Palabos-UserGuide

Copy the User's Guide in palabos.org to here to generate PDF conveniently.

License

GNU AFFERO GENERAL PUBLIC LICENSE Version 3

Table of Content

Introduction

What is Palabos?

The Palabos library is a framework for general-purpose computational fluid dynamics (CFD), with a kernel based on the lattice Boltzmann (LB) method. It is used both as a research and an engineering tool: its programming interface is straightforward and makes it possible to set up fluid flow simulations with relative ease, or, if you are knowledgeable of the lattice Boltzmann method, to extend the library with your own models.

The library's native programmer interface in written in C++. It has practically no external dependencies (only Posix and MPI) and is therefore extremely easy to deploy on various platforms, including parallel systems like the BlueGene. Additional programmer intefaces are available for the Python and Java programming languages, which make it easier to rapidly prototype and develop CFD applications. There exists currently no graphical user interface; some amount of programming is therefore necessary in order to get an application running. Especially with the Python scripting interface, the developed code can however be pretty informal, and easy to produce with help of the provided examples.

The intended audience for Palabos covers scientists and engineers with a solid background in computational fluid dynamics, and potentially some knowledge in LB modeling. The software is freely available under the terms of an open-source AGPLv3 license, a copy of which is printed in the appendix of this user guide. It is distributed in the hope that it will promote research in the field of LB modeling, and help researchers concentrate on actual physical problems instead of staying stuck in tedious software development. Furthermore, implementing new LB models in Palabos offers a simple means of promoting new models and exchanging the information between research groups.

The Palabos library shows particularly outstanding performances when it comes to the field of high performance computing, or to the simulation of complex fluid flow. In the first case, close-to ideal scaling has been observed with Palabos on machines with tens of thousands of cores. A particularly striking feature here is that the preprocessing of the simulation is integrated into Palabos' parallel pipeline. While other tools struggle to create a mesh for really large problems, Palabos uses all the available computational power to build its mesh itself, in a matter of seconds. As for complex fluids, the multi-physics framework of Palabos makes it possible to combine a vast range of physical model, and yet maintain the original parallel efficiency. Examples of complex tasks that have been executed with Palabos include for example the simulation of thermal, multi-phase flow through porous media, simulated with pore-scale accuracy.

A substantial amount of development time for Palabos went into formulating a general programming concept for LB simulation which offers an appropriate balance between generality, ease of use, and numerical efficiency. The difficulty of this task can be understood if one considers that the core ingredients of the LB method are guided by physical considerations rather than criteria from numerical analysis. As a consequence, it is straightforward to implement a LB code for a specific physical model, a rectangular-shaped numerical grid, and simple boundary conditions. However, any of the extensions of the code required for the implementation of practical problems in fluid engineering require careful considerations and a solid amount of programming efforts. It is frequent to find LB codes which implement one specific advanced feature (for example parallelism), but then find themselves in a too rigid shape to allow other extensions (for example coupling of two grids for multi-phase flows). To cope with this problem, the various ingredients of Palabos' software architecture (physical model, underlying lattice, boundary condition, geometric domain, coupling between grids, parallelism) where largely developed as orthogonal concepts. It is for example possible to formulate a variant of the classical BGK collision model, say a MRT or entropic model, without awareness of the advanced software components of Palabos, and automatically obtain a parallel version of the code, or a multi-phase code based on the new model.

A central concept in Palabos are so-called "dynamics objects" which are associated to each fluid cell and determine the nature of the local collision step. Full locality of inter-particle collisions is a key ingredient of most LB models, a fact which is acknowledged in Palabos by promoting raw numerical efficiency for such models. Specific data structures are also available for algorithms that are not strictly local, as for example

specific boundary conditions or inter-particle forces in multi-phase models. It is however assumed that these ingredients have a marginal impact on the overall efficiency pattern of the program. They are therefore implemented with reasonable efficiency, but are excluded from the low-level number-crunching strategy of Palabos in favor of higher-level programming constructs. In practice, we observe that through this approach we get a minial penalty on the program performance, but substantially improve the readability of end-user programs.

The grid structure of the simulations is based on a multi-block approach. Each block behaves like a rudimentary LB implementation, while advanced software ingredients such as parallelism and sparse domain implementations are covered through specific interface couplings between blocks.

The C++ code of Palabos makes a massive use of genericicity in its many facets. Basically, generic programming is intended to offer a single code that can serve many purposes. On one hand, the code implements dynamic genericity through by means of object-oriented mechanisms. An example of such a mechanism is provided by the lattices sites which change their behavior during program execution, to distinguish for example between bulk and boundary cells, or to modify the fluid viscosity or the value of a body force dynamically. On the other hand, C++ templates are used to achieve static genericity. As a result, it is sufficient to write a single generic code in 3D, which then runs on the D3Q15, the D3Q19, and the D3Q27 lattice.

Functionality covered by Palabos

Currently implemented

In short, the current release of Palabos covers the following ingredients:

Physics: > Incompressible Navier-Stokes equations, weakly compressible, non-thermal Navier-Stokes equations, flows with body-force term, thermal flows with Boussinesq approximation, single-component multi-phase fluids (Shan/Chen model), multi-component multi-phase fluids (Shan/Chen model, He/Lee model), free surface flows (volume-of-fluid approach), static Smagorinsky model for fluid turbulence.

Basic fluid models: > BGK (and its "incompressible" counterpart), a given MRT model (and its "incompressible" counterpart), regularized BGK, LW-ACM (and its "incompressible" counterpart), a given entropic model.

Straight-wall boundary conditions: > Zou/He, Inamuro, Skordos, regularized BC, simple equilibrium, bounce-back, periodic. All boundary conditions work for straight walls with interior/exterior corners, and can be used to implement a Dirichlet or Neumann condition for the velocity or the pressure. The bounce-back condition is also used for curved boundaries, represented by a stair-case shape.

Off-lattice boundary conditions: > GUO model, and generalized off-lattice boundary condition. Automatic, and massively parallel, voxelization of STL file and instantiation of off-lattice walls.

Particles: > Massively parallel (billions of particles are no problem on a parallel machine) simulation of passive scalars, or interacting particles.

Grid: > The implemented grids are D2Q9, D3Q13, D3Q15, D3Q19, and D3Q27. In all cases, the domain is either a regular matrix or a sparse domain, approximated by a multi-grid pattern.

Parallelism: > All mentioned models and ingredients are parallelized with MPI for shared-memory and distributed-memory platforms, including I/O operations that are implemented in terms of MPI's Parallel I/O API.

Pre-processing: > The domain of a simulation can be constructed manually, or automatically from a corresponding STL-file.

Post-processing: > The code has the ability to save the data in ASCII or binary files or to directly produce GIF images. Furthermore, the data can be saved in VTK format and further post-processed with an appropriate tool. For better efficiency, Palabos can natively post-process data, producing streamlines and iso-surfaces.

Check-pointing: > At every moment, the state of the simulation can be saved, and loaded at a later point.

Under development:

The following features have been repeatedly requested by the community and are currently developed:

- Grid refinement.
- Thermal flows using lattices with extended neighborhoods.

Project

The FlowKit Ltd. technology company manages the development of the Palabos code. It carefully reviews each line of contributed code, asserts the quality of the implemented models and algorithms, and promotes industrial and engineering usage of the code. FlowKit Ltd. also provides consulting solutions to help you get started with the code, to implement new models, or to solve specific problems.

The University of Geneva is the main research partner of the Palabos project. A pioneer in the domain of computational fluid dynamics and lattice Boltzmann modeling, the Scientific and Parallel Computing Group SPC at the University of Geneva provides the theoretical background of the Palabos kernel, and continuously develops new models and approaches in the field of complex fluid dynamics.

Authors

Project supervision and core development are taken care of by Jonas Latt (FlowKit Ltd.+Universite de Geneve, Switzerland). Orestis Malaspinas (FlowKit Ltd.+Universite de Geneve, Switzerland) writes alternative LB collision terms (MRT, entropic), some of the boundary conditions, and coupled physical models (thermal, multi-phase). Free-Surface and multi-phase models are actively developed by Andrea Parmigiani (Universite de Geneve, Switzerland) and by Dimitrios Kontaxakis (FlowKit Ltd, Switzerland). Grid refinement is taken care of by Daniel Lagrava (Universite de Geneve, Switzerland). Many other people have contributed by writing code, thinking about programming concepts, or understanding some of the LB models for which the literature is sparse. We particularly mention, in alphabetical order, the contributions of Fokko Beekhof (Universite de Geneve, Switzerland), Bastien Chopard (Universite de Geneve, Switzerland), Mathias Krause (Rechenzentrum Karlsruhe, Germany), Switzerland), and Bernd Stahl (Universite de Geneve, Switzerland).

Getting help

Installation instructions and a short overview of the code are provided in the section Getting started with Palabos. Additionally to the user guide, the following resources are available as a support for Palabos users:

The tutorial: > It is highly recommended, to get started, to work through Tutorial 1. The tutorial provides both an overview of high-level structures, and information on the way Palabos works internally.

The forum: > The forum is the right place to post questions which are not answered in the user guide or the FAQ, to exchange with other Palabos users, or to communicate with the Palabos authors.

The automatic code documentation: > This guide is automatically generated from the source code, using the program Doxygen . It has little didactic value but offers detailed insight into the class hierarchies and the syntax of the function calls.

Getting started with Palabos

Supported Compilers

We donot provide a detailed list of compilers and compiler version numbers which successfully compile Palabos. The bottom line, however, is that we have tested GCC, the Intel compiler, and the Portland Group compiler, which all work fine if you use recent versions.

Installing and compiling the code under Linux and other Unix-like systems

The most recent version of Palabos can be downloaded at the address www.palabos.org. It is now shown how to compile and run the software under Linux or other Unix-like environment. The library is exempt from external dependencies: all you need is a working compiler environment, including a modern C++ compiler (for example the free GCC compiler), the command make, and a reasonably recent version of the Python interpreter. To get started, download the tarball of the most recent version of Palabos, for example `plb-v1.1r0.tgz`, and move it to the directory in which you wish the code to reside (we will refer to this directory under the name of `$(PALABOS))`. In the following, the code is staying at this location, as there is currently no explicit installation process. Instead, compiled libraries are deposited in the directory `$(PALABOS)/lib/`. Unpack the code, using for example the command `tar xvfz plb-v1.1r0.tgz`.

The library Palabos makes use of an on-demand compilation process. The code is compiled the first time it is used by an end-user application, and then automatically re-used in future, until a new compilation is needed due to a modification of the code or compilation options. To see how this works, change into the directory of one of the example programs, such as `$(PALABOS)/examples/showCases/cylinder2D/`. Type `make` to compile both the `Palabos` library and the example program under standard conditions with the GCC compiler, and execute the program with the command `./cavity2D`. The program simulates the 2D flow around a cylinder; if the software `ImageMagick` is installed on your system, the program saves GIF images in the `tmp/` subdirectory, representing the velocity field at selected time steps.

To use a different compiler, compile for parallel execution, or modify other compilation options, edit the file Makefile in the local directory. The following options are particularly often modified:

Options	Function
debug:	Turn on this flag to compile in debug mode. The resulting executable is a bit slower but easier to analyze in search for bugs. The recommended default behavior is to turn on debug mode.
MPIparallel:	Turn on this flag to compile for parallel execution.
serialCXX:	Specify the compiler to use for serial programs.
parallelCXX:	Specify the compiler to use for parallel programs.
optimFlags:	Specify the compiler options to use when the flag `optimize` is true.
compileFlags:	Specify additional compiler options, which are for example specific to your hardware environment. Typical options are `-DPLB_BGP` or `-DPLB_MAC_OS_X` when compiling on a BlueGene or on a Mac respectively.

If MPI is installed on your system (and if you have several cores or processors available), you can try to compile the code in parallel and execute it, say with two threads, through a command of the type `mpirun -np 2 ./cavity2d`. Note that the execution time of the example program is dominated by output operations. **To observe a significant speed improvement in the parallel program, the output operations need first to be commented in the source code.**

Compilation under Mac OS X

Under Mac OS X, an easy way of getting GCC is to install xcode, after which the procedure is the same as under Linux. Alternatively, you can follow the same procedure as under Windows. **Important**: In either case, you must define the preprocessor macro PLB_MAC_OS_X (set the corresponding line in the Makefile to `compileFlags= -DPLB_MAC_OS_X`).

Compilation under Windows (and under many other OSes) with the integrated development environment Code::Blocks

The C++ Integrated Development Environment (IDE) Code::Blocks is free, and you can download binaries for various flavors of Windows and Linux, and for Mac OS X (see codeblocks). A configuration file is delivered with the Palabos releases which can be used to compile Palabos with Code::Blocks. This approach was successfully tested under Windows and Linux, and neither Python nor `make` are needed for the compilation process.

Open the project file `$(PALABOS)/codeblocks/Palabos.cbp` under code::blocks, build and run the project. By default, the example program `examples/showCases/cavity2d.cpp` is compiled. To compile another example or your own program, remove the file `cavity2d.cpp` from the list of source files, and replace it by another one. The GIF images and other output files from the program are written in the directory `$(PALABOS)/codeblocks/tmp/`.

Code::Blocks is a cross-compiler environment, and you can in principle use it with just any C++ compiler. In particular, you can use it with the free GCC compiler. This compiler does not even need to be installed separately under Windows if you download the version of Code::Blocks which is directly packaged with the GCC port MinGW.

Note that the Palabos code does not compile with Visual C++, and we donâ€™t know why; any suggestion is appreciated. In the meantime, you should use another compiler, such as GCC, the Intel compiler, or the Portland Group compiler.

As mentioned above, the Palabos download provides a project file for the free IDE Code::Blocks, through which you can easily use any of these compilers (we have only tested GCC/MinGW, though). If you have no C++ compiler installed on your system (other than Visual C++), remember to download the version of Code::Blocks which is packaged with the GCC port MinGW, after which you can directly compile Palabos.

Just as under Linux, you need to install `ImageMagick` if you want Palabos to be able to produce GIF images. If this fails, Palabos will write images in the PPM format, which is not recognized by most of the standard image viewers under Windows. In this case, you have the possibility to install the free image editor `Gimp`, which can view PPM images and convert them to another format.

It should also be reminded that the pathnames in the Palabos examples are written using the Unix convention, with slashes between directory names, while Windows uses backslashes. Under Windows, it is therefore recommended to change lines like

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

to something like

```
global::directories().setOutputDir(".\\tmp\\");
```

although the programs appear to behave reasonably well even when you donâ€™t do this.

Compilation on a BlueGene/P

On the BlueGene/P, the compilation procedure is the same as on any Unix-like system. Just remember to define the preprocessor macro `PLB_BGP` (set the corresponding line in the Makefile to `compileFlags=`

`-DPLB_BGP`). Also, note that on the BlueGene, the GCC compiler produces code that is almost as fast as (and sometimes even than) the dedicated XL compiler. Furthermore, GCC compiles much faster.

It must also be pointed out the BlueGene/P uses a Big-Endian representation of numerical values, as opposed to x86 architectures that use Little-Endian. You can therefore not export binary checkpoint files from the BlueGene to an x86 computer. BlueGene-generated VTK files on the other hand can be read on an Intel PC: just replace the string `LittleEndian` to `BigEndian` in your VTK file. You can for example to this with `sed`: `sed -i "s/LittleEndian/BigEndian/g" myOutputFile.vti`. As for the input STL meshes, it is simplest to provide them in an ASCII STL format, which is platform independent. If your mesh happens to be binary STL, you can convert it to ASCII using the program `toAsciiSTL` in the directory `utility/stl`.

Open-source libraries which are bundled with Palabos

Palabos makes use of a few other open-source libraries. They doenâ€™t need to be explicitly intalled, though, because they are part of the Palabos releases:

SConstruct (Only tested with Linux):

> This Python-based library is used in Palabos to manage the compilation process.

TinyXML :

> This library is used to read structured user input in XML format.

Recommended open-source software

Although the Palabos library is self-contained, it makes sense to combine it with other products, especially for the pre- and post-processing of data. A few recommended open-source programs are listed below:

Paraview (Linux / Mac OS X / Windows):

> Paraview offers a graphical user interface for the VTK library, and is an excellent tool for the visualization of scientific data. The results of Palabos simulations can be saved in a VTK format and then processed with Paraview. Paraview is found in the software repository of some Linux distributions.

Octave (Linux / Mac OS X / Windows):

> This command-line interpreter is very similar to Matlab. It is useful for processing ASCII data produced by Palabos, to create for example plots. Octave can be found in the software repository of most Linux distributions.

ImageMagick (Linux / Mac OS X / Windows):

> Without need for an external library, Palabos can produce snapshot images in a PPM format. If ImageMagick is installed, these pictures can be automatically converted into the more common GIF format. Furthermore, ImageMagick can be used to produce an animated GIF from a sequence of GIF images.

Meshlab (Linux / Mac Os X / Windows):

> Meshlab converts surface mesh files from many different formats into the STL format, which is required by Palabos to set up the geometry. Also, Meshlab is often able to correct mistakes in the STL file, such as, remove zero-area triangles or revert the surface normal.

The examples directory

The examples directory is divided into the three sub-directories `showCases`, `codesByTopic`, and `tutorial`. While the examples in the directory `showCases` contain full-featured implementation of simple benchmark flows, the examples in directory `codesByTopic` contain more technical code to illustrate a specific programming concept in Palabos. The content of the directory `tutorial` is discussed separately in the Palabos tutorial.

The example programs are listed and commented in the appendix Appendix: list of example programs.

The Palabos-Python interface

Important: To get Palabos up and running, it is not necessary to compile the Palabos-Python interface. The Python interface is a useful extension which you will find useful to develop your Palabos applications more rapidly.

Since version 0.7, Palabos is provided with a Python interface which offers simple scripting access to the library. The look-and-feel of the Python interface is very different from the C++ interface, and a separate user's guide will be provided at some point for this library. In the mean time, this section provides a few instructions for compiling and using the Python interface.

Compiling the Python interface

For the Palabos-Python interface to work, you must install a list of libraries, and compile Palabos to create the interface. The required libraries are part of all major Linux distributions and are easily installed through the distribution's package system. When your package manager offers plain and a `-dev` version of the libraries, you must install both of them. The required programs/libraries are:

- Swig
- NumPy
- SciPy
- Matplotlib
- Mpi4py (to install this, you may need the Python package installation system `easy_install`. On Ubuntu, you get this by installing python-setuptools. Then, type `sudo easy_install mpi4py`).

And, of course, you need Python. Optionally, you can install Mayavi2 for better graphics.

To compile the Palabos-Python interface, you need a working MPI installation and a C++ wrapper for MPI (this is usually done by installing either mpich or openmpi through the package manager). Then, follow these steps:

- Find out in which directory the source code of Palabos is located. Define an environment variable `PALABOS_ROOT` that holds this directory name. In a bash shell, you would for example type a command like `export PALABOS_ROOT=/home/joe/palabos-0.7v0/`.
- Change into the sub-directory `pythonic/src` of Palabos. Type `make`
- If the compilation is sucessful, change into the sub-directory `pythonic/examples` and type the name of one of the exmamples to execute it.

Note that the compilation is slow, and can take as much as half an hour. If the compilation fails with a "file-not-found" error message, this might be due to an improperly installed library. You can circumvent the problem by finding the directory in which the file in question is located, and add the directory to the include paths. For this, add the Makefile, and edit the line `includePaths=...` correspondingly.

Example: Compiling the Python interface under Ubuntu Linux

The following step-by-step guide to compiling the Palabos-Python was tested under both a Version-9 and a Version-10 series of Ubuntu Linux. Note that the exact name of the used packages can vary a bit in your distribution.

1) Install the required packages

Open the package manager, and install the following packages: `g++`, `mpi-default-bin`, `mpi-default-dev`, `swig`, `python`, `python-dev`, `python-setuptools`, `python-numpy`, `python-matplotlib`, and `mayavi2`.

Then, open a terminal and type `sudo easy_install mpi4py` to install the Mpi4Py library which is not found in Ubuntu's package manager.

2) Compile the Palabos-Python interface

Open a terminal, and unpack the downloaded tar-ball (e.g. `tar xvfz palabos-0.7v0.tgz`), and change into the source directory of the Palabos-Python interface (e.g. `cd /home/joe/palabos-0.7v0/pythonic/src`). Define the PALABOS_ROOT environment variable (e.g. `export PALABOS_ROOT=/home/joe/palabos-0.7v0/`). Compile the libraries by typing make.

3) Execute an example program

Change into the examples directory (e.g. `cd /home/joe/palabos-0.7v0/pythonic/exmples`) and execute one of the programs (e.g. `./cavity2d`). It can also be illuminating to execute the programs interactively by typing the commands into a Python prompt. To do this, display the content of one of the examples (e.g. `cat cavity2d`), run the Python interpreter through the command `python` and type the lines of the program manually, inspecting the results as you go.

The Palabos-Java interface

Important: To get Palabos up and running, it is not necessary to compile the Palabos-Java interface. The Java interface is a useful extension which you will find useful to develop your Palabos applications more rapidly.

Since version 1.1, Palabos is provided with a Java interface which offers an access to the library to programmers lacking a C++ culture. The look-and-feel of the Java interface is very different from the C++ interface, and a separate user's guide will be provided at some point for this library. In the mean time, this section provides a few instructions for compiling and using the Java interface.

Introduction

Jlabos is a java wrapper of the CFD C++ opensource library `Palabos` It is now part of Palabos but optionaly compiled.

Installation

Requirement

- swig1.3

- javac + java (for example openjdk-6-jdk
- mpicxx (for example libopenmpi-dev
- mpirun (for example openmpi-bin
- python

Optional:

- octave
- subversion
- convert
- inkscape

Installation

You need to have Palabos installed and configured. Short explanation: download Palabos

Define env variable `PALABOS_ROOT`. For example add this line in `~/.bashrc`: `export PALABOS_ROOT=~/location_of_palabos`

`cd $PALABOS_ROOT/jlabos/src` edit the `Makefile` nbOfCore = 4 // Number of CPU core used to do the compilation. Do not specify more than you realy have or it will be very slow. swigCompiler = swig // you can specify an alternate swig executable includePaths = // specify the location of the file `jni.h` and `jni_md.h`

Compile Jlabos : `cd $PALABOS_ROOT/jlabos/src; make` This will take a lot of time because it will compile Palabos at the same time.

Running the examples

go to the choosen example directory and type `make`. This will compile and launch the program. You can customize some elements in the Makefile. Do not launch directly the SConstruct file. If you need to specify another place for the shared libs, edit the file `launch` accordingly.

