

格子ボルツマン法オープンソース Palabos を使用した NAFEMS Benchmarks の計算

川畑真一 [1†]

[1] オープン CAE 勉強会@関西

Lattice Boltzmann method Calculation of NAFEMS Benchmarks using open source Palabos

Shinichi KAWABATA[*†]

[*]OpenCAE Local User Group @ Kansai

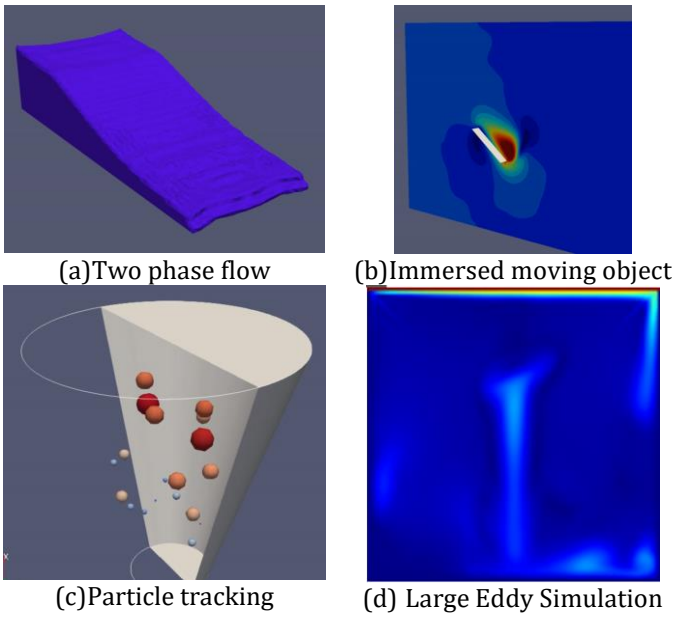
Abstract

Palabos is a library of open source lattice Boltzmann method. Palabos provides a many sample codes and it will be helpful to learn how to create two-phase flow, particle tracking, and LES (Large Eddy Simulation) solvers. In this report, we report the accuracy of the solver using Palabos as a result of checking the verification for NAFEMS Benchmarks.

Keywords: Palabos, Lattice Boltzmann method, NAFEMS Benchmarks

1. はじめに

格子ボルツマン法オープンソースコード"palabos"は FlowKit 社が公開しているライブラリである。Palabos ライブラリを使用することでソルバをより簡単に作成できる。また、Palabos は多数のサンプルコードを提供しており、二相流、粒子追跡、LES（Large Eddy Simulation）のソルバを作成する方法について参考になる。図 1 にサンプルコードのイメージを示す。本発表では Palabos ライブラリで作成したソルバを使用して、NAFEMS Benchmarks の問題を計算し、計算結果の精度について確認した結果を報告する。



(a)Two phase flow (b)Immersed moving object

(c)Particle tracking (d) Large Eddy Simulation

Fig. 1 Image of sample code

† *E-mail address of corresponding author: hammamania@gmail.com*

2. NAFEMS Benchmarks について

NAFEMS Benchmarks は NAFEMS から発行されている計算コードの検証に使用できる問題集となっており、商用コードの検証に使用されている。本発表では流体計算向けの"NAFEMS Computational Fluid Dynamics Benchmarks - Volume 1"[1]を対象に Palabos の計算結果の妥当性を検証した。

3. 検証問題

検証問題として“Laminar, Isothermal Backward Facing Step Benchmark”を使用した。モデルを図 2 に示す。この問題ではステップの後ろの流れに注目しており、しばしば流体コードの検証に使用されるバックステップ流れである。NAFEMS の例題集にはモデル寸法と合わせて物性、計算条件も記載されており、それら元に計算を実行する。

