>0) {
            pcout << "Saving Gif at time step " << iT << endl;
            writeGifs(lattice, iT);
        }
        // Execute lattice Boltzmann iteration.
        lattice.collideAndStream();
```

```
117     }
118
119     delete boundaryCondition;
120 }
```

2.5.1 Poiseuille profile

Line 20 This function computes the parabolic Poiseuille profile, in a channel of a given height, and with a given maximum velocity in the middle of the channel. The object `IncomprFlowParam` is discussed below: it stores various parameters of the simulation.

Line 26-47 This is an example for a so-called function object, or functional. It is a class which overloads the function call operator, operator(). Instances of this class behave like normal functions, but the object is more intelligent than a pure function. It can for example accept parameters at the constructor and possess a configurable behavior. The present function object encapsulates the function `poiseuilleVelocity` and is configured with an object of type `IncomprFlowParam`. This is a useful trick to avoid the need for declaring simulation parameters as global variables and making them accessible to everyone, as we did in the code 1.2.

Line 55 Specify that all exterior boundaries of the domain implement a Dirichlet boundary condition for the velocity. At this point, the value of the velocity on the boundaries is yet undefined.

Line 58 Define the value of the velocity on boundary nodes from the analytical Poiseuille profile. Note that although this function is applied to the entire domain, it has an effect only of nodes which have been previously defined as being Dirichlet boundary nodes.

2.5.2 Simulation parameters and boundary condition

Line 86 An object of type `IncomprFlowParam` stores the parameters of the simulation (Reynolds number, lattice resolution, reference velocity in lattice units, etc.), and performs unit conversions. It is for example used to compute automatically the relaxation parameter `omega` from the Reynolds number.

Line 105 Choose the algorithm which is used to implement the boundary condition. The type `LocalBoundaryCondition` implements the regularized boundary condition, which is entirely local (it doesn't need to acces neighbor nodes). The type `InterpBoundaryCondition` uses finite difference schemes to compute the strain rate tensor on walls, and is therefore non-local.

2.6 Tutorial 1.6: 3D Channel flow

To keep everything as simple as possible, all tutorials so far were based on 2D simulations. Turning to 3D is pretty easy, though. Let's reconsider the Poiseuille flow from the previous tutorial and implement in 3D. The resulting code, shown below, can also be found in the file `lesson_1_6.cpp`:

```
1  #include "palabos3D.h"
2  #include "palabos3D.hh"
3  #include <vector>
4  #include <iostream>
5  #include <iomanip>
6
7
8  /* Code 1.6 in the Palabos tutorial
9   */
10
11 using namespace plb;
```

```
using namespace std;

typedef double T;
#define DESCRIPTOR plb::descriptors::D3Q19Descriptor


/// Velocity on the parabolic Poiseuille profile
T poiseuilleVelocity(plint iY, plint iZ, IncomprFlowParam<T> const& parameters) {
    T y = (T)iY / parameters.getResolution();
    T z = (T)iZ / parameters.getResolution();
    return 4.*parameters.getLatticeU() * (y-y*y) * (z-z*z);
}

/// A functional, used to initialize the velocity for the boundary conditions
template<typename T>
class PoiseuilleVelocity {
public:
    PoiseuilleVelocity(IncomprFlowParam<T> parameters_)
        : parameters(parameters_)
    { }
    void operator()(plint iX, plint iY, plint iZ, Array<T,3>& u) const {
        u[0] = poiseuilleVelocity(iY, iZ, parameters);
        u[1] = T();
        u[2] = T();
    }
private:
    IncomprFlowParam<T> parameters;
};

void channelSetup( MultiBlockLattice3D<T,DESCRIPTOR>& lattice,
                   IncomprFlowParam<T> const& parameters,
                   OnLatticeBoundaryCondition3D<T,DESCRIPTOR>& boundaryCondition )
{
    // Create Velocity boundary conditions
    boundaryCondition.setVelocityConditionOnBlockBoundaries(lattice);

    setBoundaryVelocity (
            lattice, lattice.getBoundingBox(),
            PoiseuilleVelocity<T>(parameters) );

    lattice.initialize();
}

void writeGifs(MultiBlockLattice3D<T,DESCRIPTOR>& lattice, plint iter)
{
    const plint imSize = 600;
    const plint nx = lattice.getNx();
    const plint ny = lattice.getNy();
    const plint nz = lattice.getNz();
    Box3D slice(0, nx-1, 0, ny-1, nz/2, nz/2);
    ImageWriter<T> imageWriter("leeloo");
    imageWriter.writeScaledGif(createFileName("u", iter, 6),
                               *computeVelocityNorm(lattice, slice),
                               imSize, imSize );
}

int main(int argc, char* argv[]) {
    plbInit(&argc, &argv);
```

```
70
71      global::directories().setOutputDir("./tmp/");
72
73      // Use the class IncomprFlowParam to convert from
74      //   dimensionless variables to lattice units, in the
75      //   context of incompressible flows.
76      IncomprFlowParam<T> parameters(
77              (T) 1e-2,  // Reference velocity (the maximum velocity
78                         //   in the Poiseuille profile) in lattice units.
79              (T) 100.,  // Reynolds number
80              30,        // Resolution of the reference length (channel height).
81              3.,        // Channel length in dimensionless variables
82              1.,        // Channel height in dimensionless variables
83              1.         // Channel depth in dimensionless variables
84      );
85      const T imSave   = (T)0.02; // Time intervals at which to save GIF
86                                  //   images, in dimensionless time units.
87      const T maxT     = (T)2.5;  // Total simulation time, in dimensionless
88                                  //   time units.
89
90      writeLogFile(parameters, "3D Poiseuille flow");
91
92      MultiBlockLattice3D<T, DESCRIPTOR> lattice (
93               parameters.getNx(), parameters.getNy(), parameters.getNz(),
94               new BGKdynamics<T,DESCRIPTOR>(parameters.getOmega()) );
95
96      OnLatticeBoundaryCondition3D<T,DESCRIPTOR>*
97          //boundaryCondition = createInterpBoundaryCondition3D<T,DESCRIPTOR>();
98          boundaryCondition = createLocalBoundaryCondition3D<T,DESCRIPTOR>();
99
100     channelSetup(lattice, parameters, *boundaryCondition);
101
102     // Main loop over time iterations.
103     for (plint iT=0; iT*parameters.getDeltaT()<maxT; ++iT) {
104         if (iT%parameters.nStep(imSave)==0) {
105             pcout << "Saving Gif at time step " << iT << endl;
106             writeGifs(lattice, iT);
107         }
108         // Execute lattice Boltzmann iteration.
109         lattice.collideAndStream();
110     }
111
112     delete boundaryCondition;
113 }
```

The few lines which have changed from the 2D to the 3D code are the following:

Line 1 and 3 The header files `palabos2D.h` and `palabos2D.hh` are replaced by their 3D counterpart.

Line 16 On this line, the type of lattice is chosen. While in 2D the only available nearest-neighbor lattice is D2Q9 (D2Q7 is currently not supported), there are 3 choices in 3D: D3Q15, D3Q19, and D3Q27.

Line 19 To set up a Dirichlet condition for the inlet and the outlet, an analytic velocity profile is proposed. There exists an analytical solution for 3D channel flow, but we do not use it here to keep the code simple. Instead, the velocity profile is formulated as a tensor product of the 2D Poiseuille flow, once in each of the two axes parallel to the inlet and outlet.

Line 26-39 The function object which is used to instantiate the boundary conditions has the same form in 3D as in 2D. This time, the function call operator takes three arguments, the three space dimensions, and returns a 3D

velocity vector.

Line 41 Initialization of the geometry is exactly identical as in the previous 2D Poiseuille flow, except that the extension 2D is replaced by 3D in all keywords.

Line 55 It remains useful to write regularly GIF snapshots of the flow to monitor the flow evolution in a simple way. A 2D slice needs to be extraced from the full domain to produce an image.

Line 76 It is possible to provide the class IncomprFlowParam with an additional argument to indicate the extent of the domain in z-direction in the 3D case.

Line 92 The central object of the simulation, the `MultiBlockLattice2D`, is replaced by a `MultiBlockLattice3D`.

2.7 Tutorial 1.7: Post-processing with Paraview

In previous tutorials, two approaches to analyzing the computed numerical data have been used. The first consists of using the class `ImageWriter` to produce GIF snapshots of the velocity norm, or other scalar flow variables, during the simulation. This approach is particularly useful for obtaining a qualitative impression of the evolution of the flow, and for being able to interrupt the simulation when errors appear. Another, more quantitative approach, consists of writing the data into a file, using a plain ASCII format. In this case, the data stream is linearized, which means that structural information, such as the size of the domain in each space dimension, is lost. In the following code, based on the previous tutorial, the velocity is for example computed on a slice perpendicular to the z-axis, and written into an ASCII file:

```
// Attribute a value to nx, ny, and nz.
Box3D slice(0, nx-1, 0, ny-1, nz/2, nz/2);
plb_ofstream ofile("slice.dat");
ofile << setprecision(4) << *computeVelocityNorm(lattice, slice) << endl;
```

As you can see, the C++ stream operators have been overloaded for this task. This means that the output can be formated using I/O manipulators like `setprecision(n)` to chose the numerical precision, `setw(n)` to format each number in a cell of fixed width, or `fixed` respectively `scientific` to chose the representation of floating point variables. The flow field stored in the file `slice.dat` can for example be analyzed with help of the open-source program Octave. For this, you need to know the values of `nx`, `ny` and `nz`, because they are not stored in the file:

```
% Matlab/Octave script
% Assign a value to nx and ny
load slice.dat
slice = reshape(slice, nx, ny);
imagesc(slice)
```

In the present tutorial, an additional approach is shown in which the data is written into a file in a so-called `VTK` format, after which it can be analyzed with a general post-processing tool. The following function is used in the tutorial code `tutorial_1_7` to write VTK data of the 3D Poiseuille flow presented in tutorial 1.6:

```
void writeVTK(MultiBlockLattice3D<T,DESCRIPTOR>& lattice,
              IncomprFlowParam<T> const& parameters, plint iter)
{
    T dx = parameters.getDeltaX();
    T dt = parameters.getDeltaT();
    VtkImageOutput3D<T> vtkOut(createFileName("vtk", iter, 6), dx);
    vtkOut.writeData<float>(*computeDensity(lattice), "density", 1.);
    vtkOut.writeData<3,float>(*computeVelocity(lattice), "velocity", dx/dt);
}
```

The data in the VTK file is represented in dimensionless variables. For this, the variable `dx` is given to the constructor of the `VtkImageOutput3D` object in order to indicate the size of a cell spacing. Furthermore, every variable which is written needs to be rescaled according to its units, namely `1.` for the density and `dx/dt` for the velocity. A few comments on the VTK output are in order:

- The VTK files can hold an arbitrary number of fields, which are written one after another with the command `writeData`. In case of Vector fields, the number of components of the vector needs to be indicated as a template argument.
- The data is written in binary format, using the binary-encoded ASCII format Base64 for compatibility. Therefore, unlike the plain ASCII output used previously, the VTK format does not loose any digit of precision.
- In the above example, the data is converted from `double` to `float` to save space on the disk.

The open-source program `Paraview` provides a convenient interactive interface for analyzing the data. To end this tutorial, you are encouraged to run the program `tutorial_1_7`, install `Paraview` on your system and analyze the output. Among others, `Paraview` can easily produce an animation from the written image series.

CHAPTER

THREE

TUTORIAL 2: UNDERSTANDING THE MULTI-BLOCK STRUCTURE

The code structure of Palabos programs is driven by the duality between atomic-blocks, which represent regular data-arrays, and complex multi-blocks. Thanks to a practically identical interface, they appear to the user on a seemingly equal footing. In reality, however, there exists a hierarchical relationship bewteen them. The multi-block is actually a composite construct of several adjacent blocks and uses itself atomic-blocks for its internal implementation. Atomic- and multi-blocks come in three flavors: the (multi-) block-lattice which holds lattice Boltzmann variables, the (multi-) scalar-field for scalar data-arrays, and the (multi-) tensor-field for vector- or tensor-valued data-arrays.

Tutorial 2.1 exploits the similarity of interface between atomic-blocks and multi-blocks, and rewrites a program from Tutorial 1, replacing the multi block-lattice by a plain block-lattice. Tutorial 2.2 provides further insight in the multi-block and gives the user the possibility to construct such an object manually by explicitly specifying the position of the consisting atomic-blocks. In tutorial 2.3, a multi-block (or for all means, an atomic-block) is manipulated through the explicit use of so-called data processors. They provide explicit access to atomic-block cells inside a multi-block, even if the internal structure of the multi-block is unknown to the user. Tutorial 2.4 finally explains how a program is parallelized (again manually for didactic purposes) by attributing the components of a multi-block to different threads.

Warning: The purpose of Tutorial 2 is to provide deeper insight into the mechanisms of Palabos, and not to establish good coding practice. As already mentioned, you should in practice always prefer multi-blocks over atomic-blocks at the end-user level, in spite of the counter-example shown in Tutorial 2.1.

3.1 Tutorial 2.1: Formulating a program with an atomic-block

In *Tutorial 1.5: Boundary conditions*, a simulation for a 2D Poiseuille flow was presented, which used the data structure `MultiBlock2D` to hold the data. This is the right way of doing, because it is recommended to use multi-block structures for practically all purposes in end-user programs. Strictly speaking, a non-parallel version of this program could however also be coded using a simpler data structure, because of the simple, rectangular shape of the domain. Such a conversion is straightforward in Palabos, because most of the code is generic and works identically for atomic-blocks and multi-blocks. As it can be seen on the following code, also found in the file `tutorial_2_1.cpp`, it is sufficient to replace the keyword `MultiBlockLattice2D` by `BlockLattice2D` at two places:

```
1  /* Code 2.1 in the Palabos tutorial
2   */
3
4  // ... Definition of the Poiseuille profile, and instantiation of the geometry are
5  //     exactly identical with a multi-block or an atomic-block
6
7
8  // Multi-Block version of tutorial 1.5:
9  //   void writeGifs(MultiBlockLattice2D<T,DESCRIPTOR>& lattice, plint iter)
10 //   { ...
```

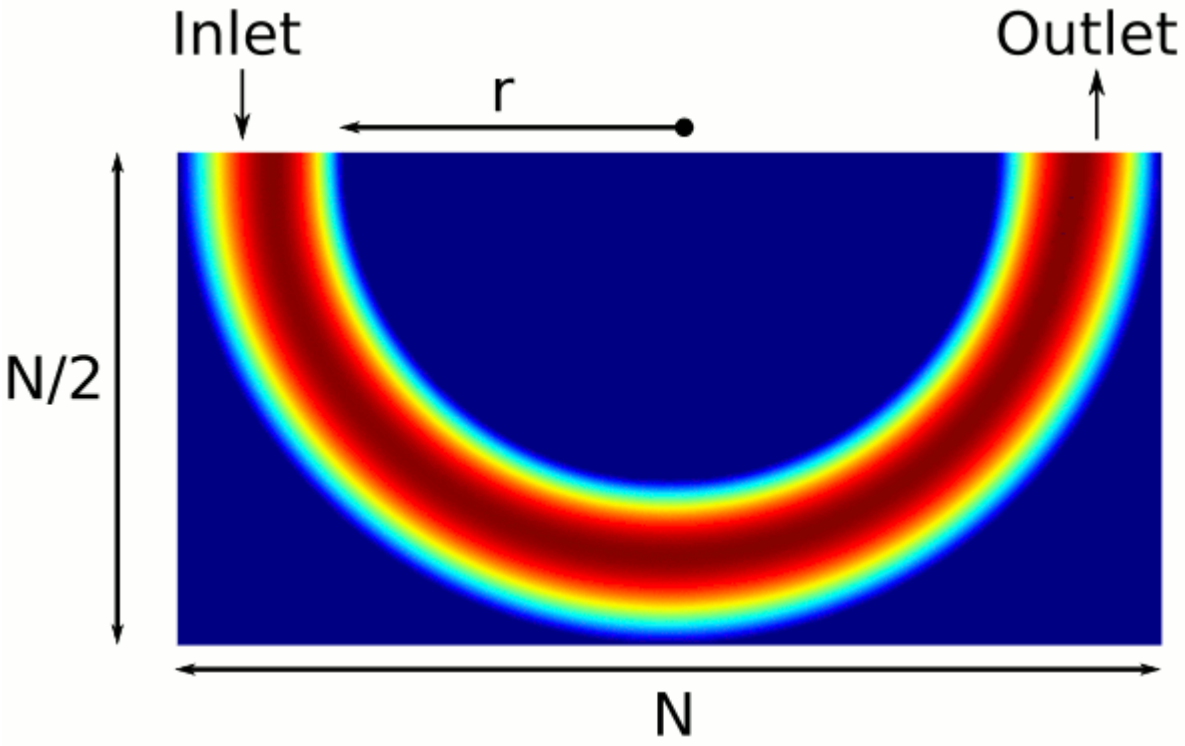
```
11
12 // Atomic-Block version of tutorial 2.1:
13 void writeGifs(BlockLattice2D<T,DESCRIPTOR>& lattice, plint iter)
14 {
15 // ...
16
17 int main(int argc, char* argv[])
18 {
19     // ... Definition of the parameters are identical in both versions
20
21     // Multi-Block version of tutorial 1.5:
22     // MultiBlockLattice2D<T, DESCRIPTOR> lattice (
23     //           parameters.getNx(), parameters.getNy(),
24     //           new BGKdynamics<T,DESCRIPTOR>(parameters.getOmega()) );
25
26     // Atomic-Block version of tutorial 2.1:
27     BlockLattice2D<T, DESCRIPTOR> lattice (
28              parameters.getNx(), parameters.getNy(),
29              new BGKdynamics<T,DESCRIPTOR>(parameters.getOmega()) );
30
31     // ... Creating the initial condition and running the simulation is
32     //      identical in both versions
33 }
```

Although very little has changed on the surface, the algorithm instantiated in the above code is much simpler than in the multi-block case. You can view the BlockLattice2D as a simple nx-by-ny-by-9 value array, just as you would use it in a straightforward example lattice Boltzmann code.

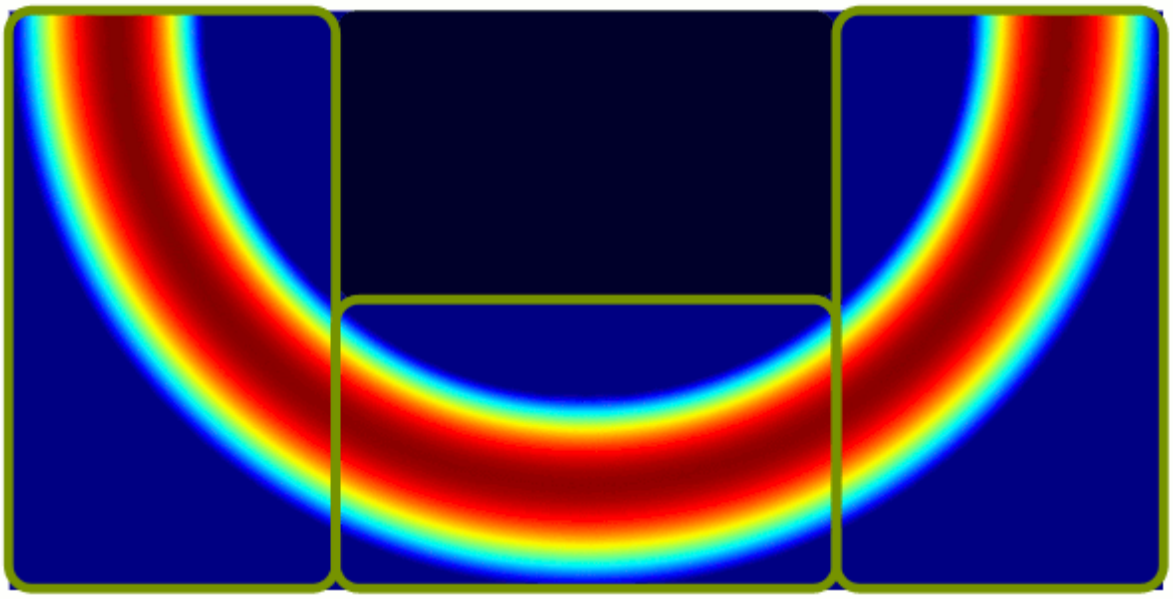
As you progress with your work with Palabos, you will learn to appreciate the abstraction mechanism through which the complex multi-block behaves like a regular construct. In practical work, it offers a relatively simple way to cope with complicated constructs.

3.2 Tutorial 2.2: Creating a multi-block structure manually

One of the uses of the multi-block consists in the memory-efficient implementation of sparse domains. To understand this, consi



The fluid is confined in a channel shaped like a half-circle. Pressure boundary conditions are used with an appropriate differenc



The construction of this sparse domain is done in the code `tutorial_2_2.cpp`, it is integrally printed here (without the usual header lines):

```
1 /// Describe the geometry of the half-circular channel, used in tutorial 2.
2 template<typename T>
```

```
class BounceBackNodes : public DomainFunctional2D {
public:
    BounceBackNodes(plint N, plint radius)
        : cx(N/2),
          cy(N/2),
          innerR(radius),
          outerR(N/2)
    { }
    /// Return true for all cells outside the channel, on which bounce-back
    ///  dynamics must be instantiated.
    virtual bool operator() (plint iX, plint iY) const {
        T rSqr = util::sqr(iX-cx) + util::sqr(iY-cy);
        return rSqr <= innerR*innerR || rSqr >= outerR*outerR;
    }
    virtual BounceBackNodes<T>* clone() const {
        return new BounceBackNodes<T>(*this);
    }
private:
    plint cx;      //< X-position of the center of the half-circle.
    plint cy;      //< Y-position of the center of the half-circle.
    plint innerR;  //< Outer radius of the half-circle.
    plint outerR;  //< Inner radius of the half-circle.
};


void halfCircleSetup (
        MultiBlockLattice2D<T,DESCRIPTOR>& lattice, plint N, plint radius,
        OnLatticeBoundaryCondition2D<T,DESCRIPTOR>& boundaryCondition )
{
    // The channel is pressure-driven, with a difference deltaRho
    //   between inlet and outlet.
    T deltaRho = 1.e-2;
    T rhoIn  = 1. + deltaRho/2.;
    T rhoOut = 1. - deltaRho/2.;

    Box2D inlet (0,     N/2, N/2, N/2);
    Box2D outlet(N/2+1, N,   N/2, N/2);

    boundaryCondition.addPressureBoundary1P(inlet, lattice);
    boundaryCondition.addPressureBoundary1P(outlet, lattice);

    // Specify the inlet and outlet density.
    setBoundaryDensity (lattice, inlet, rhoIn);
    setBoundaryDensity (lattice, outlet, rhoOut);

