Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

Las Palmeras Molecular Dynamics: A flexible and modular molecular dynamics code [☆]

Sergio Davis [*], Claudia Loyola, Felipe González, Joaquín Peralta

Departamento de Física, Facultad de Ciencias, Universidad de Chile, Las Palmeras 3425, 780-0024 Ñuñoa, Santiago, Chile

A R T I C L E   I N F O

Article history:
Received 28 May 2010
Received in revised form 19 August 2010
Accepted 25 August 2010

Keywords:
Computer simulation
Molecular dynamics
Plug-ins
Visualization

A B S T R A C T

Las Palmeras Molecular Dynamics (LPMD) is a highly modular and extensible molecular dynamics (MD) code using interatomic potential functions. LPMD is able to perform equilibrium MD simulations of bulk crystalline solids, amorphous solids and liquids, as well as non-equilibrium MD (NEMD) simulations such as shock wave propagation, projectile impacts, cluster collisions, shearing, deformation under load, heat conduction, heterogeneous melting, among others, which involve unusual MD features like non-moving atoms and walls, unstoppable atoms with constant-velocity, and external forces like electric fields. LPMD is written in C++ as a compromise between efficiency and clarity of design, and its architecture is based on separate components or *plug-ins*, implemented as modules which are loaded on demand at runtime. The advantage of this architecture is the ability to completely link together the desired components involved in the simulation in different ways at runtime, using a user-friendly *control file* language which describes the simulation work-flow.

As an added bonus, the *plug-in* API (Application Programming Interface) makes it possible to use the LPMD components to analyze data coming from other simulation packages, convert between input file formats, apply different transformations to saved MD atomic trajectories, and visualize dynamical processes either in real-time or as a post-processing step.

Individual components, such as a new potential function, a new integrator, a new file format, new properties to calculate, new real-time visualizers, and even a new algorithm for handling neighbor lists can be easily coded, compiled and tested within LPMD by virtue of its object-oriented API, without the need to modify the rest of the code.

LPMD includes already several pair potential functions such as Lennard-Jones, Morse, Buckingham, MCY and the harmonic potential, as well as embedded-atom model (EAM) functions such as the Sutton–Chen and Gupta potentials. Integrators to choose include Euler (if only for demonstration purposes), Verlet and Velocity Verlet, Leapfrog and Beeman, among others. Electrostatic forces are treated as another potential function, by default using the *plug-in* implementing the Ewald summation method.

Program summary

Program title: LPMD
Catalogue identifier: AEHG_v1_0
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AEHG_v1_0.html
Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland
Licensing provisions: GNU General Public License version 3
No. of lines in distributed program, including test data, etc.: 509 490
No. of bytes in distributed program, including test data, etc.: 6 814 754
Distribution format: tar.gz
Programming language: C++
Computer: 32-bit and 64-bit workstation
Operating system: UNIX
RAM: Minimum 1024 bytes
Classification: 7.7
External routines: zlib, OpenGL

☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author.
E-mail address: sdavis@gnm.cl (S. Davis).

doi:10.1016/j.cpc.2010.08.030

Nature of problem: Study of Statistical Mechanics and Thermodynamics of condensed matter systems, as well as kinetics of non-equilibrium processes in the same systems.
Solution method: Equilibrium and non-equilibrium molecular dynamics method, Monte Carlo methods.
Restrictions: Rigid molecules are not supported. Polarizable atoms and chemical bonds (proteins) either.
Unusual features: The program is able to change the temperature of the simulation cell, the pressure, cut regions of the cell, color the atoms by properties, even during the simulation. It is also possible to fix the positions and/or velocity of groups of atoms. Visualization of atoms and some physical properties during the simulation.
Additional comments: The program does not only perform molecular dynamics and Monte Carlo simulations, it is also able to filter and manipulate atomic configurations, read and write different file formats, convert between them, evaluate different structural and dynamical properties.
Running time: 50 seconds on a 1000-step simulation of 4000 argon atoms, running on a single 2.67 GHz Intel processor.

1. Introduction

Classical Molecular dynamics (MD) simulation has been a powerful tool since their development in 1960s to study dynamic systems of interacting atoms in various fields, like chemistry, physics and material science. The molecular dynamics method consists in the numerical integration of the Newton's equation of motion for the atomic positions in a condensed matter system. In order to compute the forces needed to obtain the atomic velocities (which in turn are needed to obtain the atomic positions), we need some kind of interatomic potential describing the interaction between different atoms. Fully empirical and also semi-empirical interatomic potentials have been used in the last decade to successfully represent different kinds of materials, from inert gases [1] to metals [2], carbon-based molecules and structures, crystalline and amorphous [3,4] metallic oxides, among others. The advantage of the classical molecular dynamics method, compared with ab initio methods, which are more exact, is the ability to handle large numbers of atoms. In a typical simulation, hundred of thousands or even several million atoms can be treated [5,6].

Although there are many general purpose MD codes, they are usually subjected to design limitations (arising mostly due to efficiency considerations) that allows only the study of certain systems and conditions. Most codes cannot handle in an easy way the requirements of some setups, such as free (i.e., non-periodic) boundary conditions, variations of density inside a sample, or highly out-of-equilibrium initial states. It might be possible to modify these codes to lift some of the limitations, but it could be cumbersome and error-prone. For these cases, a more flexible MD code is needed, even though some performance could be sacrificed.

In this spirit, we have developed *Las Palmeras Molecular Dynamics* (LPMD), a completely modular MD program. This program consists of a set of replaceable pieces which can be linked together in different ways to accommodate the needs of a non-standard MD simulation. Beyond that, the user can also perform post-simulation analysis, convert between input/output formats and prepare samples with ease. The modular design also improves efficiency in some cases. It also allows the user to add new pieces (integration methods, interatomic potentials, properties, file formats, and many others) without the need of knowing the whole code.