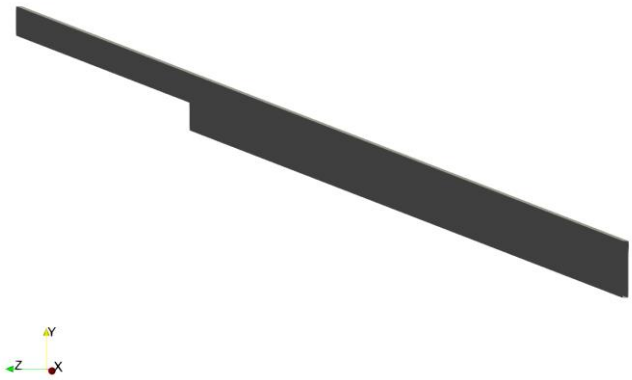


Fig. 2 Analysis model

4. 使用したソルバについて

計算は aneurysm サンプルコードを改造したソルバを使用した。aneurysm コードは動脈瘤の流れを計算するサンプルコードで STL ファイルを形状データとして使用できる。また、aneurysm コードは粗いメッシュから細かいメッシュに計算結果をマッピングして計算時間を短縮する仕組みが内包されており、形状定義および計算手法について汎用的な内容のコードとなっている。ただし、計算結果の出力が限定的であるため、計算領域全体の結果を出力するようにコードを改修した。コードは付録 A に記載した。

5. まとめ

Palabos のライブラリを使用したソルバに対して検証計算を実施した。今後も検証を実施しながらソルバを改修し、汎用的な計算ソルバの開発を実施していきたい。

参考文献

[1] A. Horvat. NAFEMS Computational Fluid Dynamics Benchmarks – Volume 1. NAFEMS. 2017. https://www.nafems.org/publications/browse_buy/browse_by_topic/cfd/r0123-nafems-computational-fluid-dynamics-benchmarks-volume-1/, (accessed 2018-11-20).
[2] FlowKit Ltd.. Palabos documentation. 2011. http://www.palabos.org/documentation/userguide/#, (accessed 2018-11-20).
[3] 蔦原道久, 高田尚樹, 片岡武. 格子気体法・格子ボルツマン法 - 新しい数値流体力学の手法 FD 付き -. コロナ社. 1999. https://ci.nii.ac.jp/ncid/BA42543301, (accessed 2018-11-20).