To clean the directory, you can use `make clean`

See the result

If needed you can plot the results with Octave or Matlab like this: `./plot.m` Jlabos can as well directly produce jpg files using the embeded library MLplot. See the examples code source. MLplot need the program `convert` from `ImageMagick`

Customize the wrapper

Jlabos is the Palabos wrapper but it is not complete. If you need more functionalities there are two places to go:

$PALABOS_ROOT/jlabos/src/jlabos => here you find high level java files that wrap to the autogenerated files here: $PALABOS_ROOT/jlabos/src/swig/pre_compiled => Swig generated files from the `.i` files present at the same place. Do not edit the java files here but instead modify the `.i` files and re-run the make process. But be warned, it's not easy to obtain something that is working!

If the functionality that you want is present in the second place, you can modify the files in the first place to wrap nicely the self generateds java files.

file

```
|---examples
|   |--- Cavity2d
|   |--- Cavity3d
|   |--- Cylinder2d
|   |--- Cylinder3d
|   |--- Porous
|--- src
    |--- compilePalabos
    |--- jlabos
    |   |--- plb
    |       |--- jlabos
    |--- lib
    |--- plbWrapper
    |   |--- block
    |   |   |--- double
    |   |   |--- float
    |   |   |--- int
    |   |--- lattice
    |   |   |--- d2q9_double
    |   |   |--- d2q9_float
    |   |   |--- d3q19_double
    |   |   |--- d3q19_float
    |   |--- utils
    |--- precompiled
    |   |--- floatOnly
    |   |   |--- double
    |   |   |--- float
    |   |--- intOrFloat
    |   |   |--- double
    |   |   |--- float
    |   |   |--- int
    |   |--- lattice
    |       |--- d2q9_double
    |       |--- d2q9_float
    |       |--- d3q19_double
    |       |--- d3q19_float
    |--- swig
    |   |--- pre_processed
    |--- testing
    |--- util
```

Technical details One part of the wrapper is generated using the Swig library (Simplified Wrapper and Interface Generator). This step produce not very usable Java files. This is why we had to write a second wrapper in Java to hide this first layer to the end user. Jlabos is not a complete wrapper for Palabos, but will evolve over time.

The directory compilepalabos is just a dummy project to force the compilation of Palabos

jlabos

In the swig directory we have all the `.i` files needed by swig.

plbwrapper contains some Palabos wrappers written in C++ and used by swig.

Programming with Palabos

Quick overview: programming guidelines

C++ is an exceptionally free language which imposes little restrictions on programming style and coding structure. It is therefore not sufficient to know the language C++ to start working with a library like Palabos; you also need to get familiar with the particular algorithmic choices and programming guidelines which were issued for the library. Once you know the rules, you'll understand more precisely what the code does and avoid the many pitfalls of programming such as memory leaks, inconsistent parallel program execution, unexpected behind-the-scene effects of object-oriented programming, and more. The user's guide tries to get you familiarized progressively with these concepts. Whenever a new concept is introduced which requires a certain amount of discipline from the programmer to guarantee correct results, the importance of this concept is highlighted in form of a rule:

Rule for this topic: You've been warned: follow this rule or your program's result is undefined.

Some of the most fundamental programming rules in Palabos are reviewed in this section, in order to clarify aspects of the code that appear puzzling at a first reading.

Basic data types

In Palabos, the signed and unsigned integer data types `int` and `unsigned int` are systematically replaced by `plint` and `pluint` (with only a few exceptions). The reason for this is based on 64-bit compatibility of the code and is easy to understand. A 32-bit signed integer can count from -2 Billion to +2 Billion, whereas a 64-bit integer goes far beyond this. The 32-bit version is therefore often insufficient in simulations with data sets larger than 2 GB. Errors in relation with the 32-bit vs. 64-bit issue are highly unpleasant, and believe me, you don't want them. Under this light, guess what's the size of the type `int` on a 64-bit platform? For some obscure reason, the answer is platform dependent, but most current 64-bit architectures made the choice of the data type `int` being 32-bit. The solution to this problem are the Palabos data types `plint` and `pluint` which are 32-bit on a 32-bit machine, and 64-bit on 64-bit machine. In conclusion, always prefer the integer data types `plint` and `pluint` over `int` and `unsigned int`, unless you have specific reasons in favor of `int` and you are very, very certain that your reason is valid.

Another cherished C++ object which you must abandon is the output stream buffer `cout`. In an MPI-parallel program each thread writes to the terminal through `cout`. To avoid this and observe a consistent behavior between sequential and parallel programs, use the buffer `pcout` instead. Similarly, replace `cerr` by `pcerr` and `clog` by `pclog`. For file output, replace `ofstream` by `plb_ofstream`.

Memory management

As C++ has no garbage collector, objects allocated dynamically with the operator `new` must be disposed of manually with the operator `delete`. Keeping track of these objects and de-allocating them at the right moment is an unpleasant burden imposed by this language. Unlike other libraries, Palabos does not include an external garbage collector or similar construct to work around the problem. Instead, discipline is the solution, and you'll avoid problems by respecting a few rules.

In general, when a function in Palabos takes a pointer to an object as a parameter, this means the function (or the object, in case of a class method) assumes responsibility over the object. You're not responsible for its deletion any more. As a matter of fact, you are not even *allowed* to delete it or, by all means, do anything with it (because you don't know at which point it is deleted). Reversely, when you get a pointer returned to you from a function, you become the owner of the object pointed to. Do what you need to do with it, and don't forget to delete it afterward.

Issues around the question of who deletes an object pointed to by many instances are most often avoided in Palabos through a one-pointer-per-object approach. Whenever a second instance needs to access an object

which is already in use, it creates its own copy. To make such copies possible, all Palabos classes (or at least those which use inheritance) have a method `clone`:

```
A* object1 = new A;
A* object2 = object1->clone();
```

You need to implement this method whenever writing a class which inherits from Palabos types. Luckily enough, the method `clone` has always exactly the same shape:

```
A* A::clone() const {
    return new A(*this);
}
```

There's unfortunately no mechanism in C++ for doing this automatically, but you'll soon be in the habit of writing it without noticing. As a side remark, it is not advised to define the method `clone` by different means than the one suggested here, for the sake of uniformity and readability of the code. Some classes will require that you write a copy constructor in order for this form of the method `clone` to be valid. However, if you write this type of classes, it is assumed that you know what you are doing. And that you knew you had to define the copy constructor.

Arrays

The right choice of a data type to store arrays of values depends basically on the size of the data. Large data sets, like the particle populations of a lattice Boltzmann simulation, are of course stored in one of Palabos' BlockLattice types. These specialized and parallelized data types are after all the reason why you started using Palabos in the first place. The syntax for instantiating such a type looks as follows:

```
// Instantiate a nx*ny*nz D3Q19 lattice with double-precision
//   floating point numbers.
MultiBlockLattice<double,D3Q19descriptor> lattice(nx,ny,nz);

// Instantiate a nx*ny*nz matrix of double-precision floating
//   point scalars.
MultiScalarField<double> field(nx,ny,nz);
```

Small, non-parallelized arrays are needed for example to store parameters of a simulation, or a couple of results which require little memory. The standard library of C++ offers the data type vector for the instantiation of dynamically sized value arrays. The vector type is efficient, elegant and easy to use:

```
plint numspecies = 5;
// Instantiate a vector of double-precision floating point
//   numbers with 5 elements.
vector<double> viscosities(numSpecies);
viscosities[3] = 0.63;
```

It is strongly recommended to prefer the type `vector` over old-style dynamical arrays allocated with the `new []` operator, to reduce the risk of programming errors and to improve the readability of the code.

For very small, non-parallelized and fixed-sized arrays, Palabos defines the type `Array`. It is commonly used to store small physical vectors, such as, a single velocity value:

```
// Instantiate a fixed-size array of type double and with
//   3 elements. The values can be initialized directly in
//   the constructor for arrays of size 2 and 3.
Array<double,3> velocity(0., 0., 0.5);
velocity[0] = 0.1;
```

Again, the type `Array` should be preferred over the fixed-size array inherited in C++ from the language C (`double velocity[3]`). From an efficiency point of view, both choices are equivalent, as the type `Array`

is defined on top of the C-array through a template mechanism. It offers however specific advantages. In particular, it performs range checks when Palabos is compiled in debugging mode, a feature which can substantially reduce the time spent in debugging.

Velocity and Density

Velocity and mass density are the two most recurrent macroscopic variables in lattice Boltzmann. It is therefore surprising that variables with names like `rho` or `u` rarely occur inside the Palabos code (nothing prevents you from perusing them in end-user code, though). Instead, internal Palabos structures most often work with the variables `rhoBar` and `j`. The variable `j` is nothing else than the first-order velocity moment which, for most models, is identical with the fluid momentum: `j = rho * u`. The definition of the variable `rhoBar` on the other hand can be customized. By default, it is defined as `rhoBar=rho-1` and is used to improve the numerical accuracy of the method. Indeed, as the density is often close to 1, you gain significant digits in the representation of floating point variables by representing a quantity fluctuating around 0 rather than 1.

Without getting into details, here are a few rules for switching between the `rho / u` and the `rhoBar / j` representation:

```
// Use custom definitions in lattice descriptor to compute rho from rhoBar
rho = Descriptor<T>::fullRho(rhoBar);

// Use custom definitions in lattice descriptor to compute rhoBar from rho
rhoBar = Descriptor<T>::rhoBar(rho);

// Momentum is equal to density times velocity (component-wise)
j[iD] = rho * u[iD];

// Velocity is equal to inverse-density times momentum (component-wise)
u[iD] = 1./rho * j[iD];

// Inverse-density can be computed right away from rhoBar. Depending on
//   the custom definitions in the descriptor, this is substantially more
//   efficient than computing 1./rho, because the Taylor expansion of the
//   inverse-function is truncated at an appropriate position.
u[iD] = Descriptor<T>::invRho(rhoBar) * j[iD];
```

Parallelism

The library Palabos uses a single-program-multiple-data (SPMD) approach to parallelism, no matter whether a shared-memory or distributed-memory platform is used. This means that your program looks the same when you write it for sequential or parallel execution (at least, it should look the same, if you are a little careful). In a distributed-memory environment ("a cluster"), multiple instances of the same program are started in distinct threads. In that case, Palabos' distributed objects (`MultiBlockLattice`, `MultiScalarField`, and `MultiTensorField`) are parallelized, i.e. the amount of memory they allocate whithin each thread is divided by the total number of threads. All other data is, in general, duplicated over all threads. There are a few exceptions to this rule, though, which are highlighted in the user's guide. For example, the default behavior of reduction operations (such as computing the average density of a block lattice) is to store the result in the main thread only, for efficiency reasons.

Non-intrusive program development with Palabos

For physicists and scientific programmers with little experience in software engineering, collaborative programming often means exchanging code snippets by e-mail and integrating them manually into existing code.

In this approach, every programmer possesses an entirely independent version of the code, and adjusts its components to fit a specific task.

This section tries to convince you to not adopt this approach with Palabos, and to use the code of this project like a library instead of a code template which is adjusted to your local needs. There are several reasons why such a non-intrusive way of working with Palabos is more efficient in the short and the long run. First of all, the Palabos source code is quite large, with more than 50'000 lines of code, and starting to modify the code as a whole rapidly degenerates into a very time consuming task. A second, even more compelling reason is that, as soon as you modify your local version of Palabos, your copy becomes independent of the main version. It becomes difficult or even impossible to take profit of subsequent Palabos releases, with bug fixes, efficiency improvement, and new features. Furthermore, you loose the possibility to easily exchange code snippets with colleagues, because you're somehow forced to shuffle around full copies of the Palabos source all the time. Compare this with the relative simplicity of exchanging just two code files based on the official Palabos version, one of which implements a new dynamics class for your own LB model, and the other a sample application.

Imagine that you are producing advanced 3D graphics based on the library OpenGL. You would certainly think about how to formulate your problem in terms of the interface offered by OpenGL, instead of modifying the source of OpenGL itself right? Think of Palabos in the same way, and you're guaranteed to save a lot of time, even though you are forced comply with a certain amount of decisions which were taken for the Palabos project.

Extending without modifying

Imagine you want to write a LB model similar to BGK, with small differences. The first thing to do is to check out section `Implemented fluid models` and make sure it is not yet implemented in Palabos. If it's not, you'll probably want to view the files `src/basicDynamics/isothermalDynamics.h` and `.hh` and discover that the class `BGKdynamics` inherits from `IsoThermalBulkDynamics` and implements the three methods `clone()`, `collide()`, and `computeEquilibrium()`. From the preceding discussion, you understand that the next step is not to modify the implementation of `collide()` in BGKdynamics, because (among many other reasons) you would then loose the ability to use both the original BGK dynamics and your new model at the same time in an application. Instead, create two new files `myNewModel.h` and `myNewModel.hh`, containing a class `MyNewDynamics` which inherits from `IsoThermalBulkDynamics` and defines the same three methods as `BGKdynamics`, adapted to your case. This implementation is clean: it does not conflict with updates to newer Palabos releases, and you can share it with colleagues by exchanging just two files.

Your new files can be located in any local directory, independent of the Palabos directory tree. Just make sure to add this directory under the keyword `includePaths` in the Makefile, so that it can be found during the compilation process.

Which parts of Palabos are extensible?

There are, roughly speaking, three ways to extend Palabos: (1) **Write a new dynamics class**, (2) **Write a new data processor**, and (3) **Write a new lattice descriptor**. The dynamics class is used to define a new local collision step, whereas a new lattice descriptor means a new lattice topology (discrete velocities, weights, etc.), and the data processor is for everything else. Technically speaking, you can also adapt Palabos' behavior by writing a custom class for floating point numbers, but this is a less common thing to do. Once you have written new dynamics classes, data processors, and/or lattice descriptors, you might of course also want to write convenience code such as wrapper functions which automatically deliver the right data processor for a given task, or which automatically add the data processor to a block-lattice. As an example of such convenience code, consider the `OnLatticeBoundaryConditionXD` classes which automatically add dynamics objects, data processors, or both to a block-lattice to create a boundary condition.

All other parts of Palabos are not extensible. It is for example a bad idea to write a new class similar to the `MultiBlockLatticeXD`, or to re-invent the `MultiScalarFieldXD`. Nothing prevents you from doing so,

of course, but redefining such a class breaks the concept of modularity in Palabos. For short, a custom implementation of a data processor is able to take profit of future innovation in Palabos, such as new approaches to parallelism, whereas a user-invented version of a `MultiBlockLatticeXD` is difficult to write and not supported by subsequent Palabos releases.

The good news is that the extensible parts of Palabos allow a lot of flexibility. Take as an example the implementation of the Shan/Chen model for the 2D and the 3D implementation of single-component multi-phase and immiscible multi-component fluids. This was simply achieved by writing a new lattice descriptor and a new data processor, without modification to any structural parts of Palabos. And, the size of the resulting source code makes up for as little as one percent of the whole project.

Sometimes, the choices made in Palabos may seem a little restrictive, though. A problem appeared for example when we implemented the thermal fluid model with Boussinesq approximation. In this case, the temperature field follows an advection-diffusion equation which is solved by means of a LB model with linear equilibrium. From the previous discussion, it is clear that this simply requires the definition of a new dynamics class which implements the collision step of the advection-diffusion equation. The problem is that the order-0 moment of the particle populations is always called "rho" in Palabos (this is imposed by the interface of dynamics classes), while the order-0 moment of the temperature model is, well, the temperature. At this point, we had to overcome our strong sense of discipline and order, accept that the temperature is called "rho" at two or three places inside the code, add a few comments to make sure other programmers are not confused by this, and wrap the final product into convenience functions which call the temperature temperature. Programming is always an exercise in compromise, and it appears that avoiding to re-write hundreds of thousands of lines of code is worth the small heresy of calling the temperature "rho" on a few occasions.

Fundamental data types

The `BlockXD` data structures

The fundamental data structures which hold the variables of a simulation are of type `Block2D` for 2D simulations, and of type `Block3D` for simulation (in the following, we use the generic name `BlockXD` to keep the discussion short, although there exists no type `BlockXD` in Palabos). From the standpoint of a user, a `BlockXD` construct represents a regular 2D or 3D array (or matrix) of data. Behind the scenes, they are sometimes really implemented as regular arrays and sometimes as more complicated constructs, which allows for example memory savings through sparse memory implementations, or parallel program executions based on a data-parallel model.

The `BlockXD` structures are specializes for different areas of application. One type of specializations is used to specify the type of data stored in the blocks. To store the particle populations of a lattice Boltzmann simulation, and potentially other variables such as external forces, you'll use a specialization in which the name Block is replaced by `BlockLattice`. To store a spatially extended scalar variable, the data type to use is a variant of the `ScalarFieldXD`, whereas vector- or tensor-valued fields are stored in a `TensorFieldXD` or similar. A second type of specialization is applied to specify the nature of the underlying data structure. The `AtomicBlockXD` data structure stands essentially for a regular data array, whereas the `MultiBlockXD` is a complex construct in which the space corresponding to a `BlockXD` is partially or entirely covered by smaller blocks of type `AtomicBlockXD`. The `MultiBlockXD` and the `AtomicBlockXD` have practically the same user interface, and you are urged to systematically use the more general `MultiBlockXD` in end-user applications. It is almost as efficient as the `AtomicBlockXD` for regular problems, it can be used to represent irregular domains, and it is automatically parallelizable.

The following figure illustrates the C++ inheritance hierarchy between the various specializations of the `BlockXD` :

The figure shows, as an example, a few functions implemented at each level of the inheritance hierarchy. For example, all blocks of type "block-lattice" have a method `collideAndStream()`, no matter if they

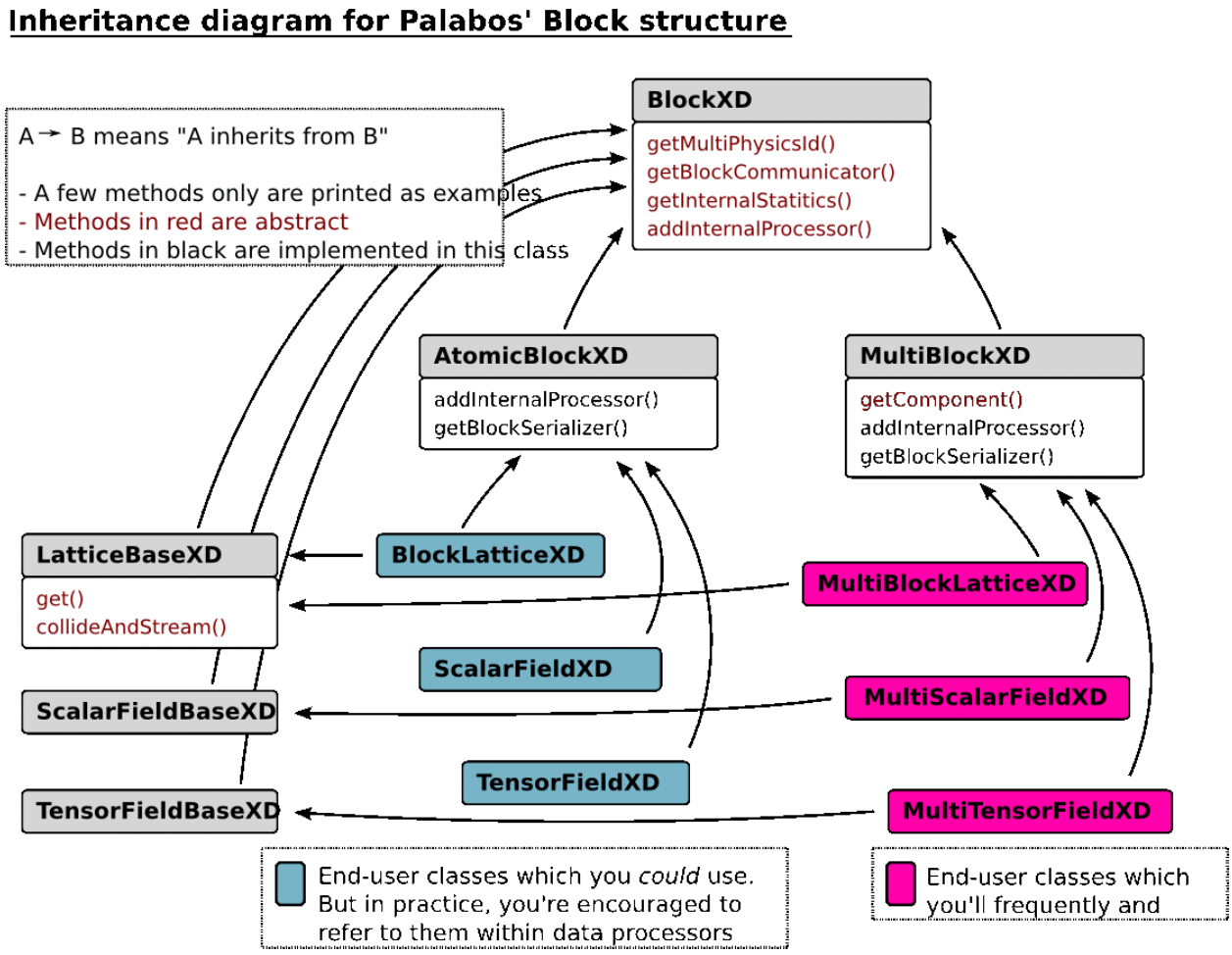


Figure 1: blockInheritance

are implemented as a multi-block or an atomic-block. In the same way, all multi-blocks have a method `getComponent()`, no matter if they are of type block-lattice, scalar-field, or tensor-field. The situation shown on this figure is referred to as "multiple inheritance", because end-user classes inherit from BlockXD in two ways: once through the atomic- vs. multi-block path, and once through the block-lattice vs. scalar- vs. tensor-field path. Note that multiple inheritance is often considered as bad practice because it can lead to error-prone code; in the present case however, we've had positive experiences so far with the inheritance diagram shown above, because it is easy to use and represents the two facets of a `BlockXD` in a natural way.