    // Create the initial condition.
    Array<T,2> zeroVelocity(0.,0.);
    T constantDensity = (T)1;
    initializeAtEquilibrium (
       lattice, lattice.getBoundingBox(), constantDensity, zeroVelocity );

    defineDynamics(lattice, lattice.getBoundingBox(),
                   new BounceBackNodes<T>(N, radius),
                   new BounceBack<T,DESCRIPTOR>);

    lattice.initialize();
}

```

```
61 void writeGifs(MultiBlockLattice2D<T,DESCRIPTOR>& lattice, plint iter)
62 {
63     const plint imSize = 600;
64     ImageWriter<T> imageWriter("leeloo");
65     imageWriter.writeScaledGif(createFileName("u", iter, 6),
66                                *computeVelocityNorm(lattice),
67                                imSize, imSize );
68 }
69
70 int main(int argc, char* argv[]) {
71     plbInit(&argc, &argv);
72
73     global::directories().setOutputDir("./tmp/");
74
75     // Parameters of the simulation
76     plint N         = 400;    // Use a 400x200 domain.
77     plint maxT      = 20001;
78     plint imageIter = 1000;
79     T omega         = 1.;
80     plint radius    = N/3;    // Inner radius of the half-circle.
81
82     // Parameters for the creation of the multi-block.
83
84     // d is the width of the block which is exempted from the full domain.
85     plint d = (plint) (2.*std::sqrt(util::sqr(radius)-util::sqr(N/4.)));
86     plint x0 = (N-d)/2 + 1;  // Begin of the exempted block.
87     plint x1 = (N+d)/2 - 1;  // End of the exempted block.
88
89     // Create a block distribution with the three added blocks.
90     plint envelopeWidth = 1;
91     SparseBlockStructure2D sparseBlock(N+1, N/2+1);
92     sparseBlock.addBlock(Box2D(0, x0,      0, N/2),   sparseBlock.
   ↪nextIncrementalId());
93     sparseBlock.addBlock(Box2D(x0+1, x1-1, 0, N/4+1), sparseBlock.
   ↪nextIncrementalId());
94     sparseBlock.addBlock(Box2D(x1, N,      0, N/2),   sparseBlock.
   ↪nextIncrementalId());
95
96     // Instantiate the multi-block, based on the created block distribution and
97     // on default parameters.
98     MultiBlockLattice2D<T, DESCRIPTOR> lattice (
99         MultiBlockManagement2D(
100             sparseBlock,
101             defaultMultiBlockPolicy2D().getThreadAttribution(), envelopeWidth ),
102         defaultMultiBlockPolicy2D().getBlockCommunicator(),
103         defaultMultiBlockPolicy2D().getCombinedStatistics(),
104         defaultMultiBlockPolicy2D().getMultiCellAccess<T,DESCRIPTOR>(),
105         new BGKdynamics<T,DESCRIPTOR>(omega)
106     );
107
108     pcout << getMultiBlockInfo(lattice) << std::endl;
109
110     OnLatticeBoundaryCondition2D<T,DESCRIPTOR>*
111         boundaryCondition = createLocalBoundaryCondition2D<T,DESCRIPTOR>();
112
113     halfCircleSetup(lattice, N, radius, *boundaryCondition);
114
115     // Main loop over time iterations.
```

```
116     for (plint iT=0; iT<maxT; ++iT) {
117         if (iT%imageIter==0) {
118             pcout << "Saving Gif at time step " << iT << endl;
119             writeGifs(lattice, iT);
120         }
121         lattice.collideAndStream();
122     }
123
124     delete boundaryCondition;
125 }
```

Among other information, the program prints the following lines to the screen, as a result of the instruction on line 108:

```
Size of the multi-block:      401-by-201
Number of atomic-blocks:      3
Smallest atomic-block:        172-by-102
Largest atomic-block:         115-by-201
Number of allocated cells:    0.063573 million
Fraction of allocated cells: 78.8737 percent
```

This behavior, as well as other parts of the code, are commented in the following:

Line 3 The half-circle shape of the simulation is described by the function class `BounceBackNodes`. Later in the program, this object is use to define all cells outside the cell as bounce-back nodes, and inside the channel as BGK dynamics nodes. The mechanisms involved here are further investigated in Tutorial 2.3 (technically, we will refer to class `BounceBackNodes` as a data-processor), while the present tutorial concentrates on sparse domain implementations.

Line 85-87 In this part, the coordinates of the three blocks which cover the channel are computed.

Line 91 A `SparseBlockStructure2D` describes the arrangement of atomic-blocks inside a multi-block. Three blocks are added. Each of them gets an integer ID, which in this case is incremental (number 0, 1, an 2). As explained in *Tutorial 2.4: Generate the sparse-block structure automatically*, this ID is used in parallel programs to associate a block to a MPI process.

Line 98 Instead of the usual default constructor of the `MultiBlockLattice2D`, a more detailed version is used which offers the possibility to control various aspects of the multi-block instantiation, and in particular to specify the arrangement of the internal atomic-blocks. The other arguments of this constructors are not discussed here. As it is shown in the code, one can always refer to the object returned by `defaultMultiBlockPolicy2D()` to use a default choice for these parameters. The argument `envelopeWidth=1` hints at the fact that the dynamics executed on this lattice uses a nearest-neighbor communication pattern, and an overlap of one-cell width between atomic-blocks is needed to express the streaming step inside an atomic-block consistently.

Line 108 The `getMultiBlockInfo` function provides some insight in the internal structure of a multi-block. You learn for example that the block added in the middle has dimensions 172-by-102, while the two later blocks have size 115-by-201. Switching from a regular to a sparse block structure obviously provided a gain of 21% in memory usage.

3.3 Tutorial 2.3: Understanding data processors

Both atomic-blocks and multi-blocks are most often manipulated with help of constructs known under the name of "data processor" in Palabos. A data processor specifies an operation to be executed on each cell of a chosen domain on the block. In case a multi-block is used, the desired domain is intersected with the region occupied by the internal atomic-blocks, and then executed on the area of intersection of each atomic-block. Data processors are often used

indirectly, through higher-level user functions. In the code of Tutorial 2.2 for example, bounce-back nodes are defined by calling the function `defineDynamics`. This function in its turn instantiates a data processor, responsible for re-defining the collision step on all the chosen nodes. Other types of data processors are used for example to attribute initial values to the simulation, define boundary conditions, or post-process data. The user functions offered in Palabos for all these operations are listed in the appendix of the user guide. It should also be mentioned that data processors can act simultaneously on several blocks, and in this way create a coupling between two block-lattices, or between a block-lattice and a scalar-field or tensor-field. This behavior is for example used to compute the velocity in a block-lattice and store the result in a three-component tensor-field, or to create the coupling between two lattices for the implementation of multi-component fluids.

In order to obtain a better understanding of how such a data processor works, the call to the function `defineDynamics` in the previous tutorial is now replaced successively by two alternative, explicit ways of attributing a BounceBack dynamics to the chosen wall cells. All discussed code constructs can also be found in the file `tutorial_2_3.cpp`.

The first approach is easiest to understand, because it is entirely manual. A loop over all space directions is written manually and, with help of the function class `BounceBackNodes` from the previous tutorial, it is decided to re-define the dynamics of chosen nodes through the interface of the multi-block:

```
/// Manual instantiation of the bounce-back nodes for the boundary.
/** Attention: This is NOT the right way of doing, because it is slow and
 *  non-parallelizable. Instead, use a data-processor.
 */
void createHalfCircleManually (
        MultiBlockLattice2D<T,DESCRIPTOR>& lattice, plint N, plint radius )
{
    BounceBackNodes<T> domain(N,radius);
    for (plint iX=0; iX<=N; ++iX) {
        for (plint iY=0; iY<=N/2; ++iY) {
            if (domain(iX,iY)) {
                defineDynamics(lattice, iX, iY, new BounceBack<T,DESCRIPTOR>);
            }
        }
    }
}
```

This approach is really self-explaining. It is however crucial to understand that such an approach should *never* be chosen in practice, for efficiency considerations. This approach is slow, because for each access to a cell through the function `defineDynamics`, the atomic-block on which the cell is located must first be determined. Even worse yet, this way of accessing a multi-block is not properly parallelized (the result of the operation is correct in parallel, but the speed of the operation is equal or even inferior to the serial execution speed). As it is shown in the next tutorial, parallelism is implemented in Palabos by assigning each atomic-block inside a multi-block to a given processor (CPU). During operations that are executed on an extended domain, such as, the definition of the collision rule to a given part of the simulation, it is expected that each processor works only with the atomic-block assigned to it. Writing out explicitly a loop over space directions however means to assign work for the whole domain to all processors. This can impact the parallel performance of a program significantly, even if it is only part of the setup of a simulation. In programs which are massively parallelized on hundreds or thousands of processors/cores, a badly parallelized initial stage can represent a bottleneck for the whole simulation. One of the most important rules in Palabos is therefore to *never write loops over space dimensions in end-user applications*. Instead, space loops are only written inside data processors, as shown below.

Here's how you can manually write a data processor which creates the half-circle domain, without resorting to the helper function `defineDynamics` (the function `defineDynamics` is of course nothing else than a wrapper which instantiates a data processor for you):