付録 A 作成したコード

Code 1 Created code

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



```
3
4   using namespace plb;
5   using namespace std;
6
7   typedef double T;
8   typedef Array<T,3> Velocity;
9   #define DESCRIPTOR descriptors::D3Q19Descriptor
10
11  plint extraLayer     = 0;
12
13  const plint blockSize = 20;
14  const plint envelopeWidth = 1;
15  const plint extendedEnvelopeWidth = 2;
16
17  bool performOutput = false;
18  bool doImages = false;
19  bool useAllDirections = false;
20  bool useRegularizedWall = false;
21  bool useIncompressible = false;
22  bool poiseuilleInlet = false;
23  bool convectiveScaling = false;
24
25  T kinematicViscosity      = 0.;
26  T averageInletVelocity    = 0.;
27  plint referenceResolution  = 0;
28  T nuLB                   = 0.;
29  T fluidDensity            = 0.;
30  T volume                 = 0.;
31  T userDefinedInletDiameter = 0.;
32
33  plint referenceDirection = 0;
34  plint openingSortDirection = 0;
35
36  T simTime = 0;
37  plint startLevel = 0;
38  plint maxLevel   = 0;
39  T epsilon = 0.;
40
41  TriangleSet<T>* triangleSet = 0;
42  T currentTime = 0;
43
44  template<typename T>
45  struct Opening {
46      bool inlet;
47      Array<T,3> center;
48      T innerRadius;
49  };
50
51  std::vector<Opening<T> > openings;
52
53  void iniLattice( MultiBlockLattice3D<T,DESCRIPTOR>& lattice,
54                   VoxelizedDomain3D<T>& voxelizedDomain )
55  {
56      defineDynamics(lattice, voxelizedDomain.getVoxelMatrix(), lattice.getBoundingBox(),
57                     new NoDynamics<T,DESCRIPTOR>, voxelFlag::outside);
58      initializeAtEquilibrium(lattice, lattice.getBoundingBox(), (T) 1., Array<T,3>((T) 0.,(T) 0.,(T) 0.));
59      lattice.initialize();
60  }
61
62  void setOpenings (
63      std::vector<BoundaryProfile3D<T,Velocity>*>& inletOutlets,
64      TriangleBoundary3D<T>& boundary, T uLB, T dx, T dt )
65  {
66      for (pluint i=0; i<openings.size(); ++i) {
67          Opening<T>& opening = openings[i];
68          opening.center = computeBaryCenter (
69                 boundary.getMesh(),
70                 boundary.getInletOutlet(openingSortDirection)[i] );
71          opening.innerRadius = computeInnerRadius (
72                 boundary.getMesh(),
73                 boundary.getInletOutlet(openingSortDirection)[i] );
74
```



```
75          if (opening.inlet) {
76              if (poiseuilleInlet) {
77                  inletOutlets.push_back (
78                          new PoiseuilleProfile3D<T>(uLB) );
79              }
80              else {
81                  inletOutlets.push_back (
82                          new VelocityPlugProfile3D<T>(uLB) );
83              }
84          }
85          else {
86              inletOutlets.push_back (
87                     new DensityNeumannBoundaryProfile3D<T> );
88          }
89      }
90  }
91
92  std::vector<T> pointMeasures (
93          MultiBlockLattice3D<T,DESCRIPTOR>& lattice,
94          Array<T,3> location, T dx, T dt )
95  {
96      std::vector<Array<T,3> > physicalPositions, positions;
97      physicalPositions.push_back(Array<T,3>(0.022046, 0.015072, 0.044152));
98      physicalPositions.push_back(Array<T,3>(0.027132, 0.049947, 0.095012));
99      physicalPositions.push_back(Array<T,3>(0.034398, 0.056487, 0.057957));
100     physicalPositions.push_back(Array<T,3>(0.031492, 0.025971, 0.084113));
101     physicalPositions.push_back(Array<T,3>(0.025679, 0.025971, 0.091379));
102     physicalPositions.push_back(Array<T,3>(0.018413, 0.011439, 0.076848));
103     positions.resize(physicalPositions.size());
104
105     for (pluint i=0; i<physicalPositions.size(); ++i) {
106         positions[i] = (physicalPositions[i]-location)/dx;
107     }
108
109     std::vector<Array<T,3> > velocities = velocitySingleProbes(lattice, positions);
110     std::vector<Array<T,3> > vorticities = vorticitySingleProbes(lattice, positions);
111     std::vector<T> densities = densitySingleProbes(lattice, positions);
112
113     std::vector<T> data;
114     for (pluint i=0; i<physicalPositions.size(); ++i) {
115         Array<T,3> pos = physicalPositions[i];
116         Array<T,3> vel = velocities[i]*dx/dt;
117         Array<T,3> vort = vorticities[i]/dt;
118         T pressure = DESCRIPTOR<T>::cs2*(densities[i]-1.)*dx*dx/(dt*dt)*fluidDensity;