2. Theoretical background

As mentioned before, the foundation of the method is taking Newton's equations of motion for each atom i,

$$m_i \frac{d^2}{dt^2}\vec{r_i} = \vec{F} = m_i\vec{a_i}, \tag{1}$$

where the force $\vec{F_i}$ is given by the gradient of the interatomic potential function $V(\vec{r})$ with respect to the atom coordinate $\vec{r_i}$,

$$\vec{F_i} = -\nabla_i V(\vec{r_1}, \vec{r_2}, \ldots, \vec{r_N}). \tag{2}$$

The potential function can be chosen from different alternative functional forms depending on the complexity of the interactions in the real material. For the case where the interatomic interactions are simple enough to be modelled by a pair potential function $\phi(r)$, which depends only on the distance r between a pair of atoms, we have

$$V(r_{ij}) = \frac{1}{2}\sum_{i=1}^{N}\sum_{j\neq i}\phi\big(|\vec{r_i} - \vec{r_j}|\big), \tag{3}$$

where $r_{ij} = |\vec{r_i} - \vec{r_j}|$. In this case, the force on each atom is given by

$$\vec{F_i} = -\sum_{j\neq i}\frac{d\phi}{dr_{ij}}\big(|\vec{r_i} - \vec{r_j}|\big)\frac{\vec{r_i} - \vec{r_j}}{|\vec{r_i} - \vec{r_j}|}. \tag{4}$$

The most typical pair potential function is the Lennard-Jones potential [7], defined by

$$\phi(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right], \tag{5}$$

where σ and ϵ are parameters fitted to match experimental properties of the desired material. For the case of Ar, $\sigma = 3.41$ Å and $\epsilon = 120\ \mathrm{K}\ k_B$.

More complex systems require not only pair interactions but may include three-body or four-body terms. Alternatively one can model the interactions of the atoms in a metal by including a "mean-field" background potential F associated with the freely moving electrons. This kind of model is known as the Embedded Atom Model (EAM) [8]. The general form of these potentials is

$$V(\vec{r_1}, \vec{r_2}, \ldots, \vec{r_N}) = \sum_{i=1}^{N}\left[F(\rho_i) + \frac{1}{2}\sum_{j\neq i}\phi\big(|\vec{r_i} - \vec{r_j}|\big)\right]. \tag{6}$$

One form of the EAM, called the Sutton–Chen [9] potential, further defines the functions $\phi(r)$ and the background potential $F(\rho)$ as

$$\phi(r) = \epsilon\left(\frac{a}{r}\right)^{n}, \tag{7}$$

$$F(\rho_i) = -\epsilon C\sqrt{\rho_i}, \tag{8}$$

where ρ_i represents the volumetric density of electrons acting on the atom i, and is also computed as a sum over pair contributions $\psi(r)$ due to surrounding atoms,

$$\rho_i = \sum_{j\neq i}\psi\big(|\vec{r_i} - \vec{r_j}|\big), \tag{9}$$

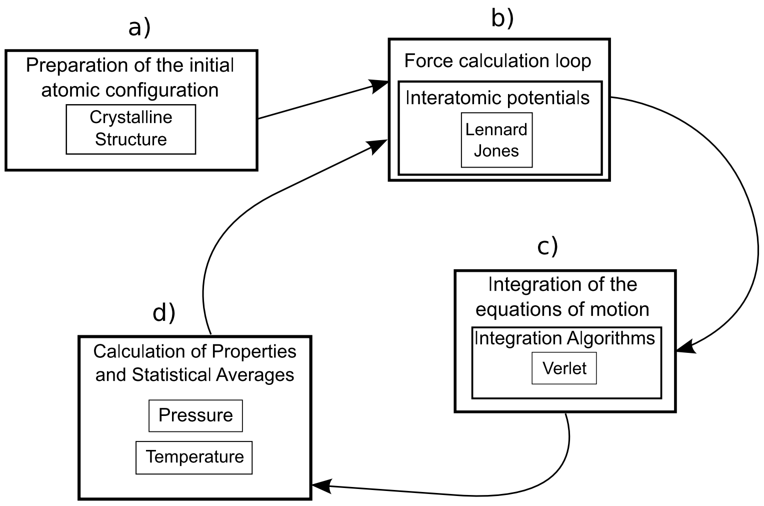
a)
Preparation of the initial atomic configuration
Crystalline Structure
b)
Force calculation loop
Interatomic potentials
Lennard Jones
c)
Integration of the equations of motion
Integration Algorithms
Verlet
d)
Calculation of Properties and Statistical Averages
Pressure
Temperature

Fig. 1. The execution flow for an MD simulation in a typical MD code.

and where

$$\psi(r) = \epsilon \left(\frac{a}{r}\right)^m. \tag{10}$$

Besides the choice of interatomic potentials, there are also many different numerical integration algorithms available for solving Eq. (1), with different precision and computational costs. One such integration procedure is the Verlet algorithm [1],

$$\vec{r}_i(t+\Delta t) = 2\vec{r}_i(t) - \vec{r}_i(t-\Delta t) + \vec{a}_i(t)\Delta t^2 + \mathcal{O}\big(\Delta t^4\big), \tag{11}$$

where Δt is the time step used for the discretization of the differential equation (1), and $\vec{a}_i = (1/m_i)\vec{F}_i$ is the acceleration "felt" by atom i.

Note that Eq. (11) only propagates the positions, the velocities are not explicitly propagated in time. Instead, one computes them indirectly from the centered-difference expression for the derivative of the coordinates,

$$\vec{v}_i(t) = \frac{\vec{r}_i(t+\Delta t) - \vec{r}_i(t-\Delta t)}{2\Delta t} + \mathcal{O}\big(\Delta t^2\big). \tag{12}$$

One improvement over the Verlet algorithm described above is the so-called Velocity Verlet algorithm [10], which improves the computation of velocities at the cost of requiring an additional update of the accelerations (and therefore the forces) at the step being calculated, $t + \Delta t$. The Velocity Verlet algorithm is implemented through the following equations,

$$\vec{r}_i(t+\Delta t) = \vec{r}_i(t) + \vec{v}_i(t)\Delta t + \frac{1}{2}\vec{a}_i(t)\Delta t^2 + \mathcal{O}\big(\Delta t^4\big), \tag{13}$$

$$\vec{v}_i(t+\Delta t) = \vec{v}_i(t) + \frac{1}{2}\vec{a}_i(t)\Delta t + \frac{1}{2}\vec{a}_i(t+\Delta t)\Delta t + \mathcal{O}\big(\Delta t^3\big). \tag{14}$$

Here we can see that, if the MD simulation does not require a high precision in the velocities, we could prefer the Verlet algorithm because it involves just one evaluation of the forces instead of two for the case of Velocity Verlet (conversely, to avoid evaluating forces twice it is possible to store the forces at the previous step, with an additional memory cost).