Lattice descriptors

All `BlockXD` constructs are templatized with respect to the underlying data type. In practice, this feature is used to switch between single-precision and double-precision arithmetics by changing just one word in the end-user program:

```
// Construct a 100x100 scalar-field with double-precision floating point values.
MultiScalarField2D<double> a(100,100);
// Construct a 100x100 scalar-field with single-precision floating point values.
MultiScalarField2D<float> b(100,100);
```

Block-lattices additionally have a template parameter, the lattice descriptor, which specifies a few topological properties of the lattice (the number of particle populations, the discrete velocities, the weights of the directions, and other lattice constants). It is therefore easy to try out different lattices in an application:

```
// Construct a 100x100 block-lattice using the D3Q19 structure.
MultiBlockLattice2D<double, D3Q19Descriptor> lattice1(100,100);
// Construct a 100x100 block-lattice using the D3Q27 structure.
MultiBlockLattice2D<double, D3Q27Descriptor> lattice2(100,100);
```

It is also easy to write a new lattice descriptor (this is currently not documented, but you can check out the files in the directory `src/latticeBoltzmann/nearestNeighborLattices2D.h` to see how it works). This is extremely useful, because it means that you don't need to re-write the lengthy code parts for the implementation of a BlockLatticeXD when you switch to a new type of lattice. This argument is part of a general concept described in the section Non-intrusive program development with Palabos.

The dynamics classes

During a time iteration of a lattice Boltzmann simulation, all cells of a block-lattice perform a local collision step, followed by a streaming step. The streaming step is hardcoded and can be influenced only by defining the discrete velocities in a lattice descriptor. The collision step on the other hand can be fully customized and can be different from one cell to another. In this way, the nature of the physics simulated on the lattice can be adjusted locally, and specific areas such as the boundaries can get an individual treatment.

Each cell of a block-lattice holds, additionally to the variables of the simulation, a pointer to an object of type `Dynamics`. This object provides most notably an implementation of the collision step. Additionally, it provides a means of computing the macroscopic variables, information for rescaling variables between grids of different size, and more model-dependent information. As a result, the list of methods declared in the class `Dynamics` is quite long, and it can seem tedious to inherit from this class. For this reason, Palabos provides a long list of classes inheriting from `Dynamics`, in which many methods are pre-defined for a specific purpose. Defining a new dynamics class therefore essentially consists in choosing the right place in the inheritance hierarchy and then overriding two or three methods. For the sake of illustration, a part of this inheritance diagram is printed in the following picture:

To learn how to define a new dynamics class, it is easiest to look at one of the classes defined in Palabos. For example, the BGK dynamics is defined in the file `src/basicDynamics/isoThermalDynamics.hh`. A good

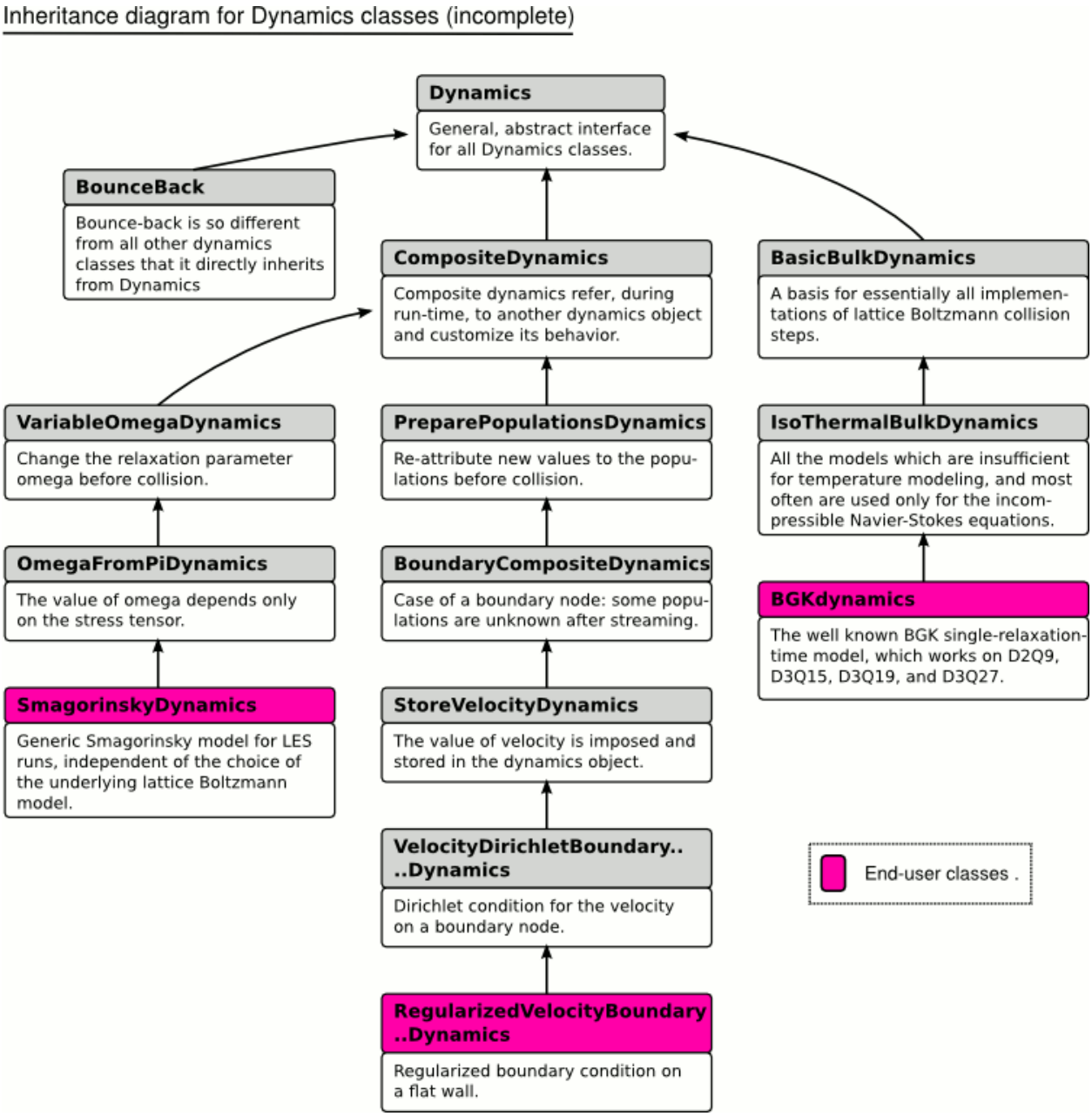


Figure 2: dynamicsInheritance

example for composite dynamics (a class which modifies the behavior of another, existing dynamics class) is the Smagorinsky dynamics, defined in `src/complexDynamics/smagorinskyDynamics.hh`.

For the implementation of the collision, the dynamics object gets a reference to a single cell; the collision step is therefore necessarily local. Non-local ingredients of a simulation are implemented with data processors, as shown in the next section.

Like the block-lattice, a dynamics object is dependent on two template parameters, one for the floating-point representation, and one for the lattice descriptor. By using the information provided in the lattice descriptor, the collision step should be written in a generic, lattice-independent way. There is obviously an efficiency trade-off in writing the algorithms in a generic way, because it is possible to formulate optimizations for specific lattices which the compiler fails to find. This problem can be circumvented by using template specializations for a given lattice. As an example, take again the implementation of the class `BGKdynamics`. The implementation of collision step refers to a generic object `dynamicsTemplates`, which is defined in the file `src/latticeBoltzmann/dynamicsTemplates.h`. Efficient specializations of this class for various 2D and 3D lattices are found in the files `dynamicsTemplates2D.h` and `dynamicsTemplates3D.h`.

Data processors

Data processors define an operation to be performed on the entire domain, or on parts of a block. On a block-lattice, they are used to implement all operation which cannot be formulated in terms of dynamics objects. These consist most notably of non-local operations, such as the evaluation of finite difference stencils for the implementation of boundary conditions. On scalar-fields and tensor-fields, data processors provide the only (sufficiently efficient and parallelizable) way to perform an operation on a spatially extended domain.

Furthermore, data processors have the ability to exchange information between several blocks, either of the same type or of different type (example: coupling between a block-lattice and scalar-field). This is for example used for the implementation of physical couplings (multi-component fluids, thermal fluids with Boussinesq approximation), for the setup of initial conditions (initialization of the velocity from the values in a vector-field), or the execution of classical array-based operations (element-wise addition of two scalar-fields).

Finally, data processors are used to perform reduction operations, on the entire block or on sub-domains. Examples range from the computation of the average kinetic energy in a simulation to the computation of the drag force acting on an obstacle.

Two different point of views are adopted for the definition and for the application of a data processor. At the application level, the user specifies an area (which can be rectangular or irregular) of a given block, on which to execute the data processor. If the block has internally a multi-block structure, the data processor is subdivided into several more specific data processors, one for each atomic-block inside the multi-block which intersects with the specified area. At the execution level, a data processor therefore always acts on an atomic-block, on an area which was previously determined by intersecting the original area with the domain of the atomic-block.

It should be mentioned that while the raw data processors are somewhat awkward to use, you are likely to never be in contact with them. Instead, Palabos offers a simplified interface through so-called data-processing functionals, which hide technical details and let you concentrate on the essential parts. The rest of the user guides concentrates exclusively on these functionals, which will be called data processors for short.

In the end, it is quite easy to define new data processors. All you need to do is write a function which receives an atomic-block and the coordinates of a sub-domain as parameters, and executes an algorithm on this sub-domain. All complex operations, like the sub-division of the operations in presence of a multi-block, or the parallelization of the code, are automatic.

An educative example of a data processor is found in the example file `examples/codeByTopics/couplings`. It shows how to initialize a block-lattice with a velocity from a vector field by writing a data processor for block-lattice vs. 2D tensor-field coupling.

More definitions of data-processors acting on block-lattices can be found in the files `src/simulationSetup/latticeInitialize` and `.hh`, and data-processors acting on scalar- or tensor-fields are defined in the files `src/simulationSetup/dataFieldInitial` and `.hh`. Examples for the evaluation of reduction operations are provided in the files `src/core/dataAnalysisXD.h` and `.hh`.

All these data-processors are wrapped up in convenience functions, which are summarized in the Appendix: partial function/class reference.

Implemented fluid models

Non-thermal Navier-Stokes equations

There exists a bunch of LB models which solve the dynamics of a compressible fluid but in which, for various reasons, the modeling of the energy equation is wrong. These models are sometimes used for small-Mach-number compressible flows with negligible temperature variations, but most often, they are simply used to solve the incompressible Navier-Stokes equations. In this case, the Mach number is kept low to damp unwanted compressibility effects, and the model is used as a so-called quasi-compressible solver.

Unless it is specified differently, all presented models can be used with the D2Q9, D3Q15, D3Q19, and D3Q27 lattices, using the corresponding descriptors `DdQqDescriptor`.

If you use a model with external force term, replace the lattice descriptor `DdQqDescriptor` by `ForcedDdQqDescriptor`.

It should be mentioned that among Palabos developers an uncertainty reigns over the entropic model (it might be fully valid on D2Q9 and D3Q27 only).

Single-relaxation-time BGK

This is so to say the mother of all LB models: it is widely used, simple to implement and surprisingly versatile.

- Dynamics for plain BGK: BGKdynamics
- Dynamics with external force: GuoExternalForceBGKdynamics, ShanChenExternalForceBGKdynamics, HeExternalForceBGKdynamics
- Reference: chen-98
- Example: `codesByTopic/navierStokesModels`

Incompressible BGK

In standard BGK, the equilibrium term is multiplied by the fluid density. In the so-called incompressible model, the density only appears in the constant term (the one without velocity), and the velocity is defined to be proportional to momentum (it is not divided by rho). This model is computationally a little bit cheaper than BGK, because it avoids the division by rho for the computation of the velocity. Furthermore, the compressibility error of a simulation as compared to an exact solution of the incompressible Navier-Stokes equations is reduced for steady flows, as it scales like the cube of the Mach number, as opposed to the square Mach number in standard BGK. In unsteady flows, the error remains proportional to the square-Mach-number, though.

- Dynamics: IncBGKdynamics
- Reference: *missing*
- Example: `codesByTopic/navierStokesModels`

Regularized BGK

Before the collision step, the particle populations are projected onto a Hermite base with polynomials up to second order. While this keeps the dynamics of the Navier-Stokes equations intact, it improves the numerical stability by imposing additional stability on the particle populations.

There exists a generic version for the regularized collision of type `CompositeDynamics`, which can be used to regularized any LB model. Additionally, a specific class is implemented for the regularized BGK model, for efficiency reasons.

- Generic Dynamics: RLBdynamics
- Dynamics for BGK: RegularizedBGKdynamics
- Reference: latt-06
- Example: `codesByTopic/navierStokesModels`

Multiple-relaxation-time with linear stability optimization

In multiple-relaxation-time models, the linear collision operator is recast into the space of velocity moments, and their corresponding relaxation parameters are individually adjusted, either to modify the physics of the model, or to improve numerical stability. In the model implemented here, specific parameters are used for the non-physical ghost modes to improve the stability.

- Dynamics: MRTdynamics
- Descriptors: MRTD2Q9Descriptor, MRTD3Q19Descriptor
- Reference: dhumieres-02
- Example: `codesByTopic/navierStokesModels`

Entropic model

In the absence of boundaries, this model is unconditionally stable because it guarantees the existence of an H-function which, just like in nature, is non-decreasing.

- Dynamics for plain model: EntropicDynamics
- Dynamics with external force: ForcedEntropicDynamics
- Reference: *missing*
- Example: `codesByTopic/navierStokesModels`

Thermal flows with Boussinesq approximation

In the Boussinesq approximation, a fluid is modeled as incompressible, and the influence of the temperature is visible only through a body-force term, representing buoyancy effects. The temperature field obeys an advection-diffusion equation.

The coupling between the temperature field and the fluid solver is implemented by means of a data processor. For the fluid, you can use any of the non-thermal Navier-Stokes models with external force term.

Advection-diffusion equation for the temperature

- Dynamics: AdvectionDiffusionBGKdynamics
- Descriptors: AdvectionDiffusionD2Q5Descriptor, AdvectionDiffusionD3Q7Descriptor
- Reference: guo-02c

Coupling between fluid and temperature

- Data Processor: BoussinesqThermalProcessorXD
- Examples: `showCases/boussinesqThermal2D` and `showCases/boussinesqThermal3D`
- Reference: guo-02c

Multi-component and multi-phase fluids

The widely used Shan/Chen models for immiscible multi-component fluids and for single-component multi-phase fluids are implemented in Palabos, in 2D and in 3D. Note that a lot of information about the theory and practical aspects of these models can be found in the book [Sukop-and-Thorne]1.

In both models, an inter-particle force term needs to be computed which is not entirely local. This force term is therefore computed by a data processor. This data processor needs to know the density and the velocity on a given lattice node (in the multi-component model, these macroscopic variables must be known for all components). To avoid computing these variables twice, as they need also to be known for the subsequent, normal collision step, they are written by the data processor to an external scalar of the cell, and then re-used by the dynamics object for the collision. It is therefore necessary to use a lattice descriptor which provides space for these external scalars, such as `ShanChenD2Q9Descriptor`, or `ForcedShanChenD2Q9Descriptor` if additionally to the inter-particle potential there is an external body force. In 3D, the same descriptors are available for the D3Q19 lattice (and other descriptors are easy to formulate). To be sure to read the macroscopic variables from external scalars during the collision step, use the classes `ExternalMomentBGKdynamics` or `ExternalMomentRegularizedBGKdynamics` for the fluid model.

Note that in both models, the effect of the external body force term is automatically accounted for if an external force is defined in the lattice descriptor. The force term is then added by the Shan/Chen data processor by means of a momentum correction to the flow velocity. Therefore, you don't need to use a specific collision model with force term like `GuoExternalForceBGKdynamics`. A plain BGK dynamics, or regularized BGK dynamics, is fine.

Palabos also implements the 3D He/Lee multi-phase fluid model, based on a two populations. This model enforces incompressibility by solving a pressure evolution equation. While it is more complicated than the Shan/Chen model from an algorithmic standpoint, it has the ability to simulate larger density and viscosity differences between the two phases, and eliminates spurious velocities to a certain extent.

Multi-component Shan/Chen model

In this model, each component is simulated on a separate block-lattice, and the components are coupled through a Shan/Chen data processor. You can couple as many components as you wish.

- Data Processor: ShanChenMultiComponentProcessorXD
- Examples: `showCases/multiComponent2d` and `showCases/multiComponent3d`
- Reference: shan-chen-93

Single-component Shan/Chen multi-phase model

In this model, the Shan/Chen data processor takes an additional argument which defines the shape of the inter-particle potential. The following potentials are defined (see the file `src/multiPhysics/interparticlePotential.h` for details):

- PsiIsRho() $\Psi = \rho$
- PsiShanChen93(rho0) $\Psi = \rho_0 \left(1 - e^{-\rho/\rho_0}\right)$
- PsiShanChen94(Psi0,rho0) $\Psi = \Psi_0 e^{-\rho_0/\rho}$
- PsiQian95(rho0,g) $\Psi = g\rho_0^2\rho^2/(2(\rho_0+\rho)^2)$

Please remark that in this model, the fluid density must be carefully adapted to the amplitude of the interaction force in order to enter the critical regime in which phase separation occurs.

- Data Processor: ShanChenSingleComponentProcessorXD
- Examples: `codesByTopic/shanChenMultiPhase`
- Reference: han-chen-93

Free-Surface Flow

In a free-surface flow, the viscosity of one of the two phases is virtually infinite. This phase is therefore neglected, and a surface-tracking approach is applied to the remaining phase. Palabos uses a volume-of-fluid like approach to trace the free surface.

As an example, a dam break application is provided in `examples/showCases/breakingDam3d`.

Large eddy simulations

Static Smagorinsky model

In the Smagorinsky model, it is assumed that the subgrid scales have the effect of a viscosity correction which is proportional to the norm of the strain-rte tensor at the level of the filtered scales, *:math: nu = nu_0 + nu_T*. The formula for the turbulent viscosity correction nuT is

$$\nu_T = C^2 \left|S\right|$$

where C is the Smagorinsky constant, and the tensor-norm of the strain rate is defined as $|S| = \sqrt{S : S}$ (attention: there is no factor 1/2 inside the square-root, as it can be found in other definitions). The value of the Smagorinsky constant depends on the physics of the problem, and usually varies between 0.1 and 0.2 far from boundaries. This model is called static because the value of the Smagorinsky constant is imposed and does not change in time.

In the Palabos implementation, the strain-rate is computed from the stress tensor Π. It is remarked that the relationship between S and Π contains depends on the relaxation time τ, and therefore on the viscosity nu. The formula for the turbulent viscosity nuT is therefore implicit, but is of second-order only and can therefore be solved explicitly.

Given that the stress tensor Π can be computed from local variables on a cell, the Smagorinsky model is entirely local and is implemented in Palabos through a dynamics class. As so often, there are several implementations for this model. The generic one, based on composite-dynamics objects, adds a Smagorinsky viscosity correction to any existing dynamics during runtime. The specific ones are written explicitly for the BGK and for the regularized BGK models, and are therefore somewhat more efficient. In each case, the value of the viscosity is modified before the execution of the usual collision step. Therefore, if during a simulation you access the relaxation parameter ω at a cell, for example through a function call `lattice.get(x,y).getOmega()`, you get the relaxation parameter related to the effective viscosity ν, and not the "molecular-scale viscosity" ν_0.

The value of the Smagorinsky constant must be provided to these dynamics classes as a constructor argument. It is important to note that the dynamics object of each cell can have a different value of the Smagorinsky constant: this "constant" can be space dependent. This is useful for example to model boundary layers, where the Smagorinsky constant drops to zero. The most convenient way to create a simulation with space-dependent Smagorinsky constant is to assign the Smagorinsky dynamics to each cell of the lattice from within a data processor which is aware of the desired value of the C as a function of space.

- Generic Dynamics: SmagorinskyDynamics
- Dynamics for BGK: SmagorinskyBGKdynamics
- Dynamics for Regularized BGK: SmagorinskyRegularizedDynamics

- Reference: missing
- Example: `codesByTopic/smagorinskyModel`

Non-Newtonian fluids

Carreau model

In the Carreau model, the value of the viscosity is adjusted, just like in the Smagorinsky model, depending on the value of the local strain-rate. The constitutive equation is given by

$\nu = \nu_0 * (1 + (\lambda * |S|)^2)^{(n-1)/2}$

where λ and n are given real parameters. As for the Smagorinsky model, the strain rate S is computed from the stress tensor Π, and the resulting equation is implicit because the S vs. Π relation is dependent on the relaxation time τ. In the present case however, there is no explicit solution, and the equation is solved at each time step through a fixed-point iteration (which in this case converges very quickly and is therefore more efficient than a gradient-based solution method).

In this implementation, the parameters ν_0, λ and n cannot be space-dependent, and they are set through a function call to the global singleton CarreauParameters. for example:

```
global::CarreauParameters().setNu0(nu0);
global::CarreauParameters().setLambda(1.);
global::CarreauParameters().setExponent(0.3);
```

- Dynamics: CarreauDynamics
- Reference: missing
- Example: `codesByTopic/nonNewtonian`

---

1 [Sukop-and-Thorne] Michael C. Sukop and Daniel T. Thorne (2006), Lattice Boltzmann Modeling; an Introduction for Geoscientists and Engineers. Springer-Verlag Berlin/Heidelberg.

Setting up a problem

Attributing dynamics objects

When constructing a new block-lattice, you must decide what type of collision is going to be executed on each of its cells, by assigning them a dynamics object. To avoid bugs related to cells without dynamics objects, the constructor of a block-lattice assigns a default-value for the dynamics to all cells, the so-called background dynamics:

```
// Construct a new block-lattice with a background-dynamics of type BGK.
MultiBlockLattice3D<T,DESCRIPTOR> lattice( nx, ny, nz,
                                           new BGKdynamics<T,DESCRIPTOR>(omega) );
```

After this, the dynamics of each cell can be redefined in order to adjust the behavior of the cells locally. The background dynamics differs from usual dynamics objects in an important way. To obtain memory savings, the constructor of the block-lattice creates only one instance of the dynamics object, and all cells refer to the same instance. If you modify a dynamics object on one cell, it changes everywhere. As an example, consider the relaxation parameter `omega`, which is stored in the dynamics, not in the cell. A function call like

```
lattice.get(0,0,0).getDynamics().setOmega(newOmega);
```

would affect the value of the relaxation parameter on each cell of the lattice. Note that this particular behavior of the background dynamics, having several cells referring to the same object, cannot be reproduced by other means than constructing a new block-lattice. All other functions which override the background-dynamics on given cells with a new dynamics object are defined in such a way as to create an independent copy of the dynamics object for each cell.