119         if (performOutput) {
120             pcout << "Pos ("
121                   << pos[0] << "," << pos[1] << "," << pos[2]
122                   << "); Velocity ("
123                   << vel[0] << "," << vel[1] << "," << vel[2]
124                   << "); Vorticity ("
125                   << vort[0] << "," << vort[1] << "," << vort[2]
126                   << "); Pressure " << pressure << std::endl;
127         }
128         data.push_back(norm(vel));
129         data.push_back(norm(vort));
130         data.push_back(pressure);
131     }
132     return data;
133 }
134
135 void writeImages (
136          OffLatticeBoundaryCondition3D<T,DESCRIPTOR,Velocity>& boundaryCondition,
137          Box3D const& imageDomain, Box3D const& vtkDomain, std::string fname, Array<T,3> location, T dx, T
    dt )
138 {
139     VtkImageOutput3D<T> vtkOut(fname, dx, location);
140     vtkOut.writeData<float>(*boundaryCondition.computePressure(vtkDomain), "p",
    util::sqr(dx/dt)*fluidDensity);
141     vtkOut.writeData<float>(*boundaryCondition.computeVelocityNorm(vtkDomain), "u", dx/dt);
142
    vtkOut.writeData<float>(*copyConvert<int,T>(*extractSubDomain(boundaryCondition.getVoxelizedDomain().get
    VoxelMatrix(),vtkDomain)), "voxel", 1.);
```



```
143
144     ImageWriter<T> imageWriter("leeloo");
145     imageWriter.writeScaledPpm(fname, *boundaryCondition.computeVelocityNorm(imageDomain));
146 }
147
148 void writeImages (
149         OffLatticeBoundaryCondition3D<T,DESCRIPTOR,Velocity>& boundaryCondition, plint level, Array<T,3>
    location, T dx, T dt )
150 {
151     plint nx = boundaryCondition.getLattice().getNx();
152     plint ny = boundaryCondition.getLattice().getNy();
153     plint nz = boundaryCondition.getLattice().getNz();
154
155     Array<T,3> yz_plane(0.016960, 0.032604, 0.057772);
156     Array<T,3> xz_plane(0.026725, 0.017978, 0.057772);
157     Array<T,3> xy_plane(0.026725, 0.032604, 0.084113);
158
159     Array<T,3> lyz_plane((yz_plane-location)/dx);
160     Array<T,3> lxz_plane((xz_plane-location)/dx);
161     Array<T,3> lxy_plane((xy_plane-location)/dx);
162
163     Box3D yz_imageDomain (
164            util::roundToInt(lyz_plane[0]), util::roundToInt(lyz_plane[0]),
165            0, ny-1, 0, nz-1 );
166     Box3D xz_imageDomain (
167            0, nx-1,
168            util::roundToInt(lxz_plane[1]), util::roundToInt(lxz_plane[1]),
169            0, nz-1 );
170     Box3D xy_imageDomain (
171            0, nx-1, 0, ny-1,
172            util::roundToInt(lxy_plane[2]), util::roundToInt(lxy_plane[2]) );
173
174     Box3D yz_vtkDomain (
175            util::roundToInt(lyz_plane[0])-3, util::roundToInt(lyz_plane[0])+3,
176            0, ny-1, 0, nz-1 );
177     Box3D xz_vtkDomain (
178            0, nx-1,
179            util::roundToInt(lxz_plane[1])-3, util::roundToInt(lxz_plane[1])+3,
180            0, nz-1 );
181     Box3D xy_vtkDomain (
182            0, nx-1, 0, ny-1,
183            util::roundToInt(lxy_plane[2])-3, util::roundToInt(lxy_plane[2])+3 );
184
185     writeImages(boundaryCondition, xy_imageDomain, xy_vtkDomain, "xy_"+util::val2str(level), location, dx,
    dt);
186     writeImages(boundaryCondition, xz_imageDomain, xz_vtkDomain, "xz_"+util::val2str(level), location, dx,
    dt);
187     writeImages(boundaryCondition, yz_imageDomain, yz_vtkDomain, "yz_"+util::val2str(level), location, dx,
    dt);
188 }

189 template<class BlockLatticeT>

190 // Add
191 void writeVTK(BlockLatticeT& lattice,
192              T dx, T dt,
193              plint level)
194 {
195     ParallelVtkImageOutput3D<T> vtkOut(createFileName("vtk", level, 6), 3, dx);
196
197     vtkOut.writeData<float>(*computeDensity(lattice), "density", 1.);
198     vtkOut.writeData<3,float>(*computeVelocity(lattice), "velocity", dx/dt);
199     vtkOut.writeData<float>(*computeVelocityNorm(lattice), "velocityNorm", dx/dt);
200 }
201 // -----
202
203 std::auto_ptr<MultiBlockLattice3D<T,DESCRIPTOR> > run (
204        plint level, MultiBlockLattice3D<T,DESCRIPTOR>* iniVal=0 )
205 {
206     plint margin = 3;
207     plint borderWidth = 1;
208
```



```
209     plint resolution = referenceResolution * util::twoToThePower(level);
210
211     DEFscaledMesh<T>* defMesh =
212         new DEFscaledMesh<T>(*triangleSet, resolution, referenceDirection, margin, extraLayer);
213     TriangleBoundary3D<T> boundary(*defMesh);
214     delete defMesh;
215     boundary.getMesh().inflate();
216
217     T nuLB_ = nuLB;
218     if (convectiveScaling) {
219         nuLB_ = nuLB * util::twoToThePower(level);
220     }
221     T dx = boundary.getDx();
222     T dt = nuLB_ / kinematicViscosity *dx*dx;
223     T uAveLB = averageInletVelocity *dt/dx;
224     T omega = 1./(3.*nuLB_+0.5);
225     Array<T,3> location(boundary.getPhysicalLocation());
226
227
228     pcout << "uLB=" << uAveLB << std::endl;
229     pcout << "nuLB=" << nuLB_ << std::endl;
230     pcout << "tau=" << 1./omega << std::endl;
231     if (performOutput) {
232         pcout << "dx=" << dx << std::endl;
233         pcout << "dt=" << dt << std::endl;
234     }
235
236     std::vector<BoundaryProfile3D<T,Velocity>*> inletOutlets;
237     setOpenings(inletOutlets, boundary, uAveLB, dx, dt);
238     Array<T,3> inletCenter(0.0, 0.0, 0.0);
239     for (pluint i=0; i<openings.size(); ++i) {
240         if (openings[i].inlet) {
241             pcout << "Inner radius of inlet " << i << " : "
242                   << openings[i].innerRadius << " lattice nodes" << std::endl;
243             inletCenter=openings[i].center;
244         }
245     }
246     T inletZpos = util::roundToInt(inletCenter[2])+1;
247     BoundaryProfiles3D<T,Velocity> profiles;
248     profiles.defineInletOutletTags(boundary, openingSortDirection);
249     profiles.setInletOutlet(inletOutlets);
250
251     const int flowType = voxelFlag::inside;
252     VoxelizedDomain3D<T> voxelizedDomain (
253             boundary, flowType, extraLayer, borderWidth, extendedEnvelopeWidth, blockSize );
254     if (performOutput) {
255         pcout << getMultiBlockInfo(voxelizedDomain.getVoxelMatrix()) << std::endl;
256     }
257
258     MultiScalarField3D<int> flagMatrix((MultiBlock3D&)voxelizedDomain.getVoxelMatrix());
259     setToConstant(flagMatrix, voxelizedDomain.getVoxelMatrix(),
260                   voxelFlag::inside, flagMatrix.getBoundingBox(), 1);
261     setToConstant(flagMatrix, voxelizedDomain.getVoxelMatrix(),
262                   voxelFlag::innerBorder, flagMatrix.getBoundingBox(), 1);
263     pcout << "Number of fluid cells: " << computeSum(flagMatrix) << std::endl;
264
265     Dynamics<T,DESCRIPTOR>* dynamics = 0;
266     if (useIncompressible) {
267         dynamics = new IncBGKdynamics<T,DESCRIPTOR>(omega);
268     }
269     else {
270         dynamics = new BGKdynamics<T,DESCRIPTOR>(omega);
271     }
272     std::auto_ptr<MultiBlockLattice3D<T,DESCRIPTOR> > lattice
273         = generateMultiBlockLattice<T,DESCRIPTOR> (
274                 voxelizedDomain.getVoxelMatrix(), envelopeWidth, dynamics );
275     lattice->toggleInternalStatistics(false);
276
277     std::vector<MultiBlock3D*> rhoBarJarg;
278     plint numScalars = 4;
279     MultiNTensorField3D<T>* rhoBarJfield =
280          generateMultiNTensorField3D<T>(*lattice, extendedEnvelopeWidth, numScalars);
```



```
281     rhoBarJfield->toggleInternalStatistics(false);
282     rhoBarJarg.push_back(rhoBarJfield);
283     plint processorLevel=0;
284     integrateProcessingFunctional (
285            new PackedRhoBarJfunctional3D<T,DESCRIPTOR>(),
286            lattice->getBoundingBox(), *lattice, *rhoBarJfield, processorLevel );
287
288     GuoOffLatticeModel3D<T,DESCRIPTOR>* model =
289            new GuoOffLatticeModel3D<T,DESCRIPTOR> (
290                new TriangleFlowShape3D<T,Array<T,3> > (
291                    voxelizedDomain.getBoundary(), profiles),
292                flowType, useAllDirections );
293     model->setVelIsJ(useIncompressible);
294     model->selectUseRegularizedModel(useRegularizedWall);
295     model->selectComputeStat(false);
296     OffLatticeBoundaryCondition3D<T,DESCRIPTOR,Velocity> boundaryCondition (
297            model, voxelizedDomain, *lattice);
298     boundaryCondition.insert(rhoBarJarg);
299
300     iniLattice(*lattice, voxelizedDomain);
301     if(iniVal) {
302         Box3D toDomain(lattice->getBoundingBox());
303         Box3D fromDomain(toDomain.shift(margin,margin,margin));
304         copy(*iniVal, fromDomain, *lattice, toDomain, modif::staticVariables);
305         computePackedRhoBarJ(*lattice, *rhoBarJfield, lattice->getBoundingBox());
306         boundaryCondition.apply(rhoBarJarg);
307     }
308
309     plint convergenceIter=20;
310     util::ValueTracer<T> velocityTracer(0.05*convergenceIter, resolution, epsilon);
311     global::timer("iteration").restart();
312     plint i = util::roundToInt(currentTime/dt);
313     lattice->resetTime(i);
314     bool checkForErrors = true;
315
316     while(!velocityTracer.hasConverged() && currentTime<simTime)
317     {
318         if (i%200==0 && performOutput) {
319             pcout << "T= " << currentTime << "; "
320                   << "Average energy: "
321                   << boundaryCondition.computeAverageEnergy()*util::sqr(dx/dt) << std::endl;
322         }
323         if (i%convergenceIter==0) {
324             velocityTracer.takeValue(computeAverageEnergy(*lattice));
325         }
326
327         lattice->collideAndStream();
328         if (checkForErrors) {
329             abortIfErrorsOccurred();
330             checkForErrors = false;
331         }
332
333         ++i;
334         currentTime = i*dt;
335     }
336     delete rhoBarJfield;
337
338     Box3D measureBox(lattice->getBoundingBox());
339     measureBox.z0=measureBox.z1=(plint)inletZpos;
340     T inletPressure = DESCRIPTOR<T>::cs2*(boundaryCondition.computeAverageDensity(measureBox)-1.);
341
342     if (doImages) {
343         writeImages(boundaryCondition, level, location, dx, dt);
344         std::vector<std::string> scalarNames;
345         scalarNames.push_back("pressure");
346         scalarNames.push_back("wss");
347         std::vector<T> scalarFactor;
348         scalarFactor.push_back(util::sqr(dx/dt)*fluidDensity);
349         scalarFactor.push_back(util::sqr(dx/dt)*fluidDensity);
350         std::vector<std::string> vectorNames;
351         vectorNames.push_back("force");
352         std::vector<T> vectorFactor;
```



```
353         vectorFactor.push_back(util::sqr(dx/dt)*fluidDensity);
354         bool dynamicMesh = false;
355         writeSurfaceVTK (
356                 boundary,
357                 *computeSurfaceForce( boundary, voxelizedDomain, *lattice, model->velIsJ(), dynamicMesh ),
358                 scalarNames, vectorNames, "surface_"+util::val2str(level)+".vtk", dynamicMesh, 0,
359                 scalarFactor, vectorFactor );
360         // Add
361         writeVTK(*lattice, dx, dt, level);
362         // -----
363     }
364
365     T averageEnergy = boundaryCondition.computeAverageEnergy()*util::sqr(dx/dt);
366     T rmsVorticity  = boundaryCondition.computeRMSvorticity()/dt;
367     T pressureDrop = inletPressure*util::sqr(dx/dt)*fluidDensity;
368     T inletAverageVel = boundaryCondition.computeAverageVelocityComponent(measureBox,2)*dx/dt;
369
370     if (performOutput) {
371         pcout << "Average energy: " << averageEnergy << std::endl;
372         pcout << "Total energy: " << averageEnergy*volume << std::endl;
373         pcout << "RMS vorticity * volume * 0.5: " << rmsVorticity*0.5*volume << std::endl;
374         pcout << "Pressure drop: " << pressureDrop << std::endl;
375         pcout << "Average velocity through inlet section: " << inletAverageVel << std::endl;
376         pcout << "Number of iterations: " << i << std::endl;
377     }
378     pcout << "Elapsed time: " << global::timer("iteration").stop() << std::endl;
379     pcout << "Total elapsed time: " << global::timer("global").getTime() << std::endl;
380
381     if (performOutput) {
382         pcout << "Description: "
383               << "Tot. energy, pressure-drop, tot. enstrophy,"
384               << "  vel1, vort1, pres1,  vel2, vort2, pres2,"
385               << "  vel3, vort3, pres3,  vel4, vort4, pres4,"
386               << "  vel5, vort5, pres5,  vel6, vort6, pres6" << std::endl;
387         pcout << "All data: ";
388     }
389     pcout << averageEnergy*volume << ", " << pressureDrop << ", " << rmsVorticity*volume*0.5 << ", ";
390     std::vector<T> pointData = pointMeasures(*lattice, location, dx, dt);
391     for (pluint i=0; i<pointData.size(); ++i) {
392         pcout << pointData[i];
393         if (i!=pointData.size()-1) {
394             pcout << ", ";
395         }
396     }
397     pcout << std::endl;
398
399     return lattice;
400 }
401
402 void readParameters(XMLreader const& document)
403 {
404     std::string meshFileName;
405     std::vector<std::string> openingType;
406     document["geometry"]["mesh"].read(meshFileName);
407     document["geometry"]["inletDiameter"].read(userDefinedInletDiameter);
408     document["geometry"]["averageInletVelocity"].read(averageInletVelocity);
409     document["geometry"]["openings"]["sortDirection"].read(openingSortDirection);
410     document["geometry"]["openings"]["type"].read(openingType);
411
412     document["fluid"]["kinematicViscosity"].read(kinematicViscosity);
413     document["fluid"]["density"].read(fluidDensity);
414     document["fluid"]["volume"].read(volume);
415
416     document["numerics"]["referenceDirection"].read(referenceDirection);
417     document["numerics"]["referenceResolution"].read(referenceResolution);
418     document["numerics"]["nuLB"].read(nuLB);
419
420     document["simulation"]["simTime"].read(simTime);
421     document["simulation"]["maxLevel"].read(maxLevel);
422     document["simulation"]["epsilon"].read(epsilon);
423
424     document["simulation"]["performOutput"].read(performOutput);
```



```
425     document["simulation"]["doImages"].read(doImages);
426     document["simulation"]["useAllDirections"].read(useAllDirections);
427     document["simulation"]["useRegularizedWall"].read(useRegularizedWall);
428     document["simulation"]["useIncompressible"].read(useIncompressible);
429     document["simulation"]["poiseuilleInlet"].read(poiseuilleInlet);
430     document["simulation"]["convectiveScaling"].read(convectiveScaling);
431
432     triangleSet = new TriangleSet<T>(meshFileName, DBL);
433     pcout << "Reynolds number, based on provided inlet diameter: "
434           << averageInletVelocity*userDefinedInletDiameter/kinematicViscosity
435           << std::endl;
436     plbIOError(openingSortDirection<0 || openingSortDirection>2,
437                "Sort-direction of opening must be 0 (x), 1 (y), or 2 (z).");
438     openings.resize(openingType.size());
439     for (pluint i=0; i<openingType.size(); ++i) {
440         std::string next_opening = util::tolower(openingType[i]);
441         if (next_opening=="inlet") {
442             openings[i].inlet = true;
443         }
444         else if (next_opening=="outlet") {
445             openings[i].inlet = false;
446         }
447         else {
448             plbIOError("Unknown opening type.");
449         }
450     }
451 }
452
453
454 int main(int argc, char* argv[])
455 {
456     plbInit(&argc, &argv);
457     global::directories().setOutputDir("./");
458     global::IOpolicy().activateParallelIO(false);
459
460     string paramXmlFileName;
461     try {
462         global::argv(1).read(paramXmlFileName);
463     }
464     catch (PlbIOException& exception) {
465         pcout << "Wrong parameters; the syntax is: "
466               << (std::string)global::argv(0) << " parameter-input-file.xml" << std::endl;
467         return -1;
468     }
469
470
471     try {
472         XMLreader document(paramXmlFileName);
473         readParameters(paramXmlFileName);
474     }
475     catch (PlbIOException& exception) {
476         pcout << "Error in input file " << paramXmlFileName
477               << ": " << exception.what() << std::endl;
478         return -1;
479     }
480
481     global::timer("global").start();
482     plint iniLevel=0;
483     std::auto_ptr<MultiBlockLattice3D<T,DESCRIPTOR> > iniConditionLattice(0);
484     try {
485         for (plint level=iniLevel; level<=maxLevel; ++level) {
486             pcout << std::endl << "Running new simulation at level " << level << std::endl;
487             std::auto_ptr<MultiBlockLattice3D<T,DESCRIPTOR> > convergedLattice (
488                    run(level, iniConditionLattice.get()) );
489             if (level != maxLevel) {
490                 plint dxScale = -1;
491                 plint dtScale = -2;
492                 if (convectiveScaling) {
493                     dtScale = -1;
494                 }
495                 iniConditionLattice = std::auto_ptr<MultiBlockLattice3D<T,DESCRIPTOR> > (
496                        refine(*convergedLattice, dxScale, dtScale, new BGKdynamics<T,DESCRIPTOR>(1.)) );
```



```
497             }
498
499         }
500     }
501     catch(PlbException& exception) {
502         pcout << exception.what() << std::endl;
503         return -1;
504     }
505 }
```