However, if the MD simulation involves the computation of quantities involving velocities such as the velocity autocorrelation function, then it is imperative to use an algorithm computing the velocities with higher precision.

The early way of doing MD was to implement a "taylor-made" computer program with precisely the chosen algorithms for numerical integration of the equations of motion and computation of the interatomic potential and forces. Thus, one different computer code for each system to be simulated.

The next stage in MD computer codes is the ability to choose the interatomic potential function at runtime (i.e., every time the program is executed, without the need to recompile for every change) along with all the other options such as the time step used for integration, total simulation time, initial conditions of pressure and temperature and so on. This has led to "general purpose" MD codes such as Moldy [11] and DLPOLY [12] among many others. While the ability to choose the potential function is commonplace nowadays, very few computer codes offer the choice of changing the integration algorithm at runtime, although several have the choice at compile-time (i.e., during the compilation stage).

From a general point of view, the MD procedure consists of four main stages, namely: (a) the initialization of the sample, (b) the calculation of interatomic forces, (c) the integration of the equations of motion, and (d) collecting statistics and the computation of properties. This is shown schematically in Fig. 1.

When the MD simulation that we intend to perform is not standard, for example in the case of simulations far away from thermodynamic equilibrium (high velocity impacts, shockwaves) or non-standard potential functions and forces (for example friction forces or external fields) one can clearly see the need for a hybrid approach between the "taylor-made" MD code (containing exactly the algorithms we need for a given simulation) and the "general purpose" MD code (with several choices available at runtime and compile-time). We would want to replace "pieces of the program" at will, including (but not limited to) integration methods, potential functions and other algorithms, such as the one responsible for computing interatomic distances or the "thermostat" algorithms used to control the applied temperature or pressure in an isothermal-isobaric (NPT) MD simulation. Here the "general purpose" approach is not general enough, only allowing some limited choices.

3. Overview of the software structure

Our design goal with LPMD is to have an MD code built only upon individual components, which could be rearranged at will to "assemble" many different kinds of MD simulations. This idea is not limited to the MD simulation itself, as the individual components could be reused for other purposes, such as post-simulation analysis of samples, or preparation of the initial configuration to be used as an input for an MD simulation. The proposed new design (which is, in fact, already implemented in LPMD) is shown schematically in Fig. 2.

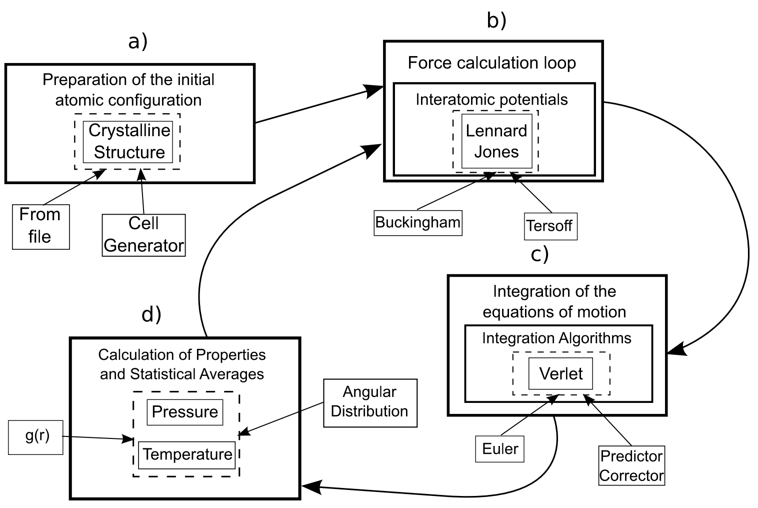

Fig. 2. The execution flow for an MD simulation as implemented in LPMD. Unlike Fig. 1, now the dashed lines represent a "placeholder" where many different pieces can fit.

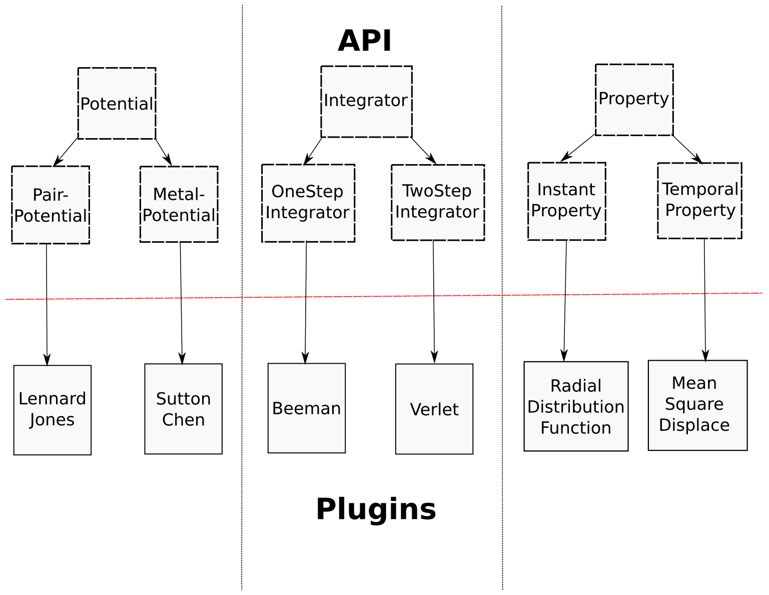

Fig. 3. Scheme showing some of the class hierarchies defined in the LPMD API.

We achieve this goal of interchangeable components using an object-oriented design. For this we designed and implemented a comprehensive API (Application Programming Interface), containing classes and interfaces representing each of the fundamental components in Fig. 2. This might be better explained by looking at the API hierarchies shown in Fig. 3. In this figure the solid-line boxes represent classes, i.e., specialized version of abstract ideas or interfaces (depicted by dashed-line boxes). As a concrete example, we can see that the `lennardjones` potential is a particular implementation of the abstract idea of a `pairpotential`, which in turn, is a specialization of the more general idea of a `potential`.