If the reference semantics of the background dynamics is contrary to your needs, you can re-define all dynamics objects right after constructing a block-lattice, to guarantee that all cells have an independent copy:

```
// Override background-dynamics to guarantee and independent per-cell
//   copy of the dynamics object.
defineDynamics( lattice, lattice.getBoundingBox(),
                new BGKdynamics<T,DESCRIPTOR> );
```

There exist different variants of the function `defineDynamics` which you can use to adjust the dynamics of a group of cells (their syntax can be found in the Appendix `Operations on the block-lattice`):

One-cell version: - Assign a new dynamics to just one cell.

- Example: `tutorial/tutorial2/tutorial2_3.cpp`

BoxXD version: - Assign a new dynamics to all cells within a rectangular domain.

- Example: `showCases/multiComponent2d/rayleighTaylor2D.cpp`, or the `3d` example.

DotListXD version: - Assign a new dynamics to several cells, listed individually in a dot-list structured.

- Example: `codesByTopic/dotList/cylinder2d.cpp`

Domain-functional version: - Provide an analytical function which indicates the coordinates of cells which get a new dynamics object.

- Example: `showCases/cylinder2d/cylinder2d.cpp`

Bool-mask version: - Specify the location of cells which get a new dynamics object through a Boolean mask, represented through a scalar-field.

- Example: `codesByTopic/io/loadGeometry.cpp`

Initial values of density and velocity

It is quite common to assign an initial value of velocity and density to a lattice by initializing all cells to an equilibrium distribution with the chosen density and velocity value. While this approach is not sufficient for time-dependent benchmark problems which depend critically on the initial state, it is most often sufficient to get a simulation started with reasonable initial values. The function `initializeAtEquilibrium` (see Appendix `Operations on the block-lattice`) is provided to initialize the cells within a rectangular-shaped domain at an equilibrium distribution. It comes in two flavors: one with a constant density and velocity within the domain (see the example `examples/showCases/cavity2d` or `3d`), and one with a space-dependent value of these macroscopic values, specified through a user-defined function (see the example `examples/showCases/poiseuille`).

To initialize cells in a different way, it is simplest to apply a custom operator to the cells with the help of one-cell functionals and indexed one-cell functionals introduced in Section `Convenience wrappers for local operations`.

Defining boundary conditions

Overview

The library Palabos currently implements a bunch of different boundary conditions for straight, grid-aligned boundaries. For general boundaries, the available options include stair-cased walls through bounce-back nodes, and higher-order accurate curved boundaries.

IMPORTANT: Curved boundaries are available as of version 1.0, and are part of a fully-featured parallel stack, including parallel voxelization and I/O extensions. Curved boundaries are not yet documented. You can however find a full-featured example in `showCases/aneurysm`.

Grid-aligned boundaries

The class OnLatticeBoundaryConditionXD

Some of the implemented boundary conditions are local (and are therefore implemented as dynamics classes), some are non-local (and are therefore implemented as data processors), and some have both local and non-local components. To offer a uniform interface to the various possibilities, Palabos offers the interface OnLatticeBoundaryConditionXD which is responsible for instantiating dynamics objects, adding data processors, or both, depending on the chosen boundary condition. The following line for example is used to chose a Zou/He boundary condition:

```
OnLatticeBoundaryCondition3D<T,DESCRIPTOR>* boundaryCondition =
        createZouHeBoundaryCondition3D<T,DESCRIPTOR>();
```

The four possibilities currently offered are (see palabos.org-bc for details):

- Regularized BC: `createLocalBoundaryConditionXD`
- Skordos BC: `createInterpBoundaryConditionXD`
- Zou/He BC: `createZouHeBoundaryConditionXD`
- Inamuro BC: `createInamuroBoundaryConditionXD`

With each of these boundary conditions, it is possible to implement velocity Dirichlet boundaries (all velocity components have an imposed value) and pressure boundaries (the pressure is imposed, and the tangential velocity components are zero). Furthermore, various types of Neumann boundary conditions are proposed. In these case, a zero-gradient for a given variable is imposed with first-order accuracy by copying the value of this variable from a neighboring cell.

Boundaries on a regular block domain

The usage of these boundary conditions is somewhat tricky, because Palabos needs to know if a given wall is a straight wall or a corner (2D), or a flat wall, an edge or a corner (3D), and it needs to know the wall orientation. To simplify this, all `OnLatticeBoundaryConditionXD` objects have a method `setVelocityConditionOnBlockBoundaries` and a method `setPressureConditionOnBlockBoundaries`, to set up boundaries which are located on a rectangular domain. If all nodes on the boundary of a given 2D box `boundaryBox` implement a Dirichlet condition, use the following command:

```
Box2D boundaryBox(x0,x1, y0,y1);
boundaryCondition->setVelocityConditionOnBlockBoundaries (
                            lattice, boundaryBox, locationOfBoundaryNodes );
```

The first argument indicates the boxed shape on which the boundary is located, and the second argument specifies on which node a velocity condition is to be instantiated. If all boundary nodes are located on the exterior bounding box of the lattice, the above command simply becomes:

```
boundaryCondition->setVelocityConditionOnBlockBoundaries(lattice, locationOfBoundaryNodes);
```

Finally, if simply all nodes of the exterior bounding box shall implement a velocity condition, this simplifies to:

```
boundaryCondition->setVelocityConditionOnBlockBoundaries(lattice);
```

Now, suppose that the upper and lower walls, including corners, in a `nx` -by- `ny` lattice implement a free-slip condition (zero-gradient for tangential velocity components), the left wall a Dirichlet condition, and the right wall and outflow condition (zero-gradient for all velocity components). This is achieved by adding one more parameter to the function call:

```
Box2D inlet(0, 0, 2, ny-2);
Box2D outlet(nx-1, nx-1, 2, ny-2);
Box2D bottomWall(0, nx-1, 0, 0);
Box2D topWall(0, nx-1, ny-1, ny-1);

boundaryCondition->setVelocityConditionOnBlockBoundaries ( lattice, inlet );
boundaryCondition->setVelocityConditionOnBlockBoundaries (
                              lattice, outlet, boundary::outflow );
boundaryCondition->setVelocityConditionOnBlockBoundaries (
                              lattice, bottomWall, boundary::freeslip );
boundaryCondition->setVelocityConditionOnBlockBoundaries (
                              lattice, topWall, boundary::freeslip );
```

Remember that if the boundary is not located on the outer bounds of the lattice, you must use an additional Box3D argument to say where the boundaries are located, additionally to saying which boundary is going to be added. This is necessary, because Palabos needs to know the orientation and the nature (flat wall, corner, etc.) of the boundary nodes.

Zero-gradient conditions

The optional arguments like `boundary::outflow` are of type `boundary::BcType`, and can have the following values for velocity boundaries:

boundary::dirichlet (default value)	Dirichlet condition: imposed velocity value.
boundary::outflow or boundary::neumann	Zero-gradient for all velocity components.
boundary::freeslip	Zero-gradient for tangential velocity components, zero value for the others.
boundary::normalOutflow	Zero-gradient for normal velocity components, zero value for the others.

For pressure boundaries, you have the following choice:

boundary::dirichlet (default value)	Dirichlet condition: imposed pressure value, zero value for the tangential velocity components.
boundary::neumann	Zero-gradient for the pressure, zero value for the tangential velocity components.

Boundary conditions cannot be overriden: It is not possible to override the type of a boundary. Once a boundary node is a Dirichlet velocity node, it stays a Dirichlet velocity node. The result of re-defining it as, say, a pressure boundary node, is undefined. Therefore, boundary conditions must be defined carefully, and piece-wise, as shown in the example above. On the other hand, the value imposed on the boundary (i.e. the velocity value on a Dirichlet velocity boundary) can be changed as often as needed.

Setting the velocity or pressure value on Dirichlet boundaries

A constant velocity value is imposed with the function `setBoundaryVelocity`. The following line sets the velocity values to zero on all nodes of a 2D domain which have previously been defined to be Dirichlet velocity nodes:

```
setBoundaryVelocity(lattice, lattice.getBoundingBox(), Array<T,2>(0.,0.) );
```

On all other nodes, this command has no effect. To set a constant pressure value (or equivalently, density value), use the command `setBoundaryDensity`:

```
setBoundaryDensity(lattice, lattice.getBoundingBox(), 1. );
```

A non-constant, space dependent velocity resp. pressure profile can be defined by replacing the velocity resp. pressure argument by either a function or by a function object (an instance of a class which overrides the function call operator) which yields a value of the velocity resp. density as a function of space position. An example is provided in the file `examples/showCases/poiseuille/poiseuille.cpp`.

Interior and exterior boundaries

Sometimes, the boundary is not located on the surface of a regular, block-shaped domain. In this case, the functions like `setVelocityConditionOnBlockBoundaries` are useless, and the boundary must be manually constructed. As an example, here's how to construct manually a free-slip condition along the top-wall, including the corner nodes:

```
// The automatic creation of a top-wall ...
Box2D topWall(0, nx-1, ny-1, ny-1);
boundaryCondition->setVelocityConditionOnBlockBoundaries (
                                topWall, lattice, boundary::freeslip );

// ... can be replaced by a manual construction as follows:
Box2D topLid(1, nx-2, ny-1, ny-1);
boundaryCondition->addVelocityBoundary1P( topLid, lattice, boundary::freeslip );
boundaryCondition->addExternalVelocityCornerNP( 0, ny-1, lattice, boundary::freeslip );
boundaryCondition->addExternalVelocityCornerPP( nx-1, ny-1, lattice, boundary::freeslip );
```

A distinction is made between external and internal corners, depending on whether the corner is convex or concave. On the following example geometry, you'll find five external and one internal corner:

The extensions like `1P`, '`NP`, and `PP` at the end of the methods of the boundary-condition object are used to indicate the orientation of the wall normal, pointing outside the fluid domain, as shown on the figure above. On a straight wall, the code `1P` means: "the wall normal points into positive y-direction". Likewise, the inlet would be labeled with the code `0N` as in "negative x-direction". On a corner, the code `NP` means "negative x-direction and positive y-direction". It is important to mention that this wall normal is purely geometrical and does not depend on whether a given wall has the function of an inlet or an outlet. In both cases, the wall normal points away from the fluid, into the non-fluid area.

In 3D, you'll use the following function for plane walls:

```
addVelocityBoundaryDO
```

where the direction D can be 0 for x, 1 for y, or 2 for z, and the orientation `O` has the value `P` for positive and `N` for negative. Edges can be internal or external. For example:

```
addInternalVelocityEdge0NP
addExternalVelocityEdge1PN
```

where the first digit of the code indicates the axis to which the edge is parallel, and the two subsequent digits indicate the orientation of the edge inside the plane normal to the edge, in the same way as the corner nodes

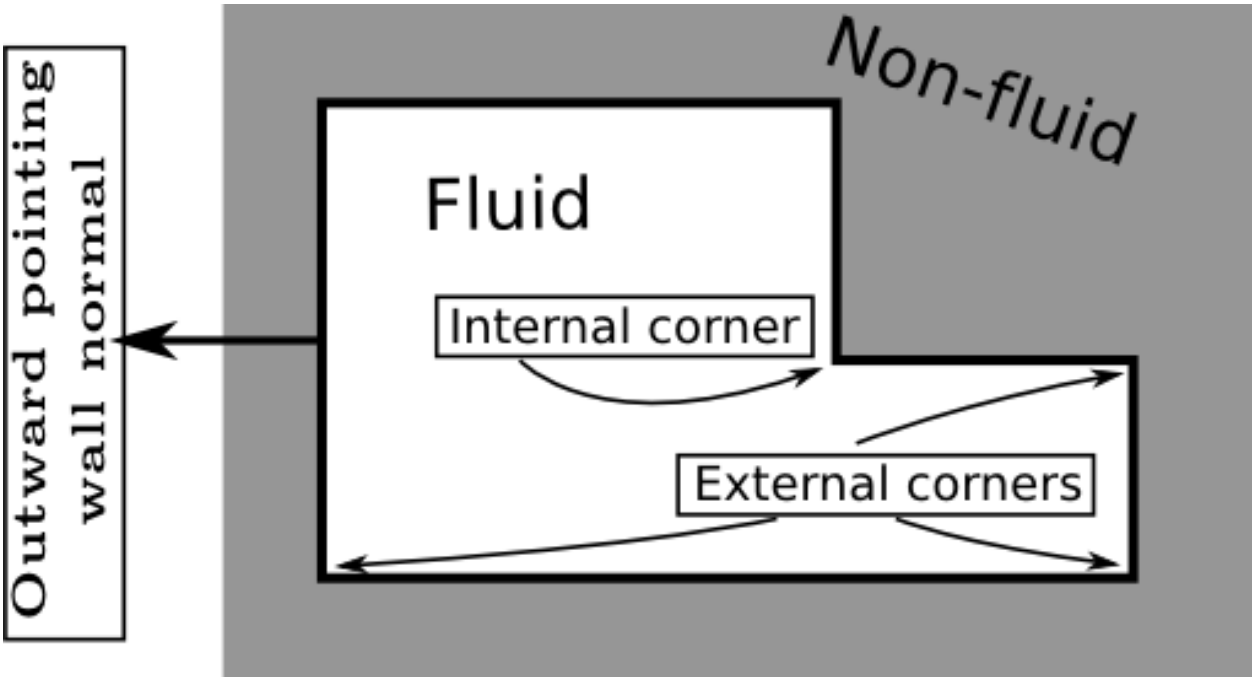


Figure 3: boundaries

in 2D. The axes are counted periodically: `0NP` means x-plane, negative y-value, and positive z-value, whereas `1PN` means y-plane, positive z-value, and negative x-value. Finally, 3D corners are constructed in the same way as in 2D, through a function call like the following:

```
addInternalVelocityCornerPNP
addExternalVelocityCornerNNN
```

Periodic boundaries

The concept of periodicity in Palabos is different from the approach chosen for the other types of boundary conditions. Periodicity can only be implemented on opposite, outer boundaries of an atomic-block or a multi-block. This behavior works also if the multi-block has a sparse memory implementation. In this case, all fluid nodes which are in contact with an outer boundary of the multi-block can be periodic, if the corresponding node on the opposite wall is a fluid node.

In the case of an atomic-block (remember that this case is uninteresting, because you should always work with multi-blocks anyway), periodicity is simply a property of the streaming operator. It has no effect for scalar-fields and tensor-fields. In the case of a multi-block however, periodicity can also affect scalar-fields and tensor-fields. Being periodic in this case means that if you define a non-local data processors which accesses nearest neighbor nodes, the value of these neighbor nodes is determined by periodicity along an outer boundary of the multi-block.

By default, all blocks are non-periodic: as mentioned previously, the behavior of outer boundary nodes defaults to one of the versions of bounce-back algorithm. To turn on periodicity along, say, the x-direction, write a command like

```
lattice.periodicity.toggle(0, true);  // Periodic block-lattice.
scalarField.periodicity.toggle(0, true);  // Periodic scalar-field.
```

You can also define all boundaries to be periodic through a single command:

```
lattice.periodicity.toggleAll(true);
```

Please note that the periodicity or non-periodicity of a block should be defined as soon as possible after the creation of the block, because Palabos needs to know if boundaries are periodic or not when adding or executing data processors.

As a last remark, you should be aware that boundaries of Neumann type (outflow, free-slip, adiabatic thermal wall, etc.) should never be defined to be periodic. This wouldn't make sense anyway, and it leads to an undefined behavior (aka the program crashes).

Bounce-back

In Palabos, all outer boundaries of a lattice which are not periodic automatically implement a version of the bounce-back boundary algorithm, according to the following algorithm. In pre-collision state, all unknown populations are assigned the post-collision value on the same node at the end of the previous iteration, from the opposite location. We will call this algorithm `Bounce-Back 1`.

A bounce-back condition can also be used elsewhere in the domain, by explicitly assigning a dynamics object of type `BounceBack<T,Descriptor>` to chosen cells. On these newly assigned nodes, the collision step is replaced by a bounce-back procedure, during which each population is replaced by the population corresponding to the opposite direction. Such a node can not be considered any more as a fluid node. Computing velocity-moments of the populations on a bounce-back node yields for example arbitrary results, because some of the populations (those which are not incoming from the fluid) have arbitrary values. Consequently, you cannot compute macroscopic quantities like density and velocity on these nodes. Instead, these nodes should be considered as pure algorithmic entities which have the purpose to re-inject into the fluid all populations that leave the domain. Consider fluid nodes adjacent to this type of bounce-back cells. If you think about it, you'll realize that these nodes behave exactly as if they were cells of type `Bounce-Back 1`, with a small difference: unknown populations at time `t` get the post-collision value from opposite populations from time `t-2` (and not time `t-1` as in `Bounce-Back 1`. We will refer to this version of bounce-back as `Bounce-Back 2`.

The effect of both versions of bounce-back is to produce a no-slip wall located half-way between nodes. In the case of `Bounce-Back 1`, this is half a cell-width beyond the bounce-back fluid cell, whereas in the case of `Bounce-Back 2` it is half-way between the last fluid cell and the non-fluid bounce-back node. Obviously, `Bounce-Back 2` wastes numerical precision over `Bounce-Back 1` by leaping over a time step, and therefore leads to a lower-order accuracy in time of the numerical method. Furthermore, both versions of bounce-back represent the wall location through a staircase approximation, which is low-order accurate ("Order-h") in space. On the other hand, `Bounce-Back 2` offers a huge advantage: its implementation is independent of the wall orientation. If you decide that a node is a bounce-back node, you simply say so; no need to know if it is a corner, a left wall, or a right wall. Constructing a geometry is then as easy as filling areas with bounce-back nodes. In many cases, this ease of use makes up, to a large extent, for the low-order accuracy of the wall representation. When setting up a new simulation, it is always good to try `Bounce-Back 2` as a first attempt, as the quality of the obtained results is often surprising. In highly complex geometry such as porous media, bounce-back is often even considered superior to other approaches.

Bounce-back domain from analytical description

The *Rule 0* in Palabos can be stated often enough: don't write loops over space in end-user code. This also true, of course, when assigning bounce-back dynamics to lattice cells. To guarantee reasonable performance, this is task is performed inside a data processor, or even better, by using the wrapper function `defineDynamics`. This process is illustrated in the example program `examples/codesByTopic/bounceBack/instantiateCylinder.cpp`. First, a class is written which defines the location of the bounce-back nodes as a function of the cell coordinates iX and iY. In the present case, the nodes are located inside a 2D circular domain:

```
template<typename T>
class CylinderShapeDomain2D : public DomainFunctional2D {
public:
    CylinderShapeDomain2D(plint cx_, plint cy_, plint radius)
        : cx(cx_),
          cy(cy_),
          radiusSqr(util::sqr(radius))
    { }
    // The function-call operator is overridden to specify the location
    //   of bounce-back nodes.
    virtual bool operator() (plint iX, plint iY) const {
        return util::sqr(iX-cx) + util::sqr(iY-cy) <= radiusSqr;
    }
    virtual CylinderShapeDomain2D<T>* clone() const {
        return new CylinderShapeDomain2D<T>(*this);
    }
private:
    plint cx, cy;
    plint radiusSqr;
};
```

An instance of this class is then provided as an argument to `defineDynamics`:

```
defineDynamics( lattice, lattice.getBoundingBox(),
                new CylinderShapeDomain2D<T>(cx, cy, radius),
                new BounceBack<T,DESCRIPTOR> );
```

The second argument is of type Box2D, and it can be used to improve the efficiency of this function call: the bounce-back nodes are attributed on cells on the intersection between this box and the domain specified by the domain-functional.

If your domain is specified as the union of a collection of simpler domains, you can use `defineDynamics` iteratively on each of the simpler domains. If the domain is the intersection or any other geometric operation of simpler domains, you'll need to play with boolean operators inside the definition of your domain-functional.

Bounce-back domain from boolean mask

In this section, it is assumed that the geometry of the domain is prescribed from an external source. This is for example the case when you simulate a porous media and possess data from a scan of the porous material in question. This easiest approach consists in storing this data as an ASCII file which distinguishes fluid nodes from solid nodes by zeros and ones, separated with spaces, tabs, or newline characters. The file represents the content of a regular array and stores only raw data, no information on the matrix dimensions. The data layout must be the same as in Palabos: an increment of the z-coordinate represents a continuous progress in the file (in 3D), followed by the y-coordinate, and then the x-coordinate. This data is simply flushed from the file into a scalar-field. The following code is taken from the example `examples/codesByTopic/io/loadGeometry.cpp`:

```
MultiScalarField2D<T> boolMask(nx, ny);
plb_ifstream ifile("geometry.dat");
ifile >> boolMask;
```

Note that currently, no error-checking is implemented for such I/O operations. It is your responsibility to ensure that the dimensions of the scalar-field correspond to the size of the data in the geometry file. As a next step, the bounce-back nodes are instantiated using one of the versions of the function `defineDynamics`:

```
defineDynamics(lattice, boolMask, new BounceBack<T,DESCRIPTOR>, true);
```

This function is currently only defined for multi-blocks using the same data type (T in this case), which explains why the scalar-field boolMask is of type T and not bool, as one could have expected. The scalar-field is not bool, as one could have expected, but the real type T.

Bounce-back domain from an STL file

It is not always convenient, or even possible by any realistic means, to define the shape of the computational domain analytically. In the general case, a surface mesh describing the computational domain must be provided to Palabos. If you decide to describe the domain boundary through BounceBack nodes, as described in the previous chapters, Palabos will then do nothing else than *voxelize* the domain, i.e. convert the surface description of the domain into a volume description, deciding which fluid node (or voxel) is inside the domain. This voxelization is done internally in Palabos, and is fully parallel: this is a step which usually takes a negligible time compared to the overall simulation time.

The Palabos voxelizer not only decides which voxels are fluid and which ones are bounce-back, it also puts in place a memory-saving scheme in which memory is allocated only in areas covered by fluid.

The surface mesh must be provided to Palabos in a binary or ASCII Stereo-Lithography format (STL). Many surface-mesh formats can be converted to STL using the open-source program Meshlab.

An example of reading an STL file an creating a computational domain surrounded by bounce-back nodes can be found in the file `examples/showCases/aneurysm/aneurysm.cpp`.

Off-lattice boundaries from an STL file

Palabos also offers the possibility to implement general boundaries (inlets, outlets, walls, etc.) with an off-lattice scheme, which provides much better accuracy than the staircase approximation of bounce-back nodes. Again, the domain creation is parallel and extremely fast, and the voxelizer includes a memory-saving approach to the domain generation.