```
1 /// This functional defines a data processor for the instantiation
2 ///   of bounce-back nodes following the half-circle geometry.
```

```
3  template<typename T, template<typename U> class Descriptor>
4  class HalfCircleInstantiateFunctional
5      : public BoxProcessingFunctional2D_L<T,Descriptor>
6  {
7  public:
8      HalfCircleInstantiateFunctional(plint N_, plint radius_)
9          : N(N_), radius(radius_)
10     { }
11     virtual void process(Box2D domain, BlockLattice2D<T,Descriptor>& lattice) {
12         BounceBackNodes<T> bbDomain(N,radius);
13         Dot2D relativeOffset = lattice.getLocation();
14         for (plint iX=domain.x0; iX<=domain.x1; ++iX) {
15             for (plint iY=domain.y0; iY<=domain.y1; ++iY) {
16                 if (bbDomain(iX+relativeOffset.x,iY+relativeOffset.y)) {
17                     lattice.attributeDynamics (
18                             iX,iY, new BounceBack<T,DESCRIPTOR> );
19                 }
20             }
21         }
22     }
23     virtual void getTypeOfModification(std::vector<modif::ModifT>& modified) const {
24         modified[0] = modif::dynamicVariables;
25     }
26     virtual HalfCircleInstantiateFunctional<T,Descriptor>* clone() const
27     {
28         return new HalfCircleInstantiateFunctional<T,Descriptor>(*this);
29     }
30 private:
31     plint N;
32     plint radius;
33 };
34
35 /// Automatic instantiation of the bounce-back nodes for the boundary,
36 ///   using a data processor.
37 void createHalfCircleFromDataProcessor (
38         MultiBlockLattice2D<T,DESCRIPTOR>& lattice, plint N, plint radius )
39 {
40     applyProcessingFunctional (
41             new HalfCircleInstantiateFunctional<T,DESCRIPTOR>(N,radius),
42             lattice.getBoundingBox(), lattice );
43 }
```

Line 4 The class `HalfCircleInstantiateFunctional` represents the data processor for the instantiation of the half-circle geometry. It inherits from `BoxProcessingFunctional2D_L`, where the `L` stands for "lattice", in reference to the fact that the processor acts on a lattice. The two other possibilities are `S` for scalar-field, and `T` for tensor-field. Similarly, a coupling between a block-lattice and a scalar-field is obtained by inheriting from `BoxProcessingFunctional2D_LS`.

Line 11 The method `process` defines the heart of the data processor. It is invoked repeatedly on all atomic blocks, after intersection of the original domain with the domain of the individual blocks: Palabos computes the subdivision, and you write the code to be executed on the sub-domains. The parameters delivered to the method `process` are an atomic-block and the domain on which the operation is executed, and it is your responsibility to write a loop over all cells of this domain. It should be pointed out that Palabos could have had a simpler interface, in which the data processor acts on a single cell, and the space loop is written somewhere in Palabos' library code. The present choice is however preferrable for efficiency reasons, because it avoids the costs of a function call on each treated cell. While such a function call is acceptable for computationally intensive operations (such as, computing the collision step for a cell), in can have an important relative performance

impact on lightweight data processors which execute, say, a simple arithmetic operation. As it is written currently, the overhead due to the data processor's interface is practically invisible, and the data is processed with the same efficiency as manually programmed, raw data constructs. When this raw efficiency is not required (for example during an initialization step which takes little time), it is possible to use more elegant interfaces to the data-processor which free you from the burden of writing loops manually. For more information, refer to the constructs `OneCellFunctionalXD` and `OneCellIndexedFunctionalXD` in the user guide (appendix-functions). An example usage of class `OneCellIndexedFunctional2D` is presented in the program `multiComponent2d/rayleighTaylor.cpp`.

Line 13 The domain delivered by parameter to the method `process` uses coordinates relative to the atomic-block. They are not compatible with the global coordinate system of the multi-block, and can therefore not be used directly to decide whether a given cell is part of the fluid channel or not. First, they need to be converted to global coordinates by accessing the relative position of the atomic-block within the multi-block, through the command `lattice.getLocation()`. Again, this coordinate transformation is done manually here for efficiency reasons; you can also chose to use a slightly less efficient, but more elegant automatic mechanism by using the construct `OneCellIndexedFunctionalXD` (see appendix-functions in the user guide).

Line 23-25 Here, you inform Palabos what type of modification your data processor has performed on the block-lattice. This information is needed internally for Palabos to set up its parallel communication pattern, and to communicate only data that has been modified. Use the identifier `modif::nothing` to say that the data processor performed read-only accesses, `modif::staticVariables` if it modified the populations (or external scalars), `modif::dynamicVariables` if it changed the content of the dynamics objects (for example, when it locally changed the relaxation parameter), `modif::allVariables` when it made changes of both kinds, and, finally, `modif::dataStructure` when it assigned a new dynamics object to the cells (as it did in the present case).

Line 26-29 The definition of the method `clone` is paradigmatic in Palabos: it is required for Palabos' internal memory management. All classes that are placed in a Palabos inheritance hierarchy have such a method, and *the definition of this method always has the exact same shape*. You'll get used to writing this method without even thinking about it.

Line 40 Through a function call to `applyProcessingFunctional`, a data processor is executed exactly once on a multi-block or an atomic-block. Alternatively, the function `integrateProcessingFunctional` is used to *add* a data processor to a block, and have it executed periodically at each time iteration cycle. The latter approach is chosen for data processors which are part of the dynamics of the simulation. Examples are non-local boundary conditions, or couplings between lattices in multi-component fluids.

3.4 Tutorial 2.4: Generate the sparse-block structure automatically

As previously pointed out, you will practically never want to compute the internal structure of a sparse multi-block manually, and instead use one of Palabos' automatic domain generators. For example, the Palabos application in the directory `examples/showCases/aneurysm` shows how you can automatically read a geometry description from an STL file, "voxelize" the domain (decide which cells are part of the fluid), and then generate the sparse structure. By the way, Palabos' automatic domain generation algorithms go even a step further than what is done in the present tutorial, as they also instantiate sophisticated algorithms for curved walls, in which the wall location is represented with second-order accuracy (while in this tutorial the wall is represented through a stair-case shape).

Anyway, let's get back to tutorial 2.2, and modify the code in such a way that the sparse block structure is generated automatically by Palabos:

```
1 ...
2
3 template<typename T>
4 class FluidNodes {
5 public:
```

```
6       FluidNodes(plint N_, plint radius_) : N(N_), radius(radius_)
7       { }
8       bool operator() (plint iX, plint iY) const {
9           return ! BounceBackNodes<T>(N,radius)(iX,iY);
10      }
11  private:
12      plint N, radius;
13  };
14
15  ...
16
17  MultiScalarField2D<int> flagMatrix(N+1, N/2+1);
18  setToFunction(flagMatrix, flagMatrix.getBoundingBox(), FluidNodes<T>(N, radius));
19  plint blockSize = 15;
20  plint envelopeWidth = 1;
21  MultiBlockManagement2D sparseBlockManagement =
22      computeSparseManagement (
23              *plb::reparallelize(flagMatrix, blockSize,blockSize),
24              envelopeWidth );
25
26  // Instantiate the multi-block, based on the created block distribution and
27  // on default parameters.
28  MultiBlockLattice2D<T, DESCRIPTOR> lattice (
29      sparseBlockManagement,
30      defaultMultiBlockPolicy2D().getBlockCommunicator(),
31      defaultMultiBlockPolicy2D().getCombinedStatistics(),
32      defaultMultiBlockPolicy2D().getMultiCellAccess<T,DESCRIPTOR>(),
33      new BGKdynamics<T,DESCRIPTOR>(omega)
34  );
```

Line 3-13 In order to create the sparse block structure, Palabos needs to know which lattice cells are part of the fluid. A specific functional is implemented here which returns this information, which is, the negation of `BounceBackNodes`.

Line 17 and 18 An integer-valued flag matrix is constructed which is zero valued on the bounce-back domain, and non-zero on fluid nodes.

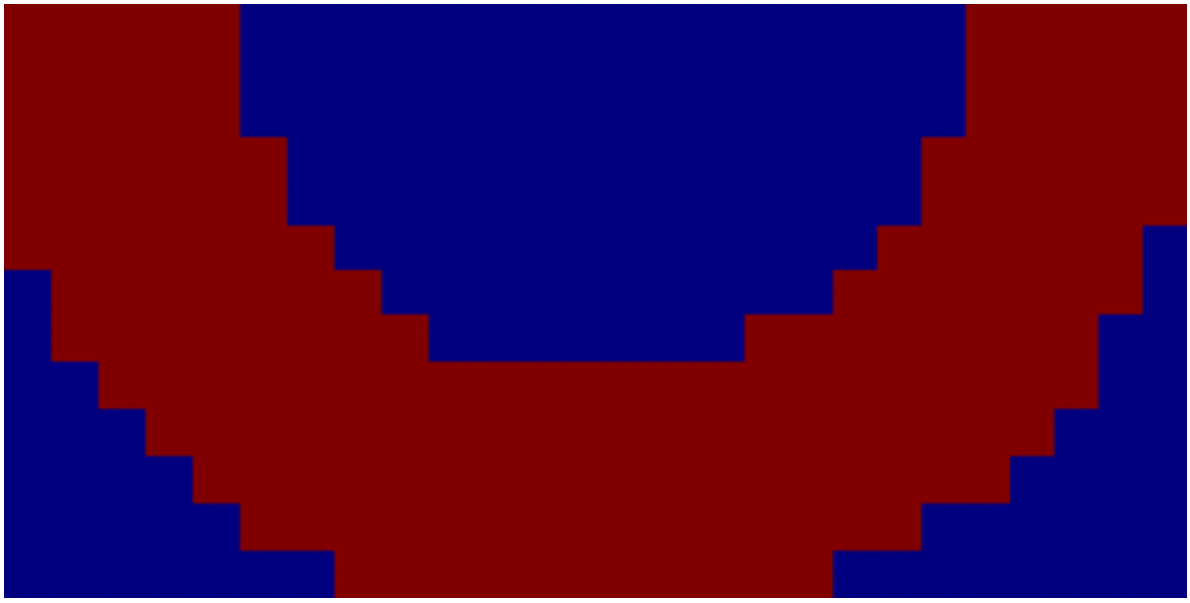
Line 19 Let's cover the domain with blocks of size (approximately) 15x15. The smaller these blocks, and the more memory efficient a simulation you get. If the blocks are chosen too small, you loose however efficiency, as the overhead of communicating between blocks is too large. The optimal atomic block size depends on your hardware and must be determined experimentally.

Line 23 To start with, the multi-block is restructured to be covered by blocks of size 15x15.

Line 22 As a next step, all blocks that have no fluid cell are removed. The result is saved in a data structure of type `MultiBlockManagement`

Line 28 The created multi-block management structure is now used to create a block-lattice with the sparse block-structure.

As it can be seen from the program's output, this code is more efficient than the one in tutorial 2.2, as it removes 45% of the cell



3.5 Tutorial 2.5: Parallelism in a manually created multi-block

If you use multi-blocks as your basic data type, if you construct them in a standard way, as in all examples of Tutorial 1, and if you compile the program with the `MPI_PARALLEL` flag, then the program is automatically parallelized. This means that the multi-block is automatically subdivided into components which are associated to the different processors.

In the previous tutorials, the multi-block structure was created manually, which means that parallelism must be handled manually too. Let's take for example the code from *Tutorial 2.2: Creating a multi-block structure manually*, edit the Makefile, set `MPI_PARALLEL=true`, and recompile. Because the domain is manually created with three blocks in this example, it can only be parallelized by means of exactly three processes: `mpirun -np 3 ./tutorial_2_2`. The algorithm which associates blocks to MPI processes is provided as an argument to the constructor of the multi-block, in the listing of *Tutorial 2.2: Creating a multi-block structure manually* above, at line 101, an argument `defaultMultiBlockPolicy2D().getThreadAttribution()`, which imposes a default parallelization strategy: the block with ID 0 go to the MPI thread with ID 0, block #1 goes to MPI thread #1, and so on. The block-to-thread attribution can however also be directed manually, by creating an object of type `ExplicitThreadAttribution`, configuring it, and providing it to the constructor of the multi-block (see file `multiBlock/threadAttribution.h` for a definition of the class `ExplicitThreadAttribution`).

We leave it as an exercice to modify the code of tutorial 2.2 in such a way that it can be executed on exactly two MPI processes, by assigning two blocks to the first process, and the third block to the second process.

Tutorials from classes and conferences:

- Palabos-Python tutorial at DSFD 2010 (PDF)

User-submitted tutorials:

CHAPTER

FOUR

GEOPHYSICS: COMPUTE THE PERMEABILITY OF A 3D POROUS MEDIUM

Main author: Wim Degruyter

This tutorial illustrates, step-by-step, how to compute the permeability of a given porous media. This process consists of three major steps:

1. Read the geometry, defined by a stack of binary (black and white) images. For this tutorial, we use an artificial geometry, consisting of two hemispheres which partially overlap. This represents a simple media with just a single tube through which the fluid flows from the inlet to the outlet. It is however easily replaced by any complex media, by substituting the bitmap images with user-supplied data. A typical example would be a porous media, with a structure obtained from experimental data.
2. Simulate a stationary (time-independent), pressure-driven flow through this media, by imposing a constant pressure at the inlet, and a constant, lower pressure at the outlet. All physical quantities (velocity, pressure, and viscosity) are, for convenience, expressed in a system of lattice units. The final quantity of interest, the permeability, has dimensions of length squared. Therefore, the actual permeability is the lattice permeability times the spatial resolution squared. If you would like to convert other physical quantities into different units, you should read the tutorial on unit conversion.
3. Compute the permeability which, according to Darcy's law, is proportional to the ratio between the flow rate through the media and the applied pressure gradient. To be able to apply Darcy's law one has to make sure the flow is laminar. Therefore it is recommended to run each simulation several times at different pressure differences. The permeability should stay a constant.

The code developed in this tutorial can be found in the directory `palabos/examples/tutorial/permeability`. This Palabos code has been developed as part of a permeability study of vesicular volcanic rocks[1].

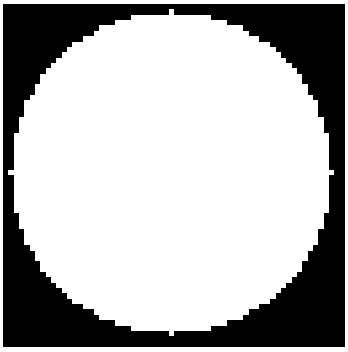
4.1 Pre-processing

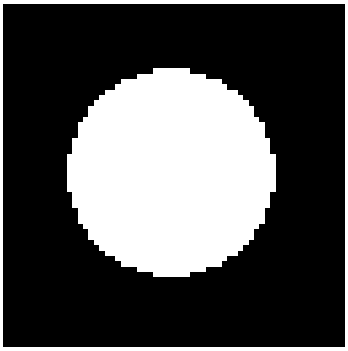
4.1.1 Requirements

We assume you already have downloaded and installed Palabos. Installation support is provided in the user's guide. We also recommend to go through the other Palabos tutorials to get acquainted with compiling and running of Palabos programs.

The input required for `permeability` are a series of black and white images in .bmp format, with filename `namexxxx.bmp` with xxxx numbers starting from 0001. In the directory `twoSpheres` one can find a series of bmp images defining two connected hemispheres. To inspect the stack of images one can use e.g. ImageJ. Here are two example slices, inlet (left) and halfway (right):

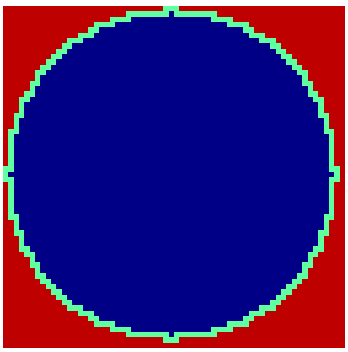
[1] Degruyter W., Bachmann O., Burgisser A., Malaspinas O. A synchrotron perspective on gas flow through pumices. *submitted to Geosphere*

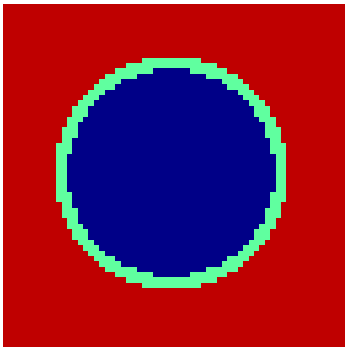




4.1.2 Convert image stack to .dat file

The first step is to create an input file from the images, so the code is able to read in the geometry. This conversion is done by a Matlab script called `createDAT.m`. Open Matlab and go to the directory of `createDAT.m`. At the prompt type `createDAT(number-of-files, path/to/inputprefix, path/to/output.dat)` e.g. try for the twoSpheres files and type `createDAT(48, 'twoSpheres/', 'twoSpheres', 'twoSpheres.dat')`. Every voxel is given a value: 0 for a fluid voxel (blue), 1 for a material voxel that touches (26-connected) a pore voxel (green), and 2 for an interior material voxel (red) illustrated again by the inlet slice (left) and the halfway slice (right):





The Matlab script visualizes the conversion for illustration purposes. If you are converting a large number of files this process can take some time and it is better to switch of the display by commenting out lines 113-115, 258-260, and 372-374.

Once the .dat file is created we can start a simulation.

4.2 Simulation

4.2.1 Brief outline of the code

The `permeability.cpp` code is listed below:

```
#include "palabos3D.h"
#include "palabos3D.hh"

#include <vector>
#include <cmath>
#include <cstdlib>

using namespace plb;

typedef double T;
#define DESCRIPTOR descriptors::D3Q19Descriptor

// This function object returns a zero velocity, and a pressure which decreases
//   linearly in x-direction. It is used to initialize the particle populations.
class PressureGradient {
public:
    PressureGradient(T deltaP_, plint nx_) : deltaP(deltaP_), nx(nx_)
    { }
    void operator() (plint iX, plint iY, plint iZ, T& density, Array<T,3>& velocity)
 ↪const
    {
        velocity.resetToZero();
        density = (T)1 - deltaP*DESCRIPTOR<T>::invCs2 / (T)(nx-1) * (T)iX;

    }
private:
    T deltaP;
    plint nx;
};

void readGeometry(std::string fNameIn, std::string fNameOut, MultiScalarField3D<int>&
 ↪geometry)
```

```
{
    const plint nx = geometry.getNx();
    const plint ny = geometry.getNy();
    const plint nz = geometry.getNz();

    Box3D sliceBox(0,0, 0,ny-1, 0,nz-1);
    std::unique_ptr<MultiScalarField3D<int> > slice = generateMultiScalarField<int>
↪(geometry, sliceBox);
    plb_ifstream geometryFile(fNameIn.c_str());
    for (plint iX=0; iX<nx-1; ++iX) {
        if (!geometryFile.is_open()) {
            pcout << "Error: could not open geometry file " << fNameIn << std::endl;
            exit(EXIT_FAILURE);
        }
        geometryFile >> *slice;
        copy(*slice, slice->getBoundingBox(), geometry, Box3D(iX,iX, 0,ny-1, 0,nz-1));
    }

    {
        VtkImageOutput3D<T> vtkOut("porousMedium", 1.0);
        vtkOut.writeData<float>(*copyConvert<int,T>(geometry, geometry.
↪getBoundingBox()), "tag", 1.0);
    }

    {
        std::unique_ptr<MultiScalarField3D<T> > floatTags = copyConvert<int,T>
↪(geometry, geometry.getBoundingBox());
        std::vector<T> isoLevels;
        isoLevels.push_back(0.5);
        typedef TriangleSet<T>::Triangle Triangle;
        std::vector<Triangle> triangles;
        Box3D domain = floatTags->getBoundingBox().enlarge(-1);
        domain.x0++;
        domain.x1--;
        isoSurfaceMarchingCube(triangles, *floatTags, isoLevels, domain);
        TriangleSet<T> set(triangles);
        std::string outDir = fNameOut + "/";
        set.writeBinarySTL(outDir + "porousMedium.stl");
    }
}

void porousMediaSetup(MultiBlockLattice3D<T,DESCRIPTOR>& lattice,
        OnLatticeBoundaryCondition3D<T,DESCRIPTOR>* boundaryCondition,
        MultiScalarField3D<int>& geometry, T deltaP)
{
    const plint nx = lattice.getNx();
    const plint ny = lattice.getNy();
    const plint nz = lattice.getNz();

    pcout << "Definition of inlet/outlet." << std::endl;
    Box3D inlet (0,0, 1,ny-2, 1,nz-2);
    boundaryCondition->addPressureBoundary0N(inlet, lattice);
    setBoundaryDensity(lattice, inlet, (T) 1.);

    Box3D outlet(nx-1,nx-1, 1,ny-2, 1,nz-2);
    boundaryCondition->addPressureBoundary0P(outlet, lattice);
    setBoundaryDensity(lattice, outlet, (T) 1. - deltaP*DESCRIPTOR<T>::invCs2);

```