This hierarchic design ensures two things. First, all components that inherit from a common interface share common attributes. For instance, all pair potential components (such as `lennardjones` and `buckingham`) have a pair-defined energy $\phi(r)$ and a force $\vec{F}(\vec{r})$. Second, only the more specific details must be filled in when implementing a new component. Being also a potential (indirectly through pair potential), `lennardjones` can automatically compute system-wide energies Φ and atomic forces $\vec{F}_i$.

The design presented in Fig. 2 shows only the typical components in an MD execution flow. In order to have a finer control over the simulation process, we have extended this design, creating additional components and hierarchies. Among the new components, perhaps the one deserving a detailed explanation, is the *CellManager*, introduced to decouple the computation of distances from the force loop.

3.1. CellManagers

One of the most computationally expensive stages (if not the most) in the MD procedure is the calculation of interatomic forces. Several algorithms have been proposed in order to handle this task (from the standard minimum image loop to more advanced techniques like the Verlet neighbors list and the link-cell algorithm) and, in analogy with the different integration algorithms or potential functions, we have decided to encapsulate the task of computing the interatomic distances and building neighbor lists in a separate component, namely, the *cellmanager*. The advantage of making the *cellmanager* an interchangeable component is flexibility, as the different algorithms may perform with different degrees of efficiency. For example, the simple minimum image loop is more efficient than the more complex algorithms in the case of very small (i.e., less than two hundred atoms) systems, but the trend reverses around one thousand atoms [13].

All the available *cellmanagers* are shown in Fig. 4.

4. Implementation

LPMD was implemented in the C++ language, taking full advantage of its object-oriented capabilities: a hierarchy of classes and interfaces and polymorphism through virtual methods and templates. The interchangeable components are implemented as *plug-ins*, which are in fact dynamic libraries loaded at runtime. The API is implemented as a shared library, providing the base classes and interfaces which every *plug-in* must implement, as well as helper functions used through the code. The applications are the implementation of the diagram depicted in Fig. 2 and provide a user-friendly end-user interface to run simulations, obtain properties and apply transformations over sets of atomic configurations.

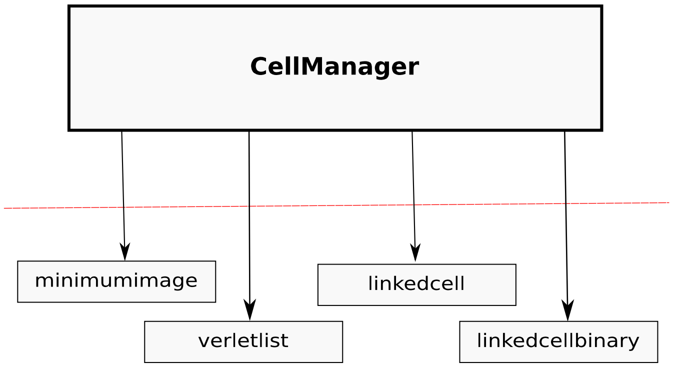

Fig. 4. Scheme showing the different *cellmanagers* available.

4.1. API

The API is composed of three types of elements: (a) base classes providing reusable basic components like Array, Matrix, and Vector, among others, (b) abstract interfaces providing generic algorithms via virtual methods, which derived classes must implement, and (c) utility or *helper* functions, which encapsulate routine tasks such as string processing. An example of the use of abstract interfaces is shown in Figs. 5 and 6. In Fig. 5, the `PairPotential` interface is declared, providing the virtual methods `pairEnergy` and `pairForce`, which are the only methods a class inheriting from `PairPotential` (i.e., any pair potential) must implement. This allows the coding of a new pair potential only by writing two method implementations, as shown in Fig. 6 for the case of the Lennard-Jones potential *plug-in*.

4.2. Plug-ins

The dynamic loading of *plug-ins* is encapsulated in an API component called PluginManager. A `PluginManager` object provides an interface for loading and unloading *plug-ins* by name, as well as an associative array (or "dictionary" in other languages, like *Python* or *Perl*) interface for retrieving references to "live" *plug-ins* already loaded. Once loaded, a reference to a *Plugin* object is kept in memory, which then can be cast down into a particular kind of base class reference, such as *Potential* or *Integrator*. The `LoadPluginAs` method does the loading and casting in one call. In Fig. 7 we can see an example showing the use of a `PluginManager` object.

4.3. Applications

Having a set of *plug-ins* and the `PluginManager` object, we can write simple programs or *applications* using the LPMD API,

```
class PairPotential: public Potential
{
 public:
  PairPotential();
  virtual ~PairPotential();

  double energy(Configuration & conf);
  double AtomEnergy(Configuration & conf, long i);
  void UpdateForces(Configuration & conf);

  virtual double pairEnergy(const double & r) const = 0;
  virtual Vector pairForce(const Vector & r) const = 0;
};
```

Fig. 5. `PairPotential`, the base class for all pair potential functions in LPMD.

```
double LennardJones::pairEnergy(const double & r) const
{
 double rtmp=sigma/r;
 double r6 = pow(rtmp,6.0);
 double r12 = r6*r6;
 return 4.0*epsilon*(r12 - r6);
}

Vector LennardJones::pairForce(const Vector & r) const
{
 double rr2 = r.SquareModule();
 double r6 = pow(sigma*sigma/rr2, 3.0);
 double r12 = r6*r6;
 double ff = -48.0*(epsilon/rr2)*(r12 - 0.50*r6);
 Vector fv = r*ff;
 return fv;
}
```

Fig. 6. Implementation of the virtual methods from `PairPotential` in the `lennardjones` *plug-in*. Note how we just specify in the `LennardJones` class the pair potential function $\phi(r)$ and its derivative $d\phi/dr$. The methods in its base class, `PairPotential`, are in charge of handling the sum over all possible pairs, and other details such as the cut-off radius.

```
// Create a plug-in manager instance
PluginManager pm;

// This loads the 'linkedcell' plug-in, getting a reference to
// a CellManager object
CellManager & cm = pm.LoadPluginAs<CellManager>("linkedcell",
    "cutoff 8.5 nx 14 ny 14 nz 14");
// This loads the 'lennardjones' plug-in, getting a reference to
// a Potential object
Potential & pot = pm.LoadPluginAs<Potential>("lennardjones",
   "sigma 3.41 epsilon 0.0103408 cutoff 8.5");
// This loads the 'beeman' plug-in, getting a reference to
// an Integrator object
Integrator & integ = m.LoadPluginAs<Integrator>("beeman", "dt 1.0");
```

Fig. 7. Use of a `PluginManager` instance to load plug-ins.

```
Simulation & md = SimulationBuilder::CreateFixedOrthogonal(108, Atom("Ar"));
BasicCell & cell = md.Cell();
cell[0] = 17.1191*e1;
cell[1] = 17.1191*e2;
cell[2] = 17.1191*e3;

PluginManager pm;

BasicParticleSet & atoms = md.Atoms();

CellGenerator & cg = pm.LoadPluginAs<CellGenerator>("crystal3d",
 "type fcc symbol Ar nx 3 ny 3 nz 3");
cg.Generate(md);

CenterByCenterOfMass(atoms, cell);
md.SetCellManager(pm.LoadPluginAs<CellManager>("linkedcell",
 "cutoff 8.5 nx 7 ny 7 nz 7"));

Potential & pot = pm.LoadPluginAs<Potential>("lennardjones",
 "sigma 3.41 epsilon 0.0103408 cutoff 8.5");
Array<Potential &> & potentials = md.Potentials();

pot.SetValidSpecies(18, 18);
potentials.Append(pot);

md.SetTemperature(168.0);
md.SetIntegrator(pm.LoadPluginAs<Integrator>("beeman", "dt 1.0"));

potentials[0].Initialize(md);
potentials[0].UpdateForces(md);

for (long i=0;i<5000;++i)
{
 md.DoStep();
 if (i % 100 == 0)
 {
  double temp = Temperature(atoms);
  std::cout << i << "  " << temp << std::endl;
 }
}
```

Fig. 8. A simple MD example using the LPMD API to simulate 108 atoms of argon in the microcanonical ensemble for 5000 steps.

which can in principle, accommodate any need, just as with the "taylor-made" computer codes, the only difference being that, now we just need to interconnect different pieces. An example of this approach is the simple MD example code shown in Fig. 8, which simulates 108 atoms of argon interacting with the Lennard-Jones potential in the microcanonical ensemble (i.e., at constant total energy but fluctuating kinetic and potential energy).

Apart from the direct use of the API to combine different *plug-ins* into a "taylor-made" MD application, a task requiring some knowledge of C++ programming, the LPMD package provides its own ready-to-use "general purpose" applications or *utilities*: `lpmd`, `lpmd-analyzer`, `lpmd-converter` and `lpmd-visualizer`:

- `lpmd-analyzer`: This application is capable of calculating temporal or instantaneous properties of a sample, such as the radial distribution function, velocity autocorrelation function, mean square displacement, coordination neighbor analysis, and others (see Tables 6 and 7).
- `lpmd-converter`: This application is capable of building crystal structures, convert between file formats (from `CONFIG` `dlpoly`'s file to an `xyz`), apply filters and modifiers (see Section 7 and Tables 4 and 5), assign colors and others.
- `lpmd-visualizer`: This application is in charge of executing LPMD's own graphic visualizer (based in OpenGL), `lpvisual` as well as text visualizers, such as `printatoms`, that displays the atoms positions on the screen (see Table 11).

These applications are internally far more complex than the demonstration code shown in Fig. 8, as they are designed to read the complete description of the simulation at run-time, be it from an input file, or even from command-line *flags* or switches without the need for an input file. Despite the complexity of the applica-

```
#This is a comment. Comments are used usually as a title:
#########################################
# System file of Au crystal using LPMD  #
#########################################
cell cubic 28.56
input module=lpmd file=300K-Gold.lpmd level=1
output module=lpmd file=au.lpmd each=15 level=1
periodic false true true
steps 5000

#Integrator
use velocityverlet as vv
    dt 1.0
enduse

#CellManager

use linkedcell
     mode auto
     cutoff 7.5
enduse

# Sutton-Chen Potential (parameters for gold)
use suttonchen as sc
    e 0.013
    n 10
    a 4.08
    m 8
    c 34.408
    cutoff 7.5
enduse

#- Plugins using -#
integrator vv
cellmanager linkedcell
potential sc Au Au
```

Fig. 9. Example of an LPMD control file. The components are loaded (`use...enduse`) and then applied.

```
LPMD 2.0 L
HDR SYM X Y Z VX VY VZ rgb
9703
70.395 0 0 4.31045e-15 70.395 0 9.18485e-15 9.18485e-15 150
Cu 0.474872 0.448718 0.950733 0 0 -0.04 <1,1,1>
Cu 0.526154 0.448718 0.950733 0 0 -0.04 <1,1,1>
Cu 0.449231 0.474359 0.950733 0 0 -0.04 <1,1,1>
Cu 0.474872 0.474359 0.9387 0 0 -0.04 <1,1,1>
Cu 0.474872 0.5 0.950733 0 0 -0.04 <1,1,1>
Cu 0.500513 0.5 0.9387 0 0 -0.04 <1,1,1>
```

Fig. 10. Example of an input/output file in **lpmd** format. The first line indicates the version of LPMD (2.0) and type of data (`L` means normal, `Z` means compressed), the second are the headers (HDR) of the columns: atomic Symbol (SYM), atomic 3D positions (X Y Z), atomic 3D velocities (VX VY VZ) and color of each atom in red-green-blue format (rgb). The third line indicates the number of atoms in the simulation, while the fourth one gives the basis vectors that forms the cell, which could be non-orthogonal.

tions, their usage is very simple from the user point of view as we will see in the next section.

Using the LPMD utilities, the user can combine all the available *plug-ins* to perform many kinds of MD simulations, post-simulation analysis and visualization of samples (which can even be imported from other MD programs) and modification or filtering of atomic configurations to use in new MD simulations (for example build mixed-phase systems, projectiles and targets for High Velocity Impact (HVI) simulations, etc.). Doing all of this is possible without any programming knowledge.

5. General input format

5.1. Control files

All the LPMD utilities (`lpmd`, `lpmd-converter`, `lpmd-analyzer` and `lpmd-visualizer`) can be executed via command line through single-line-commands where all the instructions to execute a specific molecular dynamics task are given. But sometimes the instructions are too many and typing a long single-line-command could be inconvenient and hard to read. In those cases it is more appropriate to make a script in a file to execute it later. This file is called a *control file*.

An LPMD *control file* is shown in Fig. 9. This figure shows the most basic example of an LPMD molecular dynamics run. The first non-commented line creates a cubic cell of 28.56 Å (Å being the default LPMD's distance unit) in length. Then the input and output files are set in "lpmd" format (see Fig. 10), writing the output data each 15 steps. The `level=1` flag means that the corresponding file will contain 6 columns, representing the 3-dimensional position and velocity of the atoms, plus a column containing the atomic symbol of each atom, whereas `level=0` means just positions and `level=2` represents positions, velocities and accelerations. Then, the number of simulation steps is set. The last steps are: setting an integrator of the differential equations of motion, a cellmanager and an interatomic potential, whose gradient will be the force in the equations of motion. The time step `dt` used here is measured in femtoseconds (the default LPMD's time unit). In this example, 5000 simulation steps of 1.0 femtosecond, means 5 picoseconds of total simulation time.

```
cell cubic 17.1191
input crystal3d type=fcc symbol=Ar nx=3 ny=3 nz=3
prepare temperature t=168.0
monitor step,temperature start=0 end=5000 each=100
steps 5000
use lennardjones
    sigma 3.41
    epsilon 0.0103408
    cutoff 8.5
enduse
use beeman
    dt 1.0
enduse
use linkedcell
    cutoff 8.5
    nx 7
    ny 7
    nz 7
enduse
potential lennardjones Ar Ar
integrator beeman
cellmanager linkedcell
```

Fig. 11. A simple control file using the `lpmd` utility to simulate 108 atoms of argon in the microcanonical ensemble for 5000 steps.

```
#########CELL###########
cell cubic 17.1191

#########INPUT##########
input module=xyz file=output.xyz inside=true

#########MODULES######
use minimumimage
cutoff 8.0
debug none
enduse
use msd
output msd.dat
enduse

##########APPLY########
cellmanager minimumimage
property msd
```

Fig. 12. A control file for `lpmd-analyzer`.

After all the integrators, potentials and *plug-ins* are set, they are called in the last 3 lines of this example.

To show its convenience and how easy to use a *control file* is (instead of `c++` programming), an example is shown in Fig. 11, where the same effect of the MD demo shown in Fig. 8 can be achieved by running the *control file*. This file is later executed with the command `lpmd`.

The other utilities, mentioned before, work with the same kind of *control files* as well and, in fact, they all share the same syntax and keywords. In Fig. 12 an example of a *control file*, designed for `lpmd-analyzer` to calculate the mean-square-displacement (`msd`) of the atoms contained in the `output.xyz` file and using the `minimumimage` cell manager, is shown.

Fig. 13 is an example of a *control file* written for `lpmd-converter`. This control file creates an fcc structure in a 50 Å length simulation cell. Then the atoms inside the box region defined by $0 < x < 50$, $0 < y < 50$ and $0 < z < 20$ and outside the sphere centered in $(25, 25, 10)$ Å with radius 15 Å are saved into the file `data.xyz`.

This kind of configurations (drilled slices of materials) could be used to create for example, defects on the material or make grafts of one material inside another, embedding a crystal into a liquid, among other non-trivial uses.

5.2. Allowed formats

As we saw in Fig. 9, the input file is given in the control file by the `input module=lpmd` command. In this case, some specific options can be given to the module `lpmd` (`file` and `level`). But other formats are available, like `xyz`, `CONFIG` (from `dlpoly`), `POSCAR` (from `vasp`), `zlp` (compressed `lpmd`), `mol2`, `pdb`, `raw-binary` (see Table 1). It is as simple as changing `lpmd` in the example we saw by the desired format. For example, the line

```
input module=xyz file=configuration.xyz
```

loads the file `configuration.xyz`, which is written in the widely used `xyz` format, specified by `module`. The output file (specified by `output` in Fig. 9) can be written using any available file format module, following the same structure as `input`.

Another kind of inputs are the **cell generator plugins** included in LPMD. For example, the `crystal3d` plugin "creates" a monoatomic crystal by specifying the type of crystal (bcc, fcc, sc, ...) the amount of repetitions of the primitive cell in each axis, and the atomic symbol of the element:

```
input module=crystal3d type=fcc symbol=H
  nx=8 ny=8 nz=8
```

Note that, although `crystal3d` is not a file format, it is also a module and can be used as an input.

```
cell cubic 50

input crystal3d nx=16 ny=16 nz=16 symbol=N type=fcc
output xyz file=data.xyz

filter box x=0-50 y=0-50 z=0-20

filter sphere radius=15 center=<25,25,10> inverse=true
```

Fig. 13. A control file for `lpmd-converter`.

```
# Temperature Scaling: Rise from 300K to 500K

use tempscaling as ts
 from 300
 to 500
enduse

# Cell Scaling

use cellscaling as cs
 percent 0.01
 axis all
enduse

apply cs start=0 end=20000 each=200
apply ts start=1000 end=2000 each=50
```

Fig. 14. Portion of a *control file* showing the calling and applying of modifiers.

6. The simulation cell

An LPMD simulation cell consists of a set of atoms, 3 basis vectors, the volume enclosed by these three vectors, and attributes such as periodic boundary conditions. A simulation cell keeps track of the atoms inside or outside of it. Its shape can be orthogonal or non-orthogonal, this being determined only by the basis vectors. This is specified using the `cell` command as we saw in the previous examples. For the case of non-orthogonal cells, the `cell vector` command is used.

The simulation cell is an object, and just as the other LPMD objects (vectors, atoms, particle sets) it can be modified during or after the simulation. Some of the modifications that LPMD can apply over the whole (or a region) of the simulation cell are the cell scaling (stretch and compress), temperature rising (or lowering) and cutting of some spatial region (cubic, cylindrical or spherical extraction of atoms).

All of these modifications are possible during or after the simulation, by using the *filters* and *modifiers* provided by LPMD which we are going to discuss now.

7. Modifiers, filters and tags

7.1. Modifiers applied over the simulation cell

Modifiers are a type of *plug-ins*, whose task is, as the name suggests, to apply modifications to the atoms or to the whole simulation cell itself.

The modifiers that can be applied to the whole simulation cell can do things like shearing, temperature rising, compressing and expanding the volume of the cell.

An example of a modifier is shown in Fig. 14, in which the temperature is increased from 300 K to 500 K during 1000 MD steps (each 50 MD steps), while the whole cell is submitted to a hydrostatic expansion for the first 20 000 MD steps (each 200 MD steps) through the rescaling of the basis vectors in 0.01% of their former value in each application.

7.2. Filters

In the nomenclature used in LPMD, *filter* means *atom selection*. If used without a *modifier* (see 7.1), a *filter* performs *atom elimination*. They are used to select certain types of atoms or regions of the simulation cell, eliminating every non-selected atom. Conversely (just adding `inverse=true`), it can be used to eliminate only the selected atoms.

Some type of filters available are:

- Box: Select the atoms that are in the region $[x_{\mathrm{MIN}}, x_{\mathrm{MAX}}] \times [y_{\mathrm{MIN}}, y_{\mathrm{MAX}}] \times [z_{\mathrm{MIN}}, z_{\mathrm{MAX}}]$. Usage:

  `filter box x=0-5 y=0-6 z=10-15`,

  meaning the set of points $[0, 5] \times [0, 6] \times [10, 15]$ in the respective directions. This eliminates all the atoms of the cell except the ones belonging to the selected region.
- Element: Select the atoms by element (hydrogen, oxygen, argon). Usage:

  `filter element symbol=Au`.

  This eliminates every atom except the gold ones.
- Index: Select atoms by their index in the simulation cell. It is commonly used to eliminate the first or last added atoms. Usage:

  `filter index index=1-50`.

  This eliminates the atoms within the given range.
- Tag: Select atoms by a given label (tag). An example of *Tag* usage is identifying the atoms that form a “bullet” to study HVI. Usage:

  `filter tag name=fixedpos value=true`.

  This eliminates all non-moving atoms (fixed position). In the case of the bullets, you can select a set of atoms and put a *tag* named *bullet*, then execute

  `filter tag name=bullet`

  to eliminate all the atoms that are not the bullet.

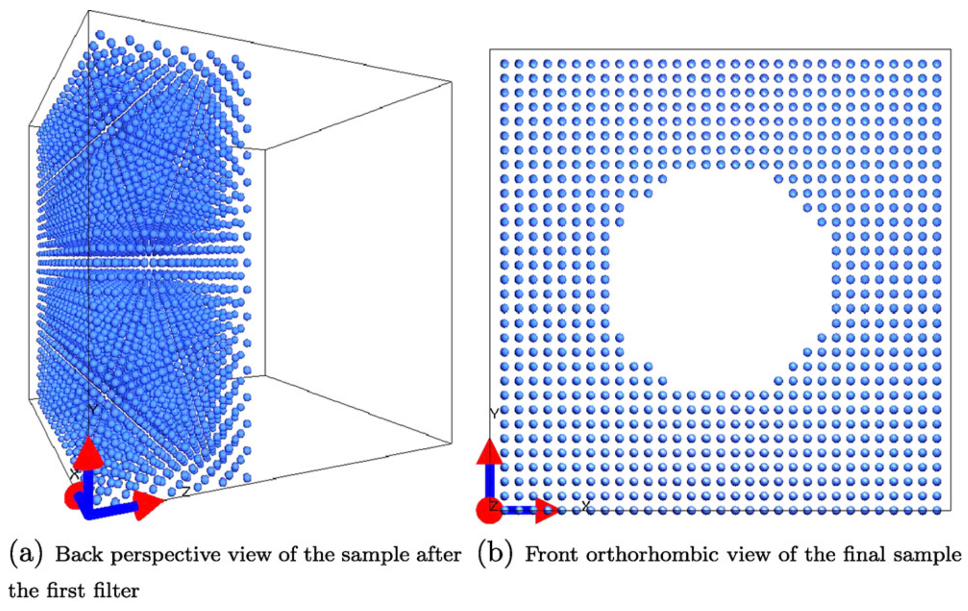
Y
X
Z
(a) Back perspective view of the sample after the first filter
(b) Front orthorhombic view of the final sample

Fig. 15. A filtered fcc crystal. A `box` filter was applied first to leave the half of the cube (left) and then a `sphere` filter was used to make a "hole" inside the "plane" of atoms (right).

```
use setvelocity
    velocity <0,0,0.05>
enduse

apply setvelocity start=0 end=1 each=1 over box x=0-26.3 y=0-26.3 z=0-7.0
```

Fig. 16. Assigning a velocity vector to a box of atoms (piece of a *control file*).

```
use setcolor as white
    color <1,1,1>
    debug none
enduse

apply white start=0 end=-1 each=1 over index index=0-99
```

Fig. 17. Assigning color to the first 100 atoms in the list (piece of a *control file*).

- Sphere: Select atoms inside a spherical region. Usage:

  `filter sphere radius=10 center=<10,40,25>.`

  This eliminates all the atoms that do not belong to the sphere with center in (10, 40, 25) Å and radius 10 Å. As mentioned before, it is possible to do the opposite, eliminating *just* the atoms inside the sphere with

  `filter sphere radius=10 center=<10,40,25>`
  `inverse=true.`

The images shown in Fig. 15 (taken from LPMD's own visualizer, *lpvisual*) show a cubic cell of 50 Å size with 4432 atoms after a filtering process.

7.3. *Modifiers applied over sets of atoms*

Modifiers are strongly linked to the *filters*, because the *filters* can select the atoms to which the modifications are applied.

Besides the modifiers already mentioned, there are modifiers for applying rotations and translations to the atoms, velocity rescaling, removing atoms, coloring them and others. A complete list of modifiers is given in Appendix A (Table 5).

Some examples of modifiers are shown in Figs. 16 to 20. Fig. 16 shows how to assign a uniform velocity to a group of atoms selected using a `box` filter, and Fig. 17 shows that a color can be assigned in the same way, using the `setcolor` *plug-in* instead of `setvelocity`. Note also that the atoms to be colored white are now filtered by `index`.

As a modular part of the code, modifiers, filters and tags have the advantage of being applied either during or after the simulation is made. An example of this is shown in Fig. 18, where filters and modifiers are combined to set up a high-velocity impact (HVI) simulation.

Figs. 19 and 20 show the effect of a shockwave in solid argon, also generated by combining filters and modifiers to create mobile "pistons" and fixed "walls".

8. Parallelization using OpenMP

Currently LPMD is able to take advantage of shared-memory parallel architectures using the OpenMP API. For this effect, the code must be compiled with the `-openmp` option. The usual environment variables used by OpenMP apply: for example, the number of threads or processors is specified by setting the `OMP_NUM_THREADS` variable to the desired number.

```
use setvelocity
    velocity <0,0,0.05>
enduse

apply setvelocity start=0 end=1 each=1 over box x=0-26.3 y=0-26.3 z=0-7.0

use settag as projectil
    tag fixedvel
    value true
enduse

use settag as wall
    tag fixedpos
    value true
enduse

use settag as projwall
    tag fixedpos
    value true
enduse

apply proyectil start=0 end=1 each=1 over box x=0-26.3 y=0-26.3 z=0-7.0
apply proywall start=45 end=46 each=1 over tag name=fixedvel value=true
apply wall start=0 end=1 each=1 over box x=0-26.3 y=0-26.3 z=100.7-105.2
```

Fig. 18. Multiple modifiers on a single simulation are also possible (piece of a *control file*). The `settag` *plug-in* is used to label a group of atoms (defined on `apply`). In the case of `wall`, the position of the atoms is set to constant (`fixedpos`) and the `projectil` remains with constant velocity (`fixedvel`), i.e., acts like an infinite-mass body.

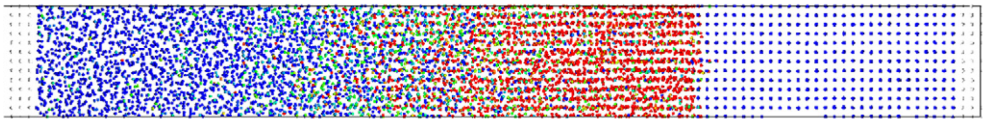

Fig. 19. A shockwave generated using modifiers. To the left, painted in white, the atoms with fixed velocities push the argon atoms as a piston, generating a shockwave, whose atoms are colored by a temperature scale. To the right, also painted in white, atoms with fixed positions act as an immobile wall.

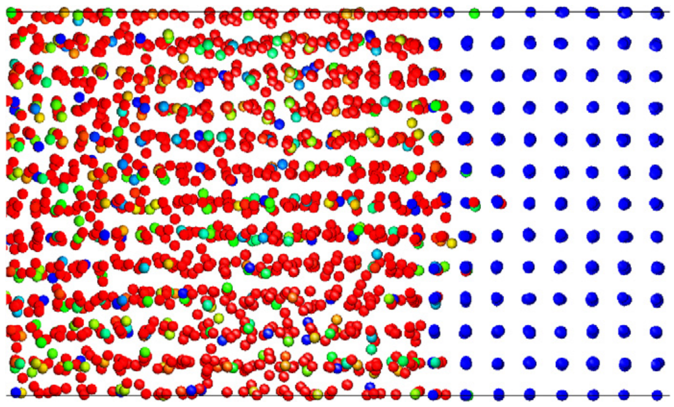

Fig. 20. A close-up to the shockwave shown above.

Fig. 21 shows the performance of LPMD using OpenMP up to 8 processors. Each simulation consisted on 500 MD steps for a system composed of 20 000 Au atoms interacting via the embedded-atom potential of Sutton and Chen. Each physical processor is an Intel(R) Xeon(R) E5420 CPU running at 2.50 GHz.

9. Final remarks

We have developed a fully modular MD program for the study of both equilibrium and non-equilibrium phenomena, as well as bulk systems and highly inhomogeneous systems. This is achieved by employing a hierarchical design and exploiting the benefits of object-oriented programming. The program is written in standard C++ which makes it portable across several platforms. All the components can be rearranged in many different set-ups, from MD simulation to post-simulation analysis and preparation of complex samples.

We have also provided a wide range of examples describing the capabilities of the code and illustrating the flexibility gained by its design, as different kinds of problems can be solved just by linking together a subset of all the available components.

Possible improvements include the parallelization of the code using the Message-Passing Interface (MPI) libraries, as well as internal optimization. Every component already available is easy to understand and modify, and new components from external parties are possible (and encouraged) by using the open framework or API we have developed.

Acknowledgements

This work is supported by Project PBCT Anillo ACT/24 *Computer Simulation of nanobio systems*. F.G. and C.L. acknowledge a PhD fellowship from CONICYT. The authors thank specially Gonzalo

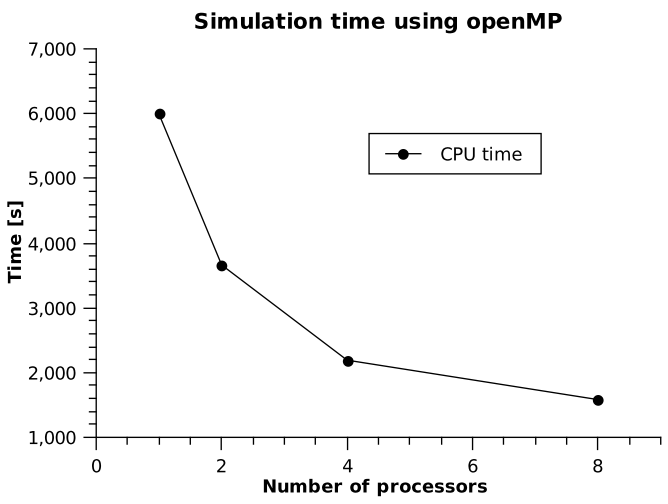

Fig. 21. Parallel performance of LPMD using OpenMP parallelization.

Gutiérrez, Eduardo Menéndez and Carlos Esparza for the constant support and for encouraging the cause.

Appendix