Palabos provides a full-featured off-lattice framework for BGK- or MRT- type fluids, and for the heat equation (advection-diffusion).

A full documentation for working with off-lattice boundaries is not yet available. An example can however be found in the file `examples/showCases/aneurysm/aneurysm.cpp`.

Running a simulation

Time cycles of a Palabos program

The lattice Boltzmann method (or at least, the "classical" lattice Boltzmann method as it is implemented in Palabos) consists of an explicit solver. With a single iteration step, the state of the fluid evolves from a time `t` to the next time `t+dt`. The following discussion is led in the units of the lattice, with `dt=1`. One time iteration of this kind takes the following form in Palabos:

1. To start with, all fluid variables are defined at time `t`. The particle populations are in pre-collision state (also called "incoming populations").
2. The collision operator is applied to all cells. They are now in post-collision state (also called "outgoing variables").
3. The streaming operator is applied to the lattice.
4. All data processors are executed, to perform non-local operations or couplings between lattices.
5. The populations are now again in pre-collision state, but at time `t+1`.

The collision step (Step 2) is executed by invoking the method `lattice.collide()`, and the streaming step (Step 3) is called with the method `lattice.stream()`. These two methods can also be combined into a single call to `lattice.collideAndStream()`. On most hardware platforms, the `collideAndStream()` version is computationally more efficient, because it is executed by traveling through the memory of the lattice only once.

The data processors can be executed manually by calling the method `lattice.executeInternalProcessors()`, as explained in Section `Executing, integrating, and wrapping up data-processing functionals`. This is however rarely done, because the data processors are also executed automatically at the end of the function `stream()` and the function `collideAndStream()`. If you'd like to call `stream()` or `collideAndStream()` without having the data processors executed, for example for debugging a program, you can use the domain-version of these functions, for example `lattice.collideAndStream(lattice.getBoundingBox())`.

In Palabos, data processors are always executed after the collision-streaming cycle. Consequently, non-local operations are always executed after the collision-streaming cycle, at a moment where the populations are in a pre-collision state (incoming populations). This bears no loss of generality, though, because executing an operation before the collision of time `t` is equivalent to executing it after the streaming of time `t-1`, right? The only problem arises with the initial condition, as it is most often desired to have the data processors executed exactly once at the very beginning, right after setting up the initial condition. This is achieved by calling the method `lattice.initialize()` right before starting the first iteration step. This method executes the data processors once, and performs an internal communication step inside the block to guarantee that its internal state is consistent.

At which point do you evaluate data?

To monitor the evolution of a program, it is useful evaluate some hydrodynamic quantities from time to time, such as the average energy:

```
pcout << computeAverageEnergy(lattice) << endl;
```

It is recommended to compute hydrodynamic variables always only when the system is in pre-collision state (incoming populations). While this distinction not really matters for the conserved variables density and velocity (they are equal in the pre- and post-collision state), it is important for the non-conserved variables such as the stress tensor. Non-conserved velocity moments can be related to hydrodynamic variables only when they are computed in the pre-collision state. Note that if you use the method `collideAndStream()`, there is no risk for doing things wrong, because you have no access to post-collision variables anyway.

Normally, the computation of hydrodynamic variables like the average energy in the example above is performed right after the call to the method `collideAndStream()`, because at this point all hydrodynamic variables are well-defined, and correspond to the same moment in time (between collision and streaming the velocity is well defined, but the strain-rate is not). An exception is made for the internal statistics of lattice. Internal statistics are automatically computed without any impact on performance (at least not in serial program; for the parallel case, see the discussion in Section `Controlling the efficiency`), as a side-effect of executing the collision step. They are however evaluated for the fluid variables at time `t`, during the collision-streaming cycle which carries the system from time `t` to time `t+1`. It is therefore usual to access the internal statistics after the call to the method `collideAndStream()`. Computing the average energy as in the example above before collision-and-streaming produces the same result as accessing the average energy from internal statistics, as in the example below, after collision-and-streaming:

```
pcout << getStoredAverageEnergy(lattice) << endl;
```

Other important things to do

Numbers are often difficult to interpret. It is therefore useful to regularly produce images in your program, so that you can monitor the state of simulation, identify problems as soon as possible, and re-run the program

when needed. Section `Producing images in 2D and 3D simulations` explains how to do this.

Finally, it is always good to save the state of a simulation from time to time, in order not to loose everything when the computer crashes, and in order to be able to recover the data if you forgot to produce a crucial output file for post-processing. This is explained in Section `Checkpointing: saving and loading the state of a simulation`.

Input/Output

Output streams: writing to the terminal and into files

In Palabos, never use the usual objects `cout`, `cerr`, or `clog` , or even worse, the ignominious `printf` to print anything to the terminal. These symbols are replaced by the corresponding parallelizable versions `pcout`, `pcerr`, and `pclog`. They have the same syntax and the same behavior as their traditional counterparts, but additionally, they offer the expected behavior in a parallel program: if you print a message from a program parallelized on a hundred cores, the message is printed only once, and not a hundred times. You can use them to print strings, numbers, or even extracts from a lattice, scalar-field, or tensor-field to the screen:

```
pcout << "The average energy is " << computeAverageEnergy(lattice) << endl;
pcout << "And here are some values of the energy, with 10-digit accuracy:" << endl;
pcout << setprecision(10)
      << *computeEnergy(lattice, Box2D(0,10, 0,10)) << endl;
```

In the same way, data can be streamed into an ASCII file (an example for the usage of output file-streams is also provided in the file `examples/codesByTopic/io/useIOstream.cpp`). In this case, the standard `ofstream` is replaced by an `plb_ofstream` as follows:

```
plb_ofstream ofile("energy.dat");
ofile << setprecision(10) << *computeEnergy(lattice) << endl;
```

This can be useful to subsequently load the data into an environment for interactive data analysis like Octave or Matlab:

```
% Analysis of the data in Octave
% Assign manually the dimensions:
nx = 100;
ny = 100;
load energy.dat;
energy = reshape(energy, nx, ny);
imagesc(energy);
```

Input streams: reading large data sets from files

Input files are handled in the same way as output files in the previous section, through objects of type `plb_ifstream`. The file `energy.dat` written in the previous code can for example be streamed back into a matrix through the command:

```
plb_ifstream ifile("energy.dat");
if (ifile.is_open()) {
    // It is your responsibility to create the matrix with the right dimensions
    //    nx-by-ny, compatible with the size of the data in "energy.dat".
    MultiScalarField2D<T> energy(nx,ny);
    ifile >> energy;
}
```

An important point to remember is that the scalar-field into which the data is streamed must be first constructed manually with the right dimensions, in order to avoid memory corruption.

This approach works however only if the data is streamed into a Palabos object. If you wish to stream data from a file into plain C++ data types, for example when accessing user-defined parameters, you may not use the `plb_ifstream` data type as illustrated above, because it leads to an unexpected behavior in parallel programs. In this case, there exist two possible workarounds. The first, which is the recommended approach, is to require that the user creates parameter files in an XML format, which then are parsed using Palabosâ€™s automatic XML parsing facilities, as explained below in sections `input-from-xml`. Doing so has many advantages, as it leads to short, well readable code, with automatic type conversion of the user input, and automatic error handling in case of erroneous input data. Furthermore, parameter files in XML format are well structured and to a large extent self-documenting.

If for some reason you are unwilling to use XML files for user input, you can still use plain `plb_ifstream` files, but you need to manually broadcast the data to all processors in order to get a working parallel program which is compatible with the data-parallel programming concept of Palabos. To illustrate this, let us consider a situation in which the dimensions `nx` and `ny` of the matrix are written at the beginning of the file `energy.dat` and are read into the program in order to construct the scalar-field automatically with the right dimensions:

```
plb_ifstream ifile("energy.dat");
if (ifile.is_open()) {
    plint nx, ny;
    ifile >> nx >> ny;
    // Broadcast the input data to all processors; otherwise, the behavior
    //   of the program is undefined.
    global::mpi().bCast(&nx, 1);
    global::mpi().bCast(&ny, 1);
    MultiScalarField2D<T> energy(nx,ny);
    ifile >> energy;
}
```

This way of handling input values is illustrated in the code `examples/codesByTopic/io/manualUserInput.cpp`. Please remark that the MPI-related code also compiles in non-parallel programs, in which it simply has no effect. It is therefore good to write them in all cases, to guarantee that the program works in parallel without modification.

There is no object `xcin` for interactive input in Palabos; user input must be received either through an input file or directly from the command line.

Accessing command-line parameters

As in usual C++ programs, command-line parameters can be accessed through the `argc` and `argv` parameters to the main function:

```
int main(int argc, char* argv[])
```

This works fine in both serial and parallel programs. Additionally to this, Palabos offers two global objects of name `argc` and `argv` through which the command-line parameters can be accessed from any location in the program. It is generally recommended to use these global objects instead of the `main` parameters, because they are more intelligent, with automatic type conversions and error handling. Their usage is illustrated in the following code:

```
int main(int argc, char* argv[]) {
    // Never forget to initialize Palabos, in every program.
    plbInit(&argc, &argv);

    pcout << "The number of input arguments is: "
```

```
        << global::argc() << endl;

    double double_argument;
    plint  plint_argument;
    string string_argument;

    try {
        global::argv(1).read(double_argument);
        global::argv(2).read(plint_argument);
        global::argv(3).read(string_argument);
    }
    // React to errors, either if there are less than 3 arguments,
    //   or if one of the arguments fails to convert to the expected
    //   data type.
    catch(PlbIOException& exception) {
        // Print the corresponding error message.
        pcout << exception.what() << endl;
        // Terminate the program.
        exit(1);
    }

    // ... Execute the main program ...
}
```

In this example, the C++ exception mechanism is used to react to errors in the user input. If you prefer not to handle exceptions, you can use the `noThrow` version of the `read` function:

```
bool ok = global::argv(1).readNoThrow(double_argument);
if (!ok) {
    pcout << "Error while reading argument 1." << endl;
}
```

As an example for the use of command-line parameters, you can also have a look at the programs `examples/showCases/rayleighTaylor2D.cpp` and `rayleighTaylor3D.cpp`.

Reading user input from an XML file

Palabos offers a way to read structured input data from an XML file. This functionality works both in serial and parallel programs. The data is however stored in plain, non-parallel data structures, which are duplicated over all threads of a parallel program. XML files should therefore be used only for small data sets, such as, the parameters needed to set up a simulation. Large data sets, such as, the content of a lattice, should instead be read into a Palabos data structures, as explained in section `input-streams`, so the data is distributed over the parallel machine.

The XML data is parsed with help of the open-source library `TinyXML`. This library needs not be installed manually, as it is packaged with the Palabos releases.

Palabos can read any well formed XML document, with three restrictions:

- Document Type Definitions (DTDs) or the eXtensible Stylesheet Language (XSL) cannot be parsed.
- Attributes are not recognized (but a tag is still properly parsed, even if attributes are present). For example, the value of the attribute `id` in the following XML tab cannot be accessed in Palabos: `<someTag id="5">`.
- A given tag name can be used only once. If it is used multiple times, only the last occurrence will be accessed in Palabos.

These restrictions are made to simplify the syntax of the XML parser in Palabos, and because they donâ€™t restrict the generality of the input format. The following is a typical input file which could be used with Palabos:

XML file `myInput.xml`:

```
<?xml version="1.0" ?>
<!-- Configuration parameters for my simulation -->

<Geometry>
    <inputFile> /home/user/data/inputFile.dat </inputFile>
    <size>
    <!-- Number of lattice nodes in each space direction. -->
        <nx> 10 </nx>
        <ny> 20 </ny>
        <nz> 30 </nz>
    </size>
    <viscosity>
    <!-- Viscosity in lattice units. -->
        0.023
    </viscosity>
    <umax>
    <!-- Maximum velocity in lattice units. -->
        0.01
    </umax>
</Geometry>

<Inlet>
    <!-- Use a velocity Dirichlet boundary condition. -->
    <kind> Dirichlet </kind>
    <!-- Velocity profile values on the inlet, in
         dimensionless units. -->
    <values> 0 0.1 0.2 0.3 0.4 0.4 0.3 0.2 0.1 0 </values>
</Inlet>
```

And hereâ€™s how they would be parsed in a Palabos program:

```
try {
    // Open the XMl file.
    XMLreader xmlFile("myInput.xml");

    // It's good policy to flush the content of the XML
    //   file to the terminal, so that in future, when
    //   you check the program's output, you remember
    //   which parameters where used.
    pcout << "CONFIGURATION" << endl;
    pcout << "=============" << endl << endl;
    xmlFile.print(0);

    string inputFile;
    xmlFile["Geometry"]["inputFile"].read(inputFile);

    plint nx, ny, nz;
    xmlFile["Geometry"]["size"]["nx"].read(nx);
    xmlFile["Geometry"]["size"]["ny"].read(ny);
    xmlFile["Geometry"]["size"]["nz"].read(nz);
```

```
    T viscosity, uMax;
    xmlFile["Geometry"]["viscosity"].read(viscosity);
    xmlFile["Geometry"]["umax"].read(uMax);

    string inletKind;
    xmlFile["Inlet"]["kind"].read(inletKind);
    vector<T> inletValues;
    xmlFile["Inlet"]["values"].read(inletValues);
}
catch (PlbIOException& exception) {
    pcout << exception.what() << endl;
    return -1;
}
```

It is particularly noted that one can read full data arrays from a single tag, as shown with the the last tag in the above example, which is read into the vector `inletValues`.

Producing images in 2D and 3D simulations

It is possible to produce images from the entire domain, a sub-domain, or a slice of a scalar-field. By default, Palabos writes images in the PPM format, an ASCII format which is easily produced without the need for an external graphics library. If the package `ImageMagick` (and in particular, the command-line tool `convert` contained in this package) is installed, it is also possible to write images in the more standard GIF format. This works as well under Linux as under Windows. Most examples in the directory `examples/showCases` illustrate how to write GIF images.

As a first step, you need to create an object of type `ImageWriter`, as in the following example:

```
ImageWriter<T> imageWriter("leeloo");
```

The parameter string stands for the colormap used to map the scalar variables to a three-component RGB color scheme. The available default maps are `earth`, `water`, `air`, `fire`, and `leeloo`.

The next step is to write a PPM or GIF image, either with a colormap adjusted to a fixed range of color values, or with a scaled range which is automatically adjusted to fit the minimum and maximum value of the data. With the GIF format, the image can also be rescaled to a different size:

```
// Write a PPM image with a color map adapted to the range [minValue, maxValue].
imageWriter.writePpm(scalarField, "myImage", minValue, maxValue);

// Write a PPM image with an automatically scaled color map.
imageWriter.writeScaledPpm(scalarField, "myImage");

// Write a GIF image with a color map adapted to the range [minValue, maxValue].
imageWriter.writeGif(scalarField, "myImage", minValue, maxValue);

// Write a GIF image with an automatically scaled color map.
imageWriter.writeScaledGif(scalarField, "myImage");

// Write a GIF image with an automatically scaled color map, and rescale it in
//  order to fit in a siyeX-by-sizeY box (the aspect ratio is preserved, though).
imageWriter.writeScaledGif(scalarField, "myImage", sizeX, sizeY);
```

It is also extremely useful to use this approach to produce snapshots from a 3D simulation. In this case, you need to extract a slice from the 3D data, in form of a 3D Box with one degenerate dimension (a dimension which has a one-cell-width):

```
Box3D slice(0,nx-1, 0,ny-1, zValue, zValue);
ImageWriter<T>("earth").writeScaledGif(*computeVelocity(lattice, slice));
```

VTK output for post-processing

All data from a block-lattice, a scalar-field, or a tensor-field can be written into a file in a `VTK` data format. From there, it can be further post-processed by a scientific data visualization tool such as `Paraview`.

The VTK format used in Palabos is based on a lossless binary representation of the data by means of the Base64 format (itâ€™s an ASCII-based binary format, the same which is probably used by your e-mail program to encode images in the e-mail). It is often an overkill to use a format which preserves the full numerical accuracy of your simulated data, because post-processing operations often require less numerical precision than CFD simulations. The least you can do to save some memory is to convert the data from double-precision do single-precision arithmetics, as shown in the following 3D example:

```
// Evaluate the discretization parameters, in order to write the data
//    in dimensionless units into the VTK file.
T dx = parameters.getDeltaX();
T dt = parameters.getDeltaT();

// Create a VTK data object and indicate the cell width through the
//   parameter dx. This object is of the same type as the simulation, type T.
VtkImageOutput3D<T> vtkOut("simulationData.dat", dx);

// Add a 3D tensor-field to the VTK file, representing the velocity, and rescale
//   with the units of a velocity, dx/dt. Explicitly convert the data to single-
//   precision floats in order to save storage space.
vtkOut.writeData<3,float>(*computeVelocity(lattice), "velocity", dx/dt);

// Add another 3D tensor-field for the vorticity, again as floats.
vtkOut.writeData<3,float>(*computeVorticity(*computeVelocity(lattice)), "vorticity", 1./dt);

// To end-with add a scalar-field for the density.
vtkOut.writeData<1,float>(*computeDensity(lattice), "density", 1.);
```

You can add as many blocks as you wish to the VTK file, and each of them can have an arbitrary number of components (in the example above, 3-component tensor-fields and a 1-component scalar-field were used).

Similarly, 2D data can be written in a VTK format by using an object of type VtkImageOutput2D (an example is provided in the directory `examples/showCases/poiseuille/`. A tool like Paraview is however not always the best choice for the analysis of 2D data. You are often better advised to flush the data into a text file, as explained in Section `output-streams`, and then visualize it with Octave or Matlab (or Python or R or SciLab or OO-Spreadsheet, or whatever your favorite data interpreter is).

Checkpointing: saving and loading the state of a simulation

While the results of a simulation can be written into a VTK file with the help of a `VtkImageOutputXD` object, the reverse procedure is not implemented: it is not possible to read VTK data into an Palabos data structure. If you wish to save the state of the system in order to resume the simulation at a later time, use the function saveBinaryBlock, as illustrated for example in the file `examples/codesByTopic/io/checkPointing.cpp`:

```
saveBinaryBlock(lattice, "checkpoint.dat");
```

and load it later on with the function loadBinaryBlock:

```
loadBinaryBlock(lattice, "checkpoint.dat");
```

With this procedure, you can only write data one block at a time. If the state of a simulation is represented by various blocks (e.g. in a multi-component fluid which uses a few block-lattices), each of them needs to be saved in a separate file. It is currently not possible to save structural data (dynamics objects and data processors). When loading the state of a simulation, the context, such as boundary conditions, must therefore be recreated before loading the data.

Data evaluation

The variables simulated in a lattice Boltzmann program, the particle populations, contrast with the quantities a typical fluid engineer is interested in, the macroscopic variables pressure, velocity, and others. Obviously, one needs to somehow convert the data before providing it to a standard post-processing tool. Many functions for conversion, evaluation, or transformation of data are provided in Palabos, as listed in `Appendix Non-mutable operations for data analysis and other purposes`. In the present section, the function `computeVelocity` is discussed as an example, as the other functions work in a very similar way. These functions are defined for the 2D and the 3D case, and they work with `atomic-blocks` and `multi-blocks`. For the sake of illustration, the following codes refer to the 3D multi-block case.

The function `computeVelocity` is provided in three versions:

```
// Version 1
void computeVelocity(MultiBlockLattice3D<T,Descriptor>& lattice,
                     MultiTensorField3D<T,Descriptor<T>::d>& velocity, Box3D domain);

// Version 2
std::auto_ptr<MultiTensorField3D<T,3> >
    computeVelocity(MultiBlockLattice3D<T,Descriptor>& lattice, Box3D domain);

// Version 3
std::auto_ptr<MultiTensorField3D<T,3> >
    computeVelocity(MultiBlockLattice3D<T,Descriptor>& lattice);
```

In the first version, the velocity is computed on a sub-domain of a block-lattice, and the result is written to a corresponding sub-domain of a three-component tensor-field. The block-lattice and the tensor-field don't need to have the same size, nor do they need to have the same internal block arrangement. If the domain exceeds the dimensions of either the block-lattice or the tensor-field, the domain is trimmed correspondingly.

In the second version, which is much more often used in practice, a tensor-field with the size of `domain` is automatically created and returned from the function. The auto-pointer keyword used in the return value of the function refers to a class of smart-pointers offered by the C++ standard library. From the user's point of view, it is practically the same as a pointer. This means that you treat an object of type `std::auto_ptr<MultiTensorField3D<T,3> >` in the same way as you would treat an object of type `MultiTensorField3D<T,3>*`. The big difference between the two is that the auto-pointer has an automatic memory management mechanism, and that you never need to call the operator delete on this type of pointers. If the return value of the function were a raw pointer, you wouldn't be able to pipeline different operators, as shown in the next section, because you'd never get an explicit pointer to them and therefore couldn't delete them.

The following code shows a typical use case of the function `computeVelocity`:

```
pcout << *computeVelocity(lattice, domain) << endl;
```

Here, the computed velocity values are immediately redirected to the terminal. The star in front of the function call is used to dereference the pointer to the velocity field. At the end of this program line, the memory for the velocity field is automatically de-allocated, and does not need to be disposed of explicitly.

The third version is a pure convenience function in which the argument `domain` is replaced by the full domain of the lattice, `lattice.getBoundingBox()`.

Pipelining data evaluation operators

The return value of a function like `computeVelocity` can directly be reused as the argument of other data evaluation operators. In this way, it is possible to construct complex expressions. For example, the function `computeAverage` in the previous section could be replaced by a computation of the velocity field, followed by the computation of the norm-square of each element, a division by two, and finally a computation of the average value:

```
pcout << "The value "
      << *computeAverageEnergy(lattice) << " is the same as "
      << computeAverage (
                 *multiply ( 0.5,
                         *computeNormSqr (
                                 *computeVelocity(lattice) ) ) )
      << endl;
```

All the functions used in this example are listed in the Appendix `Appendix: partial function/class reference`. More examples of the evaluation of data, constructions of scalar fields, and combination of data evaluation operators are provided in the directory `examples/codesByTopic/scalarField`.

Particles

Overview

Palabos offers a fully parallel framework for simulating particles, and implementing particle-fluid interaction. We are currently working on the documentation of this feature. In the mean time, get a first flavor by looking at the example code `codesByTopic/particlesInTube`.

Grid refinement

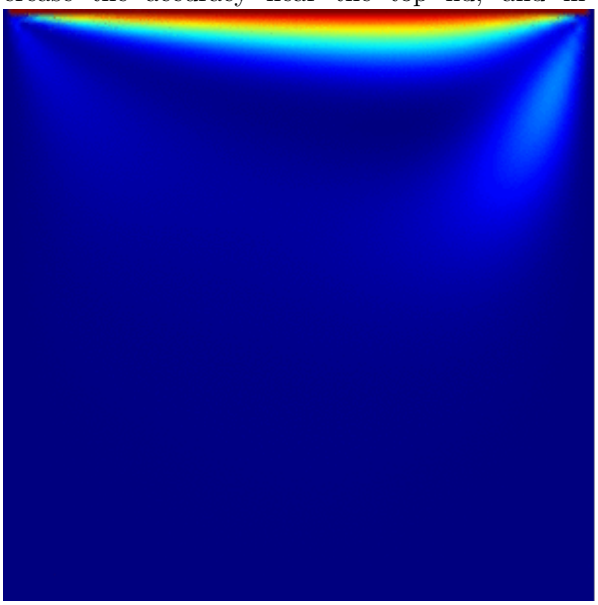
Overview

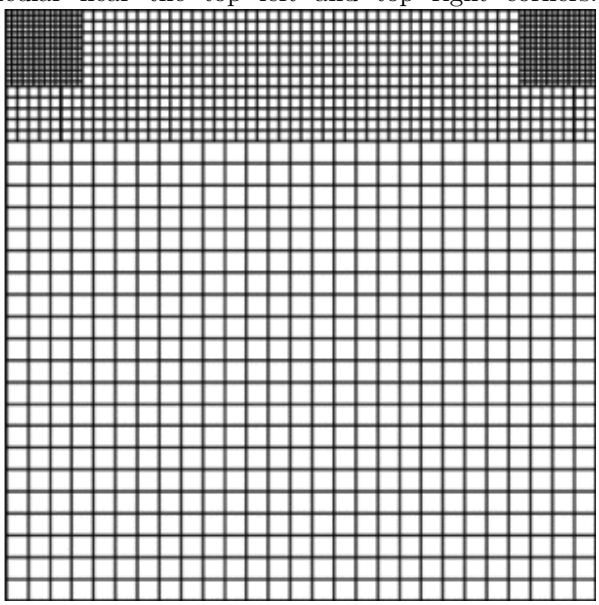
Since Version 0.7, Palabos offers the possibility to work with refined grid. At this point, only 2D grid refinement is implemented. As for the other aspects of Palabos, grid refinement is orthogonal to other implemented concepts. This means for example that previously implemented dynamics objects and data processors work also with a refined grid. Furthermore, refined grids can be parallelized in a general way (the boundaries between the domains on different processes can be chosen without restrictions).

Multi-layer grid refinement

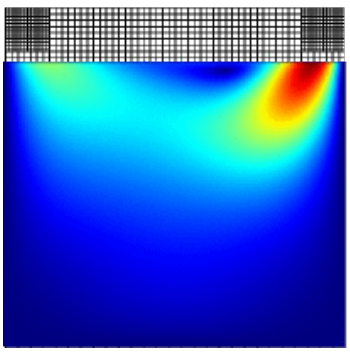
Grid refinement in Palabos is an extension of the multi-block concept, and is referred to as “multi-grid”. Every level of grid refinement is covered by a certain number of blocks and is represented by a multi-block. The full grid - the “multi-grid” - is consituted by a superposition of all these levels.

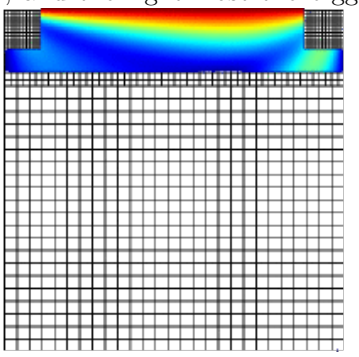
Take as an example the 2D lid-driven cavity, represented by three levels of refined grids, to increase the accuracy near the top lid, and in particular near the top left and top right corners:

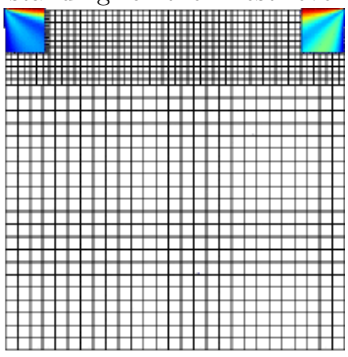




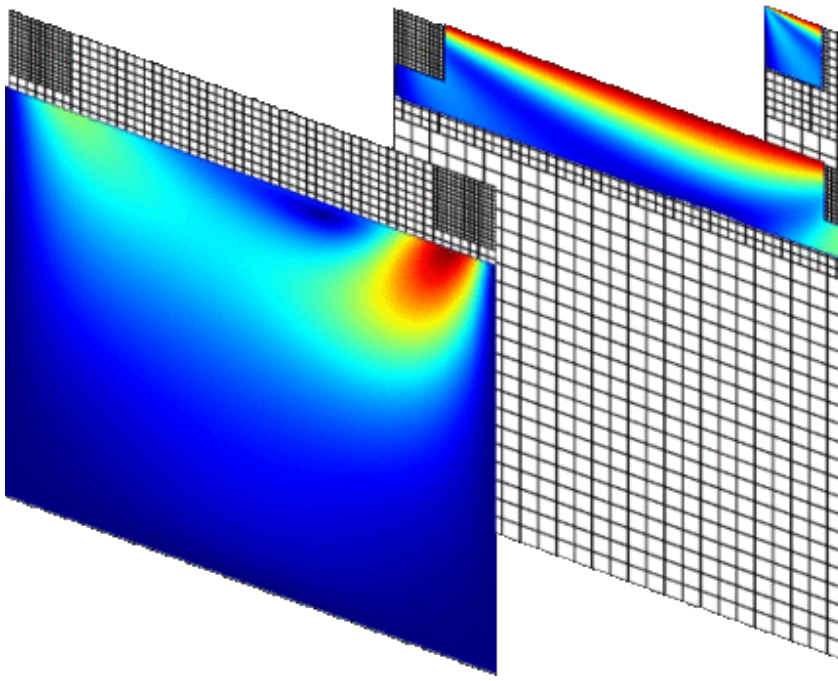
Here's the trick: Palabos simply generates for you three separate multi-blocks, each of which holds the data at a certain level of grid refinement. The multi-blocks have a sparse memory representation, which means that the whole manoeuvre does not cost anything in terms of memory and efficiency. In our case, there are three multi-blocks, as suggested in the image below. The left-most multi-block is the smallest once, because it holds the data at the coarsest level, and the right-most the biggest one, standing for the finest level.







Palabos then gathers the three multi-blocks into a single data structure (but, for convenience, you can still ac-



cess them individually), called “multi-grid” structure:

Interactive creation of a multi-grid

The usage of the multi-grid data structure is illustrated by the example program `showCases/gridRefinement2d`.

Parallelism

Parallel programming approach

Programs written with Palabos can be automatically parallelized, at least if they are following the recommendations provided in this user’s guide. Parallelization is performed with the message-passing paradigm of the MPI library. This approach works fine on distributed-memory platforms (e.g. clusters) as well as on shared-memory platforms (SMP or multi-core architectures). As a matter of fact, due to the high availability of dual-core or many-core processors, it is recommended to compile Palabos programs practically always in parallel, even during the development phase. The significant speedup obtained from exploiting all cores of a machines leads to an improved efficiency during code development and program testing cycle.

To achieve parallelism with programs which have the look and feel of serial applications, Palabos distinguishes between two classes of data. Data which is spatially distributed, such as the block-lattices, the scalar- or the tensor-fields, are handled through a data-parallel paradigm. The data space is partitioned into smaller regions that are distributed over the nodes or cores of a parallel machine. Other data types which require a small amount of storage space are duplicated on every node. All native C++ data types, as well as the data types of the C++ STL, and small data-types of Palabos like the `Array<T,nDim>` are automatically duplicated, by virtue of the Single-Program-Multiple-Data model of MPI. These types should be used to handle scalar values, such as the parameters of the simulation, or integral values over the solution (e.g. the average energy).

Finally, it is pointed out once more that only the multi-block structures (multi-block lattice, multi-scalar field, and multi-tensor field) are data parallel, while their atomic counterparts are duplicated overall cores if they are created in end-user code. For this reason, it is important to always work with multi-block structures in order to get parallel performance improvements.

Controlling the efficiency

Rules for efficient parallel programs

The Rule 0 for efficient program implementations in Palabos has been emphasized and repeated many times in the user's guide, because it is extremely important.

Rule 0 for efficient program implementations:: An end-user application should never contain a loop which runs over the indices of a multi-block. It is OK to access the cells of a multi-block occasionally, through their array-like interface, to evaluate or modify data. But it is NOT OK to access them repeatedly from within a space loop, because this would be catastrophically slow in a parallel programs. Space loops must appear inside data processors and nowhere else.

Once this rule is respected, most other ingredients leading to an efficient parallel program are simple common sense. For example, input/output operations are relatively slow in parallel programs, because of the way they are currently implemented (see Section bottlenecks). You should therefore make sure that the program doesn't output data too often. Many of the programs provided in the directory examples are for example pretty slow in parallel, because they frequently produce images to illustrate a point. If you try to benchmark their parallel performance, you should therefore comment out the output operations.

In many cases, it is also possible to improve the efficiency of a program substantially by avoiding collective MPI operations, such as reductions. The lattice Boltzmann algorithms don't actually need reduction operations to execute their time iterations. Reductions are just executed out of convenience at every time step, to evaluate the internal statistics (the average energy and the average density). These internal statistics are extremely useful in serial programs, because they are computed at practically no charge, but they can virtually ruin the performance in parallel programs. This is due to the fact that a reduction acts like a synchronization barrier, a fact which is observed to have a negative impact on performance, especially on cluster-like parallel machines. It is therefore possible to turn off the internal statistics (and thus, the reduction operations), through a function call

```
lattice.toggleInternalStatistics(false);
```

If you'd just like to know the internal statistics from time to time, say, once in twenty time iterations, it is possible to turn them on just for a short moment,

```
lattice.toggleInternalStatistics(true);
```

then, execute the time iteration:

```
lattice.collideAndStream();
```

and, finally, access the statistics and turn them off again:

```
T rho = getStoredAverageDensity<T>(lattice);
lattice.toggleInternalStatistics(false);  /// The value in website is true. I think it is a mistake.
```

How many processors to use?

When a program is parallelized on few cores, it is common for the parallel efficiency to be nearly optimal. Each time the number of cores is doubled, the execution speed is almost doubled as well. As the number of cores if further increased, the speed up curve starts flattening, and at some point it's not worth to add any more cores. It is common to speak about the efficiency of the program as the execution time on one core, divided by the execution time on a given number n of cores, and divided by `n`. If the program runs at, say, 75% efficiency, you can argue that it works reasonably well, because parallelization is never optimal. If on the other hand the efficiency drops below 50%, it is reasonable to demand that you reduce the number of cores, as you are just wasting the resources of your institute.

Without getting into the details of the theory of parallelism, let us just mention that the optimal number of cores to use for a problem depends on many parameters. The parallel efficiency tends to be better for

simulations on large domains, and it is favorably influenced by a good interconnecting network (actually, by a high ratio of communication speed over the computation speed of a core). In practice, the optimal number of cores must be determined empirically for a given problem and a given parallel machine. A value which tends to be easy to remember is the size of the sub-domain associated to each core when the program runs in an optimal regime. On a given home-grown cluster, the example program `cavity3d` was for example observed once to run at 75% efficiency when the per-core domain size (the total domain size divided by the number of cores) was approximately 20-by-20-by-20. Since then, this machine is associated with the "20-cube-rule", by which it is easy to rapidly evaluate the optimal number of cores to be used for a given problem size.

Bottlenecks in the current implementation

While intense efforts where invested to obtain an excellent parallel efficiency in Palabos, there is still room for improvement. Currently, one of the most severe bottlenecks appears to be given by input/output operations. For both input and output, the data is always transferred through one of the cores. These operations are therefore non-parallel: they are not faster, or only marginally faster on many cores than on a single core. While this is something one can live with on smaller parallel machines with a few hundred cores, it tends to become problematic on thousand-core hardware were input/output operations can dominate the overall execution time of a program. Parallel input/output is therefore one of the most important features which needs to be implemented in a near future.

Data processors for non-local operations and couplings between blocks

Dynamics classes and data processors are the nuts and bolts of a Palabos program. Conceptually speaking, you can view dynamics classes as an implementation of a raw "lattice Boltzmann paradigm", whereas data processors are rather a realization of a more general data-parallel multi-block paradigm. Dynamics classes are (relatively) easy, because lattice Boltzmann is easy, as long as there are no boundaries, refined grids, parallel programs, or any other advanced structural ingredients. Data processors can be a little bit harder to understand, because they have the difficult task to solve "everything else" which is not taken charge of by dynamics classes. They implement all non-local ingredients of a model, they execute operations on scalar-fields and tensor-fields, and they create couplings between all types of blocks. And, just like dynamics objects, they handle reduction operations, and they must be efficient and inherently parallelizable.

The importance of data processors is particularly obvious when you are working with scalar-fields and tensor-fields. Unlike block-lattices, these data fields do not have intelligent cells with a local reference to a dynamics object; data processors are therefore the only efficient way of performing collective operations on them. As an example, consider a field of type `TensorField3D<T,3>` that represents a velocity field (each element is a velocity vector of type `Array<T,3>`). Such a field is for example obtained from a LB simulation by calling the function `computeVelocity`. It might be interesting to compute space-derivatives of the velocity through finite differences. The only morally right way of doing this is through a data processor, because it is the only approach which is parallelizable, scalable, efficient, and forward-compatible with future versions of Palabos. This is exactly what Palabos does when you call one of the functions like `computeVorticity()` or `computeStrainRate()` (see the appendix `[tensor-field -> tensor-field]`). Note, once more, that you could also evaluate the finite difference scheme by writing a simple loop over the space indices of the tensor-field. This would produce the correct result in serial and in parallel, and it would even be pretty efficient in serial, but it is not parallelizable (in parallel, efficiency is lost instead of gained).

All in all, it should have become clear that while data processors are powerful objects, they also need to address a certain amount of complexity and are therefore conceptually less simple than other components of Palabos. Consequently, data-processors are most certainly the part of Palabos' user's interface which is toughest to understand, and this section is filled with ad-hoc rule which you just need to learn at some point. To alleviate this thing a bit, the section starts with two relatively simple topics which teach you how to solve

many tasks in Palabos through the available helper functions and avoid explicitly writing data processors in many cases. The remainder of this section contains many links to existing implementations in Palabos, and you are strongly encouraged to actually have a look at these examples to assimilate the theoretical concepts.

Using helper functions to avoid explicitly writing data processors

Data processors can be (arbitrarily) split into three categories, according to the use which is made out of them. The first category is about setting up a simulation, assigning dynamics objects to a sub-domain of the lattice, initializing the populations, and so on. These methods are explained in the sections `Initial values of density and velocity` and `Defining boundary conditions`, and the functions are listed in the appendix `Mutable (in-place) operations for simulation setup and other purposes`. The second category embraces data processors which are added to a lattice to implement a physical model. Many models are predefined, as presented in section `Implemented fluid models`. Finally, the last category is for the evaluation of the data, and for the execution of short post-processing task, as in the above example of the computation of a velocity gradient. Examples are found in the section `Data evaluation`, and the available functions are listed in the appendix `Non-mutable operations for data analysis and other purposes`.

Convenience wrappers for local operations

Imagine that you have to perform a local initialization task on a block-lattice, i.e. a task for which you don't need to access the value of surrounding neighbors, and for which the available Palabos functions are insufficient. As a lightweight alternative to writing a classical data-processing functional, you can implement a class which inherits from `OneCellFunctionalXD` and implements the virtual method (here in 2D)

```
virtual void execute(Cell<T,Descriptor>& cell) const;
```

In the body of this method, simply perform the desired action on the argument cell. If the action depends on the space position, you can instead inherit from `OneCellIndexedFunctionalXD` and implement the method (again illustrated for the 2D case)

```
virtual void execute(plint iX, plint iY, Cell<T,Descriptor>& cell) const;
```

An instance of this one-cell functional is then applied to the lattice through a function call like

```
// Case of a plain one-cell functional.
applyIndexed(lattice, domain, new MyOneCellFunctional<T,Descriptor>);
// Case of an indexed one-cell functional.
applyIndexed(lattice, domain, new MyOneCellIndexedFunctional<T,Descriptor>);
```

This method is used in the example program located in `examples/showCases/multiComponent2d`. Here, a customized initialization process is required to get access to the external scalars of a lattice for a thermal simulation.

These one-cell functionals are less general than usual data processing functionals, because they cannot be used for non-local operations. Furthermore, they tend to be numerically somewhat less efficient, because Palabos needs to perform a virtual function call to the method execute of the one-cell functional on each cell of the domain. However, this loss of efficiency is usually completely negligible during the initialization stage, where it is important to have a code which scales on a parallel machine, but not to optimize the code for a gain of a micro-second. In this case you should prefer a code which, as proposed by the one-cell functionals, is shorter and easier to read.

Writing data processors (or actually, data-processing functionals)

A common way to execute an operation on a matrix is to write some sort of loop over all elements of the matrix, or at least over a given sub-domain, and to execute the operation on each cell. If the memory of

the matrix is subdivided into smaller components, as it is the case for Palabos' multi-blocks, and these components are distributed over the nodes of a parallel machine, then your loop needs also to be subdivided into corresponding smaller loops. The purpose of the data processors in Palabos is to perform this subdivision automatically for you. It then provides the coordinates of the subdivided domains, and requires from you to execute the operation on these domains, instead of the original full domain.

As a developer of a data processor, you're almost always in touch with so-called data-processing functionals which provide a simplified interface by performing a part of the repetitive tasks behind the scenes. There exist many different types of data-processing functionals, as listed in the next section. For the sake of illustration, we consider now the case of a data-processor which acts on a single 2D block-lattice or multi-block-lattice, and which does not perform any data reduction (it returns no value). Let's say that the aim of the operation is to exchange the value of `f[1]` and `f[5]` on a given amount on lattice cells. This could (and should, for the sake of code clarity) be done with the simple one-cell-functional introduced in Section `Convenience wrappers for local operations`, but we want to do the real thing now, and get to the core of data functionals.

The data-processing functional for this job must inherit from `BoxProcessingFunctional2D_L` (the L indicates that the data processor acts on a single lattice), and implement, among other methods described below, the virtual method process:

```
template<typename T, template<typename U> class Descriptor>
class Invert_1_5_Functional2D : public BoxProcessingFunctional2D_L<T,Descriptor> {
    public:
    // ... implement other important methods here.
    void process(Box2D domain, BlockLattice2D<T,Descriptor>& lattice)
    {
        for (plint iX=domain.x0; iX<=domain.x1; ++iX) {
            for (plint iY=domain.y0; iY<=domain.y1; ++iY) {
                Cell<T,Descriptor>& cell = lattice.get(iX,iY);
                // Use the function swap from the C++ STL to swap the two values.
                std::swap(cell[1], cell[5]);
            }
        }
    }
};
```

The first argument of the function `process` corresponds to domain computed by Palabos after sub-dividing the original domain to fit to the small components of the original block. The second argument is corresponds to the lattice on which the operation is to be performed. This argument is always an atomic-block (i.e. it is always a block-lattice and never a multi-block-lattice), because at the point of the function call to `process`, Palabos has already subdivided the original block and is accessing its internal, atomic sub-blocks. If you compare this to the procedure of writing a one-cell-functional as shown in Section `Convenience wrappers for local operations`, you will see that the additional work you need to do in the present case is to write yourself the loop over the space indices of the domain. Having to write out these loops by hand all the time is tiring, especially when you write many data processors, and it is error-prone. But it is the cost to pay for optimal efficiency, and in the field of computational physics, efficiency counts just a bit more than in other domains of software engineering and must be valued, unfortunately, against elegance from time to time.

Another advantage against one-cell-functionals is the possibility to implement non-local operations. In the following example, the value of `f[1]` is swapped with `f[5]` on the right neighboring cell:

```
void process(Box2D domain, BlockLattice2D<T,Descriptor>& lattice)
{
    for (plint iX=domain.x0; iX<=domain.x1; ++iX) {
        for (plint iY=domain.y0; iY<=domain.y1; ++iY) {
            Cell<T,Descriptor>& cell = lattice.get(iX,iY);
            Cell<T,Descriptor>& partner = lattice.get(iX+1,iY);
            std::swap(cell[1], partner[5]);
```

```
        }
    }
}
```

You can do this without risking to access cells outside the range of the lattice if you respect two rules:

1. On nearest-neighbor lattices (D2Q9, D3Q19, etc.), you can be non-local by one cell but no more (you may write `lattice.get(iX+1,iY)` but not `lattice.get(iX+2,iY)`). On a lattice with extended neighborhood you can also extend the distance at which you access neighboring cells in a data processor. The amount of allowed non-locality is determined by the constant `Descriptor<T>::vicinity`.
2. Non-local operations are not allowed in data processors which act on the communication envelope (Section `The methods you need to override` explains what this means).

To conclude this sub-section, let's summarize the things you are allowed to do in a data processors, and the things you are not allowed to. You are allowed to access the cells in the provided range (plus the nearest neighbors, or a few more neighbors according to the lattice topology), to read them and to modify them. The operation which you perform can be space dependent, but this space dependency must be generic and cannot depend on the specific coordinates of the argument `domain` provided to the function `process`. This is an extremely important point which we shall line out as the Rule 0 of data processors:

Rule 0 of data processors:: A data processor must always be written in such a way that executing the data processor on a given domain has the same effect as splitting the domain into two sub-domains, and then executing the data processor consecutively on each of these sub-domains.

In practice, this means that you are not allowed to make any logical decision based on the parameters `x0`, `x1`, `y0`, or `y1` of the argument `domain`, or directly based on the indices `iX` or `iY`. Instead, these local indices must first be converted to global ones, independent of the sub-division of the data processor, as explained in Section `Absolute and relative position`.

Categories of data-processing functionals

Depending on the type of blocks on which a data processor is applied, there exist different types of processing functionals, as listed below:

```
Class BoxProcessingFunctionalXD<T>
void processGenericBlocks(BoxXD domain, std::vector<AtomicBlockXD<T>*> atomicBlocks);
```

This class is practically never used. It is the fall-back option when everything else fails. It handles an arbitrary number of blocks of arbitrary type, which were casted to the generic type `AtomicBlockXD`. Before use, you need to cast them back to their real type.

```
Class LatticeBoxProcessingFunctionalXD<T,Descriptor>
void process(BoxXD domain, std::vector<BlockLatticeXD<T,Descriptor>*> lattices);
```

Use this class to process an arbitrary number of block-lattices, and potentially create a coupling between them. This data-processing functional is for example used to define a coupling between an arbitrary number of lattice for the Shan/Chen multi-component model defined in the files `src/multiPhysics/shanChenProcessorsXD.h` and `.hh`. This type of data-processing functional is not very frequently used either, as the two-block versions listed below are more appropriate in most cases.

```
Class ScalarFieldBoxProcessingFunctionalXD<T>
void process(BoxXD domain, std::vector<ScalarFieldXD<T>*> fields);
```

Same as above, applied to scalar-fields.

```
Class TensorFieldBoxProcessingFunctionalXD<T,nDim>
void process(BoxXD domain, std::vector<TensorFieldXD<T,nDim>*> fields);
```

Same as above, applied to tensor-fields.

```
Class BoxProcessingFunctionalXD_L<T,Descriptor>
void process(BoxXD domain, BlockLatticeXD<T,Descriptor>& lattice);
```

Data processor acting on a single lattice.

```
Class BoxProcessingFunctionalXD_S<T>
void process(BoxXD domain, ScalarFieldXD<T>& field);
```

Data processor acting on a single scalar-field.

```
Class BoxProcessingFunctionalXD_T<T,nDim>
void process(BoxXD domain, TensorFieldXD<T,nDim>& field);
```

Data processor acting on a single tensor-field.

```
Class BoxProcessingFunctionalXD_LL<T,Descriptor1,Descriptor2>
void process(BoxXD domain, BlockLatticeXD<T,Descriptor1>& lattice1, BlockLatticeXD<T,Descriptor2>& latti
```

Data processor for processing and/or coupling two lattices with potentially different descriptors. Similarly, there is an `SS` version for two scalar-fields, and a `TT` version for two tensor-fields with potentially different dimensionality nDim.

```
Class BoxProcessingFunctionalXD_LS<T,Descriptor>
void process(BoxXD domain, BlockLatticeXD<T,Descriptor>& lattice, ScalarFieldXD<T>& field);
```

Data processor for processing and/or coupling a lattice and a scalar-field. Similarly, there is an `LT` and an `ST` version for the lattice-tensor and the scalar-tensor case.

For each of these processing functionals, there exists a "reductive" version (e.g. `ReductiveBoxProcessingFuncionalXD_L`) for the case that the data processor performs a reduction operation and returns a value.

The methods you need to override

Additionally to the method `process`, a data-processing functional must override three methods. The use of these three methods is now illustrated for the example of class `Invert_1_5_Functional2D` introduced at the beginning of this section:

```
BlockDomain::DomainT appliesTo() const
{
    return BlockDomain::bulk;
}

void getModificationPattern(std::vector<bool>& isWritten) const
{
    isWritten[0] = true;
}

Invert_1_5_Functional2D<T,Descriptor>* clone() const
{
    return new Invert_1_5_Functional2D<T,Descriptor>(*this);
}
```

To start with, you need to provide the method `clone()` which is paradigmatic in Palabos (see Section `Programming with Palabos`). Next, you need to tell Palabos which of the blocks treated by the data processor are being modified. In the present case, there is only one block. In the general case, the size of the vector `isWritten` is equal to the number of involved blocks, and you must assign a flag `true/false` to each of them. Among others, this information is exploited by Palabos to decide whether an inter-process communication for the block is needed after execution of the data processor.

The third method, method `appliesTo` is used to decide whether the data processor acts only on the actual domain of the simulation (`BlockDomain::bulk`) or if it also includes the communication envelopes (`BlockDomain::bulkAndEnvelope`). Let's remember that the atomic-blocks which are the components of a multi-block are extended by a single cell layer (or a multiple cell-layer for extended lattices) to incorporate communication between blocks. This envelope overlaps with the bulk of another atomic-block, and the information is duplicated between the corresponding bulk and envelope cells. It is this envelope which makes it possible to implement a non-local data processor without incurring the danger of accessing out-of-range data. Normally, it is sufficient to execute a data processor on the bulk of the atomic blocks, and it is better to do so, in order to avoid the restrictions listed below when using the envelopes. This is sufficient, because a communication is automatically initiated between the envelopes and the bulk of neighboring blocks to update the values in the envelope if needed. Including the envelope is only needed if (1) you assign a new dynamics object to some or all of the cells (as it is done when you call the function `defineDynamics`), or (2) if you modify the internal state of the dynamics object (as it is done when you call the function `defineVelocity` to assign a new velocity value on Dirichlet boundary nodes). In these cases, including the envelope is necessary, because the nature and the content of dynamics objects is not transferred during the communication step between atomic-blocks. The only information which is transferred is the cell data (the particle populations and the external scalars).

If you decide to include the envelope into the application area of the data processor, you must however respect the two following rules. Otherwise, undefined behavior shall arise.

1. The data processor must be entirely local, because there are no additional envelopes available to cope with non-local data access.
2. The data processor can have write access to at most one of the involved blocks (the vector `isWritten` returned from the method `getModificationPattern()` can have the value true at most at one place).

Absolute and relative position

The coordinates `iX` and `iY` used in the space loop of a data processor are pretty useless for anything else than the execution of the loop, because they represent local variables of an atomic-block, which is itself situated at a random position inside the overall multi-block. To make decision depending on a space position, the local coordinates must therefore first be converted to global ones:

```
// Access the position of the atomic-block inside the multi-block.
Dot2D relativePosition = lattice.getLocation();

// Convert local coordinates to global ones.
plint globalX = iX + relativePosition.x;
plint globaly = iY + relativePosition.y;
/// It is different between here and website. I think it is a mistake on the website.
```

An example is provided in the directory `examples/showCases/boussinesqThermal2d/`. This conversion is a bit awkward, and this is again a good reason to use the one-cell functionals presented in Section Convenience wrappers for local operations, which do the job automatically for you.

Similarly, if you execute a data processor on more than just one block, the relative coordinates are not necessarily the same in all involved blocks. If you measure things in global coordinates, then the argument domain of the method process always overlaps with all of the involved blocks. This is something which is guaranteed by the algorithm implemented in Palabos. However, all multi-blocks on which the data processor is applied are not necessarily working with the same internal data distribution, and have potentially a different interpretation of local coordinates. The argument domain of the method process is always provided as local coordinates of the first atomic-block. To get at the coordinates of the other blocks, a corresponding conversion must be applied:

```
Dot2D offset_0_1 = computeRelativeDisplacement(lattice0, lattice1);
Dot2D offset_0_2 = computeRelativeDisplacement(lattice0, lattice2);
```

```
plint iX1 = iX + offset_0_1.x;
plint iY1 = iY + offset_0_1.y;
plint iX2 = iX + offset_0_2.x;
plint iY2 = iY + offset_0_2.y;
```

Again, this process is illustrated in the example in `examples/showCases/boussinesqThermal2d/`. This displacement needs to be computed if any of the following conditions is verified (if you are unsure, it is best to compute the displacement by default):

1. The multi-blocks on which the data processor is applied don't have the same data distribution, because they were constructed differently.
2. The multi-blocks on which the data processor is applied don't have the same data distribution, because they don't have the same size. This is the case for all functions like `computeVelocity`, which computes the velocity on a sub-domain of the lattice. It uses a data-processor which acts on the original lattice (which is big) and the velocity field (which can be smaller because it has the size of the sub-domain).
3. The data processor includes the envelope. In this case, a relative displacement stems from the fact that bulk nodes are coupled with envelope nodes from a different atomic-block. This is one more reason why it is generally better not to include the envelope it the application domain of a data processor.

Executing, integrating, and wrapping up data-processing functionals

There are basically two ways of using a data processor. In the first case, the processor is executed just once, on one or more blocks, through a call to the function `executeDataProcessor`. In the second case, the processor is added to a block through a call to the function `addInternalProcessor`, and then adopts the role of an internal data processor. An internal data processor is part of the block and can be executed as many times as wished by calling the method `executeInternalProcessors` of this block. This approach is typically chosen when the data processing step is part of the algorithm of the fluid solver. As examples, consider the non-local parts of boundary conditions, the coupling between components in a multi-component fluid, or the coupling between the fluid and the temperature field in a thermal code with Boussinesq approximation. In a block-lattice, internal processors have a special role, because the method `executeInternalProcessors` is automatically invoked at the end of the method `collideAndStream()` and of the method `stream()`. This behavior is based on the assumption that `collideAndStream()` represents a full lattice Boltzmann iteration cycle, and `stream()`, if used, stands at the end of such a cycle. The internal processors are therefore considered to be part of a lattice Boltzmann iteration and are executed at the very end, after the collision and the streaming step.

For convenience, the function call to `executeDataProcessor` and to `addInternalProcessor` was redefined for each type of data-processing functional introduced in Section `Categories of data-processing functionals`, and the new functions are called `applyProcessingFunctional` and `integrateProcessingFunctional` respectively. To execute for example a data-processing functional of type `BoxProcessingFunctional2D_LS` on the whole domain of a given lattice and scalar field (they can be either of type multi-block or atomic-block), the function call to use has the form

```
applyProcessingFunctional (
    new MyFunctional<T,Descriptor>, lattice.getBoundingBox(),
    lattice, scalarField );
```

All predefined data-processing functionals in Palabos are additionally wrapped in a convenience function, in order to simplify the syntax. For example, one of the three versions of the function `computeVelocityNorm` for 2D fields is defined in the file `src/multiBlock/multiDataAnalysis2D.hh` as follows:

```
template<typename T, template<typename U> class Descriptor>
void computeVelocity( MultiBlockLattice2D<T,Descriptor>& lattice,
                      MultiTensorField2D<T,Descriptor<T>::d>& velocity,
```

```
                        Box2D domain )
{
    applyProcessingFunctional (
            new BoxVelocityFunctional2D<T,Descriptor>, domain, lattice, velocity );
}
```

Execution order of internal data processors

There are different ways to control the order in which internal data processors are executed in the function call `executeInternalProcessors()`. First of all, each data processor is attributed to a processor level, and these processor levels are traversed in increasing order, starting with level 0. By default, all internal processors are attributed to level 0, but you have the possibility to put them into any other level, specified as the last, optional parameter of the function `addInternalProcessor` or `integrateProcessingFunctional`. Inside a processor level, the data processors are executed in the order in which they were added to the block. Additionally to imposing an order of execution, the attribution of data processors to a given level has an influence on the communication pattern inside multi-blocks. As a matter of fact, communication is not immediately performed after the execution of a data processor with write access, but only when switching from one level to the next. In this way, all MPI communication required for by the data processors within one level is bundled and executed more efficiently. To clarify the situation, let us write down the details of one iteration cycle of a block-lattice which has data processors at level 0 and at level 1 and automatically executes them at the end of the function call `collideAndStream`:

1. Execute the local collision, followed by a streaming step.
2. Execute the data processors at level 0. No communication has been made so far. Therefore, the data processors at this level have only a restricted ability to perform non-local operations, because the cell data in the communication envelopes is erroneous.
3. Execute a communication between the atomic-blocks of the block-lattice to update the envelopes. If any other, external blocks (lattice, scalar-field or tensor-field) were modified by any of the data processors at level 0, update the envelopes in these blocks as well.
4. Execute the data processors at level 1.
5. If the block-lattice or any other, external blocks were modified by any of the data processors at level1, update the envelopes correspondingly.

Although this behavior may seem a bit complicated, it leads to an intuitive behavior of the program and offers a general way to control the execution of data processors. It should be specially emphasized that if a data processor B depends on data produced previously by another data processor A, you must make sure that a proper causality relation between A and B is implemented. In all cases, B must be executed after A. Additionally, if B is non-local (and therefore accesses data on the envelopes) and A is a bulk-only data-processor, it is required that a communication step is executed between the execution of A and B. Therefore, A and B must be defined on different processor levels.

If you execute data processors manually, you can choose to execute only the processors of a given level, by indicating the level as an optional parameter of the method `executeInternalProcessors(plint level)`. It should also be mentioned that a processor level can have a negative value. The advantage of a negative processor level is that it is not executed automatically through the default function call `executeInternalProcessors()`. It can only be executed manually through the call `executeInternalProcessors(plint level)`. It makes sense to exploit this behavior for data processors which are executed often, but not at every iteration step. Calling `applyProcessingFunctional` each time would be somewhat less efficient, because an overhead is incurred by the decomposition of the data processor over internal atomic-blocks.

Utilities

Timer

Timer objects can be used to make time measurements and performance evaluations. They measure time intervals in terms of wall-clock time, with help of the C function `clock()` in serial programs, and with the MPI function `MPI_Wtime()` in parallel programs. A timer object is accessed through a command like

```
global::timer("nameOfTimer")
```

where `nameOfTimer` is an arbitrary string of your choice. You can use as many different timer objects as needed, and the timers are automatically created the first time you refer to them. Timer objects are global and can measure time intervals cumulatively. You can therefore start measuring a time interval in one file of the program, and then add up more time from a place in another file, by referring to the timer with the same string argument.

A timer behaves like a stop-watch. When you call the method `global::timer("nameOfTimer").start()`, it starts measuring time. After calling the method `stop()`, the clock stops moving, but proceeds from where it was the next time you call `start()`. To reset the clock to zero, call the method `reset()` or `restart()`. And example for the usage of a timer object is provided in the directory `examples/codesByTopic/smagorinskyModel`.

Value tracer

Value tracers are most often used to decide when a simulation has reached a steady state. They track the time evolution of a scalar value, such as the average energy, and decide that a steady state is reached when the standard deviation of this quantity, as measured over a fixed time windows, falls below a given threshold value. An example is provided in the directories `examples/showCases/boussinesqThermal2D` and `3D`.

Appendix: list of example programs

The directory `examples/tutorials` is treated separately in the Palabos tutorial.

Please note: the only purpose of these example codes is to teach the use of Palabos, and in many cases, we have not validated the results from a scientific standpoint. It is your responsibility to understand and possibly adapt the code if you use them in your research.

Directory examples/showCases

This directory contains full-featured (but basic) programs for the simulation of well-known benchmark problems.

aneurysm:

This is the most full-featured example, and is therefore not the best one to pick to get started with Palabos. It illustrates crucial concepts for advanced Palabos simulations:

- Reading a mesh in STL format, voxelizing the domain, and generating an off-lattice boundary condition.
- Copying the result of the simulation from a coarse to a finer grid. In this example, this technique is used to accelerate convergence to a steady state (converge first on the coarse mesh, and use this as an initial condition on the fine mesh).
- Reading user input parameters from an XML file.

- Various types of output, including surface data (wall-shear-stress).

boussinesqThermal2d:

A fluid constrained between a hot bottom wall (no-slip for the velocity) and a cold top wall (no-slip for the velocity). The lateral walls are periodic. Under the influence of gravity, convection rolls are formed. Thermal effects are modeled by means of a Boussinesq approximation: the fluid is incompressible, and the influence of the temperature is visible only through a body-force term, representing buoyancy effects. The temperature field obeys an advection-diffusion equation.

The simulation is first created in a fully symmetric manner. The symmetry is therefore not spontaneously broken; while the temperature drops linearly between the hot and and cold wall, the convection rolls fail to appear at this point. In a second stage, a random noise is added to trigger the instability.

This application is technically a bit more advanced than the other ones, because it illustrates the concept of data processors. In the present case, they are used to create the initial condition, and to trigger the instability.

Techniques illustrated in this code:

- Coupling between a Navier-Stokes and an Advection-Diffusion simulation to simulate a thermal fluid with Boussinesq approximation.
- Use of value-tracer classes to decide when a simulation has reached a steady state.

boussinesqThermal3d:

A fluid constrained between a hot bottom wall (no-slip for the velocity) and a cold top wall (no-slip for the velocity). The lateral walls are periodic. Under the influence of gravity, convection rolls are formed. Thermal effects are modeled by means of a Boussinesq approximation: the fluid is incompressible, and the influence of the temperature is visible only through a body-force term, representing buoyancy effects. The temperature field obeys an advection-diffusion equation.

The simulation is first created in a fully symmetric manner. The symmetry is therefore not spontaneously broken; while the temperature drops linearly between the hot and and cold wall, the convection rolls fail to appear at this point. In a second stage, a random noise is added to trigger the instability.

This application is technically a bit more advanced than the other ones, because it illustrates the concept of data processors. In the present case, they are used to create the initial condition, and to trigger the instability.

Techniques illustrated in this code:

- Coupling between a Navier-Stokes and an Advection-Diffusion simulation to simulate a thermal fluid with Boussinesq approximation.
- Use of value-tracer classes to decide when a simulation has reached a steady state.

breakingDam3d:

In this example, a free-surface flow is implemented that simulates a dam break problem, with a down-stream collision of the water against an obstacle. The example illustrates:

- How to set up a free surface flow problem (two-phase problem in which the effect of one phase is neglected).
- How to describe the geometry of the flow in the free-surface case.
- How to export data, and in particular, how to write an STL file describing the shape of the free surface.

cavity2d:

Flow in a lid-driven 2D cavity. The cavity is square and has no-slip walls, except for the top wall which is driven to the right with a constant velocity. The benchmark is challenging because of the velocity discontinuities on corner nodes. The code on the other hand is very simple. It could for example be used as a first example, to get familiar with Palabos.

Techniques illustrated in this code:

- Use of timer object to benchmark the code.
- Data analysis: computation of velocity, velocity-norm, and vorticity.
- Creation of GIF images and VTK files from 2D simulation data.

cavity3d:

Flow in a diagonally lid-driven 3D cavity. In this 3D analog of the 2D cavity, the top-lid is driven with constant velocity in a direction parallel to one of the two diagonals. The benchmark is challenging because of the velocity discontinuities on corner nodes. The code on the other hand is very simple. It could for example be used as a first 3D example, to get familiar with Palabos.

Techniques illustrated in this code:

- Use of timer object to benchmark the code.
- Data analysis: computation of velocity, velocity-norm, and vorticity.

collidingBubbles3d:

Collision between two bubbles projected against each other, in 3D. The application makes use of the He/Lee multi-phase fluid model.

Techniques illustrated in this code:

- Coupling between lattices and various scalar- and tensor-fields, with the He/Lee algorithm for multi-phase fluids.
- Straightforward initialization of scalar- and tensor-field values through data processing functionals.

cylinder2d:

Flow around a 2D cylinder inside a channel, with the creation of a von Karman vortex street. This example makes use of bounce-back nodes to describe the shape of the cylinder. The outlet is modeled through a Neumann (zero velocity-gradient) condition.

Techniques illustrated in this code:

- Use of a domain functional to instantiate bounce-back nodes on specific locations, from an analytical prescription.
- Instantiation of a Neumann velocity condition for the outlet.
- Accessing the internal statistics to compute quantities like the average kinetic energy and the average density without computational effort.

multiComponent2d:

Rayleigh-Taylor instability occurring with a heavy fluid residing initially on top of a light fluid, in 2D. The application makes use of the Shan/Chen multi-component model for multi-phase fluids.

Techniques illustrated in this code:

- Coupling between two Navier-Stokes lattices with the Shan/Chen algorithm to simulate an immiscible multi-component fluid.
- Simple creation of space-dependent boundary conditions with a one-cell indexed functional.

multiComponent3d:

Rayleigh-Taylor instability occurring with a heavy fluid residing initially on top of a light fluid, in 3D. The application makes use of the Shan/Chen multi-component model for multi-phase fluids.

Techniques illustrated in this code:

- Coupling between two Navier-Stokes lattices with the Shan/Chen algorithm to simulate an immiscible multi-component fluid.
- Simple creation of space-dependent boundary conditions with a one-cell indexed functional.

poiseuille:

Implementation of a stationary, pressure-driven 2D channel flow, and comparison with the analytical Poiseuille profile. The velocity is initialized to zero, and converges only slowly to the expected parabola. This application illustrates a full production cycle in a CFD application, ranging from the creation of a geometry and definition of boundary conditions over the program execution to the evaluation of results and production of instantaneous graphical snapshots. From a technical standpoint, this showcase is not trivial: it implements for example hybrid velocity/pressure boundaries, and uses an analytical profile to set up the boundary and initial conditions, and to compute the error. As a first Palabos example, you might prefer to look at a more straightforward code, such as cavity2d.

Techniques illustrated in this code:

- Definition of functionals (or function objects) to create space-dependent initial and boundary conditions.
- Comparison of the computed velocity field with an analytical formula, and computation the root-mean-square error.
- Definition of velocity or pressure boundaries, or hybrid cases using a velocity condition on some parts of the boundary, and a pressure condition on other parts.

rectangularChannel3d:

Extension of the Poiseuille flow to 3D, in a channel with rectangular cross section. This flow is stationary, and the numerical result is compared to an analytical formula.

womersley:

Implementation of a pulsatile, force-driven 2D channel flow, and comparison with the analytical Womersley profile. The external force varies in time as a simple cosine; the resulting velocity profile is far from trivial, and is described by the Womersley solution.

Techniques illustrated in this code:

- Usage of an external force field (which can be space-dependent; in this example it is however space-independent, but time dependent).
- Implementation of periodic boundaries in one direction.

Directory examples/codesByTopic

particlesInTube

3D steady flow simulation in a tube. Passive scalar particles are injected, and written into an ASCII file for visualization.

bounceBack

Techniques illustrated in this directory:

- `computeDrag.cpp`: Computation of drag and lift forces acting on a bounce-back obstacle, by evaluating the momentum exchange.

boundaryCondition

Techniques illustrated in this directory:

- `neumannOutlets.cpp`: Two different ways of implementing a Neumann-type outlet. (1) Obatin a zero velocity-gradient by copying the velocity from the previous site and use it for the Dirichlet boundary condition. (2) Copy all unknown particle populations from the previous site.

dataAnalysis

Techniques illustrated in this directory:

- `cavity3d.cpp`: Compare (efficient) internal statistics with manually computed values and conclude that they are equal.

dotList

- `cylinder2d.cpp`: Use a data processor based on a dot-list instead of the usual boxed data-processor to create a complex domain.

couplings

Techniques illustrated in this directory:

- `coupleVelocityField.cpp`: Write a data processor to create a coupling between a scalar-field and a block-lattice. In this example, the values of the scalar-field are used to setup the boundary condition in the block-lattice.

include

This directory aggregates common functionalists used by various example codes.

io

Techniques illustrated in this directory:

- `checkPointing.cpp`: Save the state of the simulation to proceed at a later time, after an interruption of the program.
- `loadGeometry.cpp`: Read a boolean mask from an ASCII file and set up bounce-back nodes, based on the values of this bool-mask. Such an approach is useful for complex geometries, such as for example porous media.
- `useIOstream.cpp`: Using output stream to write simulation results in a formatted way into files or to the terminal.
- `userInput.cpp`: Getting input parameters from the command line or from an input file.

multiBlock

Techniques illustrated in this directory:

- `manualBlockCavity2d.cpp`: Use all parameters to the multi-block-lattice constructor to create the distribution of blocks explicitly.
- `manualBlockCavity3d.cpp`: Use all parameters to the multi-block-lattice constructor to create the distribution of blocks explicitly.
- `manualBlockPoiseuille2d.cpp`: Use all parameters to the multi-block-lattice constructor to create the distribution of blocks explicitly.

navierStokesModels

Techniques illustrated in this directory:

- `allModels2d.cpp`: Lists all models available for the incompressible Navier-Stokes equations: BGK, regularized, MRT, entropic.

nonNewtonian

Techniques illustrated in this directory:

- `carreauPoiseuille.cpp`: Use the non-Newtonian Carreau model in a 2D channel.

scalarField

Techniques illustrated in this directory:

- `scalarField2d.cpp`: Perform array-based operations on 2D scalar-fields. In this example, the values of the scalar-field are initialized to a space-dependent sine wave.
- `scalarField3d.cpp`: Perform array-based operations on 3D scalar-fields. In this example, the values of the scalar-field are initialized to a space-dependent sine wave.

smagorinskyModel

Techniques illustrated in this directory:

- `smagorinskyCavity3D.cpp`: LES simulation in a lid-driven cavity with a static Smaogrinsky model. Note that the result is not trustworthy, because the boundary layers are not modeled appropriately.

Appendix: partial function/class reference

Mutable (in-place) operations for simulation setup and other purposes

This appendix presents predefined mutable operations for block-lattices and data fields. Mutable means that the original block-lattice or data field is modified. For analysis and extraction of data, it is most often more appropriate to use non-mutable operations presented in the appendix `Non-mutable operations for data analysis and other purposes`.

Some of these functions accept in their argument list so-called "functionals", or user-defined objects which behave like functions. These functionals inherit from one of the three following virtual base classes:

`OneCellFunctional3D`

> This functional is used to specify a fully local action to be performed on a subset of cells of a lattice. The action is defined by overriding the virtual function `void OneCellFunctional3D::execute(Cell<T,Descriptor>& cell) const`.

`OneCellIndexedFunctional3D`

> With this functional, a fully local, but space-dependent action is performed on a subset of cells of a lattice. The action is defined by overriding the virtual function `void OneCellIndexedFunctional3D::execute(plint iX, plint iY, plint iZ, Cell<T,Descriptor>& cell) const`.

`DomainFunctional3D`

> This functional is used to define the region on a lattice on which an action is performed. The region is defined by overriding the virtual function `bool DomainFunctional3D::operator() (plint iX, plint iY, plint iZ) const`.

Finally, some functions make use of static functionals which, thanks to template mechanisms, are exempt from an inheritance hierarchy. Their syntax depends on the context. For example, the functional `Function f` of the function `setBoundaryDensity` must define a method of the form `T operator() (plint iX, plint iY, plint iZ)`, which returns a value for the density at each space point.

All operations are applied to a sub-domain of the block, which is either rectangular-shaped when specified through the argument domain of type `DomainXD`, or general when specified through the argument domainFunctional of type `DomainFunctional3D`.

Parameters followed by a star `(*)` are passed by-pointer, while all others are passed by-value or by-reference.

Operations on the block-lattice

`apply(lattice, domain, oneCellFunctional*)`

> Apply the same operation to all cells of the domain.

`applyIndexed(lattice, domain, oneCellIndexedFunctional*)`

> Apply a cell-dependent operation to all cells of the domain.

`defineDynamics(lattice, domain, dynamics*)`

Assign a new dynamics object to all cells of the domain. Each cell gets an independent copy of the dynamics object.

`defineDynamics(lattice, boundingBox, domainFunctional*, dynamics*)`

Assign a new dynamics object to all cells of the domain. Each cell gets an independent copy of the dynamics object. The argument `boundingBox` is of type `BoxXD`, and it must contain the domain described by `domainFunctional`. It is used for efficiency reasons, to avoid evaluating the domain-functional unnecessarily often.

`defineDynamics(lattice, dotList, dynamics*)`

Assign a new dynamics object to all cells contained in the dot-list. Each cell gets an independent copy of the dynamics object.

`defineDynamics(lattice, boolMask, domain, dynamics*, flag)`

Assign a new dynamics object to all cells where boolMask evaluates to `flag`. Each cell gets an independent copy of the dynamics object. The flag `flag` is boolean, and `boolMask` is of type `MultiScalaFieldXD<T>`, but is evaluated to Boolean values (smaller than 0.5 means false). If the argument `domain` is not specified, the whole domain of `lattice` is used.

`defineDynamics(lattice, iX, iY, iZ, dynamics*)`

Assign a new dynamics object to the cell at the position `(iX,iY,iZ)`.

`setBoundaryVelocity(lattice, domain, velocity)`

Assign the same velocity to all cells inside the domain which implement a Dirichlet boundary condition for the velocity. On all other cells, this function has no effect.

`setBoundaryVelocity(lattice, domain, Function f)`

Assign as cell-dependent velocity to all cells inside the domain which implement a Dirichlet boundary condition for the velocity. On all other cells, this function has no effect.

`setBoundaryDensity(lattice, domain, rho)`

Assign as cell-dependent density to all cells inside the domain which implement a Dirichlet boundary condition for the density (or for the pressure). On all other cells, this function has no effect.

`setBoundaryDensity(lattice, domain, Function f)`

Assign as cell-dependent density to all cells inside the domain which implement a Dirichlet boundary condition for the density (or for the pressure). On all other cells, this function has no effect.

`initializeAtEquilibrium(lattice, domain, rho, velocity)`

Initialize the cells of the domain at an equilibrium distribution, using the same density and velocity for all of them. The exact form of the equilibrium is defined by the dynamics object residing on the cell at this moment.

`initializeAtEquilibrium(lattice, domain, Function f)`

Initialize the cells of the domain at an equilibrium distribution, using a cell-dependent density and velocity. The exact form of the equilibrium is defined by the dynamics object residing on the cell at this moment.

`stripeOffDensityOffset(lattice, domain, deltaRho)`

Decompose the particle populations into macroscopic variables and off-equilibrium parts. Then, subtract the value deltaRho from the density, and recompose the populations.

`setCompositeDynamics(lattice, domain, compositeDynamics*)`

Assign a composite-dynamics object to all cells of the domain, using the dynamics currently residing on the cell as base dynamics.

`setExternalScalar(lattice, domain, whichScalar, externalScalar)`

Assign the same constant value to the external scalar `whichScalar` on all cells of the domain.

`setExternalVector(lattice, domain, vectorStartsAt, externalVector)`

Here, `externalVector` is of type Array<T,d>. Assign the content of this array to assign a value to 2 (2D) or 3 (3D) external scalars on each cell of the domain. This function is often used to initialize the external force term in forced lattice Boltzmann simulations.

Operations on scalar-fields

`setToConstant(field, domain, value)`

Initialize all scalars in the domain to a constant value.

`setToFunction(field, domain, Function f)`

Initialize all scalars in the domain to a value which is a function of space (function `f` takes two (2D) or three (3D) integer coordinate values, and returns T).

`setToCoordinate(field, domain, index)`

Replace each scalar in the domain by the `index`-component of its space position (x=0, y=1, and z=2 for `index`).

`apply(Function f, field, domain)`

Apply any function or functional f (which takes T and returns T) to the domain.

Scalar-fields: A = A operator alpha

`addInPlace(field, double scalar, Box2D domain)`

Add the scalar value `scalar` to each value of `field` inside the domain.

`subtractInPlace(field, double scalar, Box2D domain)`

Subtract the scalar value `scalar` from each value of `field` inside the domain.

`multiplyInPlace(field, double scalar, Box2D domain)`

Multiply each value of `field` by the scalar `scalar` inside the domain.

`divideInPlace(field, double scalar, Box2D domain)`

Divide each value of `field` by the scalar `scalar` inside the domain.

Scalar-fields: A = A operator B

`addInPlace(A, B, domain)`

Compute A = A+B on the domain.

`subtractInPlace(A, B, domain)`

Compute A = A-B on the domain.

`multiplyInPlace(A, B, domain)`

Compute A = A*B on the domain.

`divideInPlace(A, B, domain)`

Compute A = A/B on the domain.

Operations on tensor-fields

`setToConstant(field, domain, value)`

Initialize all tensors/vectors in the domain to a constant value (`value` is of type `Array<T,nDim>`).

`setToFunction(field, domain, Function f)`

Initialize all tensors/vectors in the domain to a value which is a function of space (function `f` takes two (2D) or three (3D) integer coordinate values, and returns `Array<T,nDim>`).

`setToCoordinates(field, domain, index)`

Replace each vector in the domain by its space position. The tensor-field `field` must hold vectors of dimension 2 (2D) or 3 (3D).

Tensor-fields: A = A operator B

`addInPlace(A, B, domain)`

Compute A = A+B, element-wise on the domain.

`subtractInPlace(A, B, domain)`

Compute A = A-B, element-wise on the domain.

`multiplyInPlace(A, B, domain)`

Compute A = A*B, element-wise on the domain.

`divideInPlace(A, B, domain)`

Compute A = A/B, element-wise on the domain.

Non-mutable operations for data analysis and other purposes

This appendix presents predefined non-mutable operations for block-lattices and data fields. Non-mutable means that the original block-lattice or data field is not modified, and that the result is stored in a new block-lattice or data-field. These operations are often used for data post-processing, and are therefore presented under the name of "data analysis". For actual computations during a simulation, it is most often more appropriate to use in-place operations presented in the appendix `Mutable (in-place) operations for simulation setup and other purposes`.

The operators are defined internally as data processors, and presented in the user interface through convenience functions, as shown in the following for the function `computeDensity()`:

```
std::auto_ptr<MultiScalarField3D<T> > computeDensity(MultiBlockLattice3D<T,Descriptor>&
lattice)
```

This function applies the requested operation over the whole domain of `lattice`, and constructs a new scalar-field into which the result is stored. A function of this form is available for each operator.

`std::auto_ptr<MultiScalarField3D<T> > computeDensity(MultiBlockLattice3D<T,Descriptor>& lattice, Box3D domain)`

In this case, the operation is only applied to the sub-domain domain, and the resulting scalar-field has a size corresponding to `domain`. A function of this form is available for each operator, and this form only is listed in the following to document the operator.

`void computeDensity(MultiBlockLattice3D<T,Descriptor>& lattice, MultiScalarField3D<T>& density, Box3D domain)`

In this function, no new field is allocated, as the result is stored in the field `density`, which is given to this function by argument. A function of this form is always available, except for operators which return a scalar value such as `computeAverageDensity()`.

More explanations on using the non-mutable operations can be found in Section `Data evaluation`, and the exact syntax of the functions is given in the Palabos reference guide.

[block-lattice -> scalar]

`computeAverageDensity(lattice, domain)`

Return the average value of the density, computed over the domain.

`computeAverageRhoBar(lattice, domain)`

Return the average value of `rhoBar`, computed over the domain. Note that while conceptually, the command `computeAverageDensity()` is equivalent to `Descriptor<T>::fullRho(computeRhoBar())`, the latter is likely to be less affected by round-off errors.

`computeAverageEnergy(lattice, domain)`

Return the average of the norm-square of the velocity, divided by two.

[block-lattice -> block-lattice]

`extractSubDomain(lattice, domain)`

Copy the values from the domain `domain` of `lattice` and deposit them in a new lattice of corresponding size.

[block-lattice -> scalar-field], [block-lattice -> tensor-field]

`computeDensity(lattice, domain)`

Compute the density `rho` on each cell.

`computeRhoBar(lattice, domain)`

Compute the value `rhoBar` on each cell.

`computeKineticEnergy(lattice, domain)`

Compute the kinetic energy (half the squared velocity-norm) on each cell.

`computeVelocityNorm(lattice, domain)`

Compute the velocity-norm on each cell.

`computeVelocityComponent(lattice, domain, iComponent)`

Compute the component iComponent of the velocity (`iComponent` ranges from 0 to 1 in 2D, and from 0 to 2 in 3D) on each cell.

`computeVelocity(lattice, domain)`

Compute all components of the velocity on each cell.

`computeDeviatoricStress(lattice, domain)`

Compute the off-equilibrium part of the stress tensor `Pi` on each cell. The result is model-dependent and is determined by the dynamics object defined at the cell location. The result is stored in a 3-component (2D) or 6-component (3D) tensor-field (the number of components is reduced because the tensor is symmetric). To know the index of a specific component, use a notation like `SymmetricTensorImpl<T,3>::xz` for the the xy-component of a 3D symmetric tensor.

`computeStrainRateFromStress(lattice, domain)`

Compute the strain rate S=1/2(grad(u)+grad(u)ˆT), assuming that it is proportional to the deviatoric stress. The result is model-dependent and is determined by the dynamics object defined at the cell location. The result is stored in a 3-component (2D) or 6-component (3D) tensor-field (the number of components is reduced because the tensor is symmetric). To know the index of a specific component, use a notation like `SymmetricTensorImpl<T,3>::xz` for the the xy-component of a 3D symmetric tensor.

`computePopulation(lattice, domain, iPop)`

Extract the population `iPop` on each cell, and deposit the result in a scalar-field. The value `iPop` ranges from 0 to q-1.

[scalar-field –> scalar]

`computeAverage(scalarField, domain)`

Return the average of the values in the scalar-field, over the domain.

`computeBoundedAverage(scalarField, domain)`

Approximate an integral over the domain with second-order accuracy, using the trapezium rule (as compared to `computeAverage()`, the boundary nodes are accounted for with a smaller weight than bulk nodes.

`computeMin(scalarField, domain)`

Return the minimum of the values in the scalar-field, over the domain (the location of this value is not returned).

`computeMax(scalarField, domain)`

Return the maximum of the values in the scalar-field, over the domain (the location of this value is not returned).

###[scalar-field -> scalar-field] `extractSubDomain(scalarField, domain)` > Copy the values from the domain `domain` of `scalarField` and deposit them in a new scalar-field of corresponding size.

Scalar-field: B = A operator alpha

`add(scalar, field, domain)`

Return the result of computing `scalar+field(x,y,z)` on each element of the domain.

`add(field, scalar, domain)`

Return the result of computing `field(x,y,z)+scalar` on each element of the domain.

`subtract(scalar, field, domain)`

Return the result of computing `scalar-field(x,y,z)` on each element of the domain.

`subtract(field, scalar, domain)`

Return the result of computing `field(x,y,z)-scalar` on each element of the domain.

`multiply(scalar, field, domain)`

Return the result of computing `scalar*field(x,y,z)` on each element of the domain.

`multiply(field, scalar, domain)`

Return the result of computing `field(x,y,z)*scalar` on each element of the domain.

`divide(scalar, field, domain)`

Return the result of computing `scalar/field(x,y,z)` on each element of the domain.

`divide(field, scalar, domain)`

Return the result of computing `field(x,y,z)/scalar` on each element of the domain.

Scalar-field: C = A operator B

`add(A, B, domain)`

Return `C=A+B`, applied to every element of the domain.

`subtract(A, B, domain)`

Return `C=A-B`, applied to every element of the domain.

`multiply(A, B, domain)`

Return `C=A*B`, applied to every element of the domain.

`divide(A, B, domain)`

Return `C=A/B`, applied to every element of the domain.

[tensor-field -> tensor-field]

`extractSubDomain(tensorField, domain)`

Copy the values from the domain `domain` of `tensorField` and deposit them in a new tensor-field of corresponding size.

`extractComponent(tensorField, domain, iComponent)`

Copy the component `iComponent` from each cell of `tensorField` into a resulting scalar-field.

`computeVorticity(velocity, domain)` [3D only]

Compute the vorticity of each cell in a 3D velocity field, and store the result in a 3D vorticity field. The vorticity is evaluated through a central (second-order accurate) finite difference scheme on the bulk of the domain, and through an asymmetric, nearest-neighbor scheme on boundary nodes.

`computeBulkVorticity(velocity, domain)` [3D only]

Compute the vorticity of each cell in a 3D velocity field, and store the result in a 3D vorticity field. The vorticity is evaluated through a central (second-order accurate) finite difference scheme on all cells. If `domain` includes non-periodic boundary nodes of the simulation, you should rather use `computeVorticity()` to account for boundary effects.

`computeStrainRate(velocity, domain)`

Compute the strain rate (S=1/2(grad(U)+grad(U)^T) at each cell of a velocity field, and store the result in a 3-component (2D) or 6-component (3D) tensor-field (the number of components is reduced because the tensor is symmetric). To know the index of a specific component, use a notation like `SymmetricTensorImpl<T,3>::xz` for the the xy-component of a 3D symmetric tensor. The Strain-rate is evaluated through a central (second-order accurate) finite difference scheme on the bulk of the domain, and through an asymmetric, nearest-neighbor scheme on boundary nodes.

`computeBulkStrainRate(velocity, domain)`

Compute the strain rate at each cell of a velocity field, and store the result in a 3-component (2D) or 6-component (3D) tensor-field. The strain-rate is evaluated through a central (second-order accurate) finite difference scheme on all cells. If `domain` includes non-periodic boundary nodes of the simulation, you should rather use `computeStrainRate()` to account for boundary effects.

[tensor-field -> scalar-field]

`computeNorm(tensorField, domain)`

Compute the Euclidean norm of each cell in `tensorField`.

`computeNormSqr(tensorField, domain)`

Compute the square of the Euclidean norm of each cell in `tensorField`.

`computeSymmetricTensorNorm(tensorField, domain)`

Compute the Euclidean norm of each cell in the symmetric tensor `tensorField`. It is assumed that `tensorField` only stores half the off-diagonal components, due to symmetry.

`computeSymmetricTensorNormSqr(tensorField, domain)`

Compute the square of the Euclidean norm of each cell in the symmetric tensor `tensorField`. It is assumed that `tensorField` only stores half the off-diagonal components, due to symmetry.

`computeSymmetricTensorTrace(tensorField, domain)`

Compute the trace of each cell in the symmetric tensor `tensorField`. It is assumed that `tensorField` only stores half the off-diagonal components, due to symmetry.

`computeVorticity(velocity, domain)` [2D only]

Compute the vorticity of each cell in a 2D velocity field, and store the result in a scalar-field. The vorticity is evaluated through a central (second-order accurate) finite difference scheme on the bulk of the domain, and through an asymmetric, nearest-neighbor scheme on boundary nodes.

`computeBulkVorticity(velocity, domain)` [2D only]

Compute the vorticity of each cell in a 2D velocity field, and store the result in a scalar-field. The vorticity is evaluated through a central (second-order accurate) finite difference scheme on all cells. If `domain` includes non-periodic boundary nodes of the simulation, you should rather use `computeVorticity()` to account for boundary effects.

Tensor-field: C = A operator B

`add(A, B, domain)`

Return `C=A+B`, applied to every element of the domain.

`subtract(A, B, domain)`

Return `C=A-B`, applied to every element of the domain.

`multiply(A, B, domain)`

Return `C=A*B`, applied to every element of the domain.

`divide(A, B, domain)`

Return `C=A/B`, applied to every element of the domain.

Appendix: Copyright and license agreements

Copyright of the Palabos user guide

Tiny changes are made by JeffHugh at 2019.

Copyright and open-source license for Palabos

GNU AFFERO GENERAL PUBLIC LICENSE Version 3, 19 November 2007

Preamble

The GNU Affero General Public License is a free, copyleft license for software and other kinds of works, specifically designed to ensure cooperation with the community in the case of network server software.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, our General Public Licenses are intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

Developers that use our General Public Licenses protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License which gives you legal permission to copy, distribute and/or modify the software.

A secondary benefit of defending all users' freedom is that improvements made in alternate versions of the program, if they receive widespread use, become available for other developers to incorporate. Many developers of free software are heartened and encouraged by the resulting cooperation. However, in the case of software used on network servers, this result may fail to come about. The GNU General Public License permits making a modified version and letting the public access it on a server without ever releasing its source code to the public.

The GNU Affero General Public License is designed specifically to ensure that, in such cases, the modified source code becomes available to the community. It requires the operator of a network server to provide the source code of the modified version running there to the users of that server. Therefore, public use of a modified version, on a publicly accessible server, gives the public access to the source code of the modified version.

An older license, called the Affero General Public License and published by Affero, was designed to accomplish similar goals. This is a different license, not a version of the Affero GPL, but Affero has released a new version of the Affero GPL which permits relicensing under this license.

The precise terms and conditions for copying, distribution and modification follow.

Terms and conditions

0. Definitions. "This License" refers to version 3 of the GNU Affero General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code. The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions. All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law. No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies. You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions. You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a: The work must carry prominent notices stating that you modified it, and giving a relevant date.

b: The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c: You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d: If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms. You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a: Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b: Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c: Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d: Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
e: Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d. A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms. "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in

certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a: Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b: Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c: Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d: Limiting the use for publicity purposes of names of licensors or authors of the material; or

e: Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f: Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination. You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies. You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients. Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents. A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom. If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Remote Network Interaction; Use with the GNU General Public License. Notwithstanding any other provision of this License, if you modify the Program, your modified version must prominently offer all users interacting with it remotely through a computer network (if your version supports such interaction) an opportunity to receive the Corresponding Source of your version by providing access to the Corresponding Source from a network server at no charge, through some standard or customary means of facilitating copying of software. This Corresponding Source shall include the Corresponding Source for any work covered by version 3 of the GNU General Public License that is incorporated pursuant to the following paragraph.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the work with which it is combined will remain governed by version 3 of the GNU General Public License.

14. Revised Versions of this License. The Free Software Foundation may publish revised and/or new versions of the GNU Affero General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU Affero General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU Affero General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU Affero General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty. THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE

COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16. If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.:

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU Affero General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If your software can interact with users remotely through a computer network, you should also make sure that it provides a way for users to get its source. For example, if your program is a web application, its interface could display a "Source" link that leads users to an archive of the code. There are many ways you could offer source, and different solutions will be better for different programs; see section 13 for the specific requirements.

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU AGPL, see http://www.gnu.org/licenses/.