```
86      pcout << "Definition of the geometry." << std::endl;
87      // Where "geometry" evaluates to 1, use bounce-back.
88      defineDynamics(lattice, geometry, new BounceBack<T,DESCRIPTOR>(), 1);
89      // Where "geometry" evaluates to 2, use no-dynamics (which does nothing).
90      defineDynamics(lattice, geometry, new NoDynamics<T,DESCRIPTOR>(), 2);
91
92      pcout << "Initilization of rho and u." << std::endl;
93      initializeAtEquilibrium( lattice, lattice.getBoundingBox(),␣
    ↪PressureGradient(deltaP, nx) );
94
95      lattice.initialize();
96      delete boundaryCondition;
97  }
98
99  void writeGifs(MultiBlockLattice3D<T,DESCRIPTOR>& lattice, plint iter)
100 {
101     const plint nx = lattice.getNx();
102     const plint ny = lattice.getNy();
103     const plint nz = lattice.getNz();
104
105     const plint imSize = 600;
106     ImageWriter<T> imageWriter("leeloo");
107
108     // Write velocity-norm at x=0.
109     imageWriter.writeScaledGif(createFileName("ux_inlet", iter, 6),
110             *computeVelocityNorm(lattice, Box3D(0,0, 0,ny-1, 0,nz-1)),
111             imSize, imSize );
112
113     // Write velocity-norm at x=nx/2.
114     imageWriter.writeScaledGif(createFileName("ux_half", iter, 6),
115             *computeVelocityNorm(lattice, Box3D(nx/2,nx/2, 0,ny-1, 0,nz-1)),
116             imSize, imSize );
117 }
118
119 void writeVTK(MultiBlockLattice3D<T,DESCRIPTOR>& lattice, plint iter)
120 {
121     VtkImageOutput3D<T> vtkOut(createFileName("vtk", iter, 6), 1.);
122     vtkOut.writeData<float>(*computeVelocityNorm(lattice), "velocityNorm", 1.);
123     vtkOut.writeData<3,float>(*computeVelocity(lattice), "velocity", 1.);
124 }
125
126 T computePermeability(MultiBlockLattice3D<T,DESCRIPTOR>& lattice, T nu, T deltaP,␣
    ↪Box3D domain )
127 {
128     pcout << "Computing the permeability." << std::endl;
129
130     // Compute only the x-direction of the velocity (direction of the flow).
131     plint xComponent = 0;
132     plint nx = lattice.getNx();
133
134     T meanU = computeAverage(*computeVelocityComponent(lattice, domain, xComponent));
135
136     pcout << "Average velocity     = " << meanU                          << std::endl;
137     pcout << "Lattice viscosity nu = " << nu                             << std::endl;
138     pcout << "Grad P               = " << deltaP/(T)(nx-1)               << std::endl;
139     pcout << "Permeability         = " << nu*meanU / (deltaP/(T)(nx-1)) << std::endl;
140
141     return meanU;
```

```
}

int main(int argc, char **argv)
{
    plbInit(&argc, &argv);

    if (argc!=7) {
        pcout << "Error missing some input parameter\n";
        pcout << "The structure is :\n";
        pcout << "1. Input file name.\n";
        pcout << "2. Output directory name.\n";
        pcout << "3. number of cells in X direction.\n";
        pcout << "4. number of cells in Y direction.\n";
        pcout << "5. number of cells in Z direction.\n";
        pcout << "6. Delta P .\n";
        pcout << "Example: " << argv[0] << " twoSpheres.dat tmp/ 48 64 64 0.00005\n";
        exit (EXIT_FAILURE);
    }
    std::string fNameIn  = argv[1];
    std::string fNameOut = argv[2];

    const plint nx = atoi(argv[3]);
    const plint ny = atoi(argv[4]);
    const plint nz = atoi(argv[5]);
    const T deltaP = atof(argv[6]);

    global::directories().setOutputDir(fNameOut+"/");

    const T omega = 1.0;
    const T nu    = ((T)1/omega- (T)0.5)/DESCRIPTOR<T>::invCs2;

    pcout << "Creation of the lattice." << std::endl;
    MultiBlockLattice3D<T,DESCRIPTOR> lattice(nx,ny,nz, new BGKdynamics<T,DESCRIPTOR>
↪(omega));
    // Switch off periodicity.
    lattice.periodicity().toggleAll(false);

    pcout << "Reading the geometry file." << std::endl;
    MultiScalarField3D<int> geometry(nx,ny,nz);
    readGeometry(fNameIn, fNameOut, geometry);

    pcout << "nu = " << nu << std::endl;
    pcout << "deltaP = " << deltaP << std::endl;
    pcout << "omega = " << omega << std::endl;
    pcout << "nx = " << lattice.getNx() << std::endl;
    pcout << "ny = " << lattice.getNy() << std::endl;
    pcout << "nz = " << lattice.getNz() << std::endl;

    porousMediaSetup(lattice, createLocalBoundaryCondition3D<T,DESCRIPTOR>(),␣
↪geometry, deltaP);

    // The value-tracer is used to stop the simulation once is has converged.
    // 1st parameter:velocity
    // 2nd parameter:size
    // 3rd parameters:threshold
    // 1st and second parameters ae used for the length of the time average (size/
↪velocity)
    util::ValueTracer<T> converge(1.0,1000.0,1.0e-4);
```

```
197
198     pcout << "Simulation begins" << std::endl;
199     plint iT=0;
200
201     const plint maxT = 30000;
202     for (;iT<maxT; ++iT) {
203         if (iT % 20 == 0) {
204             pcout << "Iteration " << iT << std::endl;
205         }
206         if (iT % 500 == 0 && iT>0) {
207             writeGifs(lattice,iT);
208         }
209
210         lattice.collideAndStream();
211         converge.takeValue(getStoredAverageEnergy(lattice),true);
212
213         if (converge.hasConverged()) {
214             break;
215         }
216     }
217
218     pcout << "End of simulation at iteration " << iT << std::endl;
219
220     pcout << "Permeability:" << std::endl << std::endl;
221     computePermeability(lattice, nu, deltaP, lattice.getBoundingBox());
222     pcout << std::endl;
223
224     pcout << "Writing VTK file ..." << std::endl << std::endl;
225     writeVTK(lattice,iT);
226     pcout << "Finished!" << std::endl << std::endl;
227
228     return 0;
229 }
```

Line 1-11 General definitions and includes are discussed in the other tutorials of Palabos. Here we use the D3Q19 lattice, but other lattices (e.g. D3Q15 or D3Q27) are also possible.

Line 15-28 This functional defines the initial conditions. It is used in function *porousMediaSetup*: the fluid has initially zero velocity, with a linear pressure gradient in the x-direction.

Line 30-67 This function reads the porous medium geometry from the input file given as a command-line argument and creates some output.

Line 69-97 Here, the boundary conditions are set, i.e. all voxels read in as 1 are defined to have bounce back boundary conditions, and all voxels read in with value two are set to carry no dynamics. The inlet and outlet are defined to have a fixed pressure difference.

Line 99-142 Various outputs are created. During the simulation .gif files showing the velocity distribution within a slice are made (line 99-117). A .vti file with the velocity distribution is written, which can be read by Paraview (line 119-124). The permeability is computed by applying Darcy's law to the simulated velocity data (line 126-142).

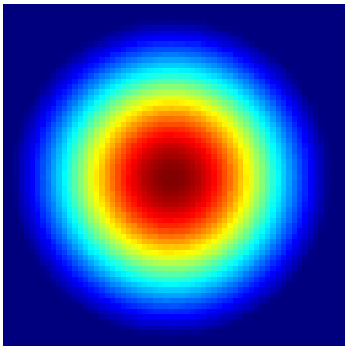
Line 144-229 Main part of the code. The code requires 6 input values from the user: the input.dat file, the output directory, the size of the file in each orthogonal direction, and finally the pressure difference between the inlet and the outlet. From these input values, the whole run is instantiated and the simulation starts (line 198). The ValueTracer (line 196) is used to stop the simulation when steady state is reached. A maximum of number of iterations is defined (line 201). Every 500 steps a .gif file is written to the output directory (line 207). Once converged the permeability of the medium is written to the screen and a .vti file is created.

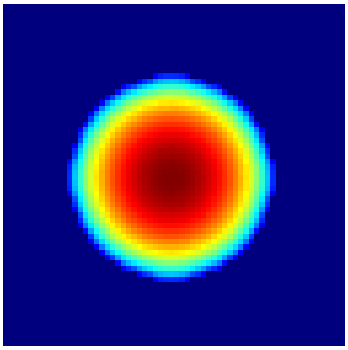
4.2.2 Running a simulation

First the code needs to be compiled. Check if the `Makefile` is in order. Open a terminal and type `make`, once in the permeability directory. To run a simulation type `./permeability path/to/input.dat path/to/outputdir/ nx ny nz deltaP` in a terminal in the permeability directory. Let's test it on our example by typing `./permeability twoSpheres.dat tmp/ 48 64 64 0.00005`. The progress of the simulation is written to the screen. One can find the .gif and .vti files in the tmp subdirectory.

4.3 Post-processing

Use our favorite image program to visualize the .gif files. Here we see the velocity distribution at the inlet (left) and halfway (right) after 4500 iterations:





Paraview allows you to visualize the .vti file in 3D. Here are some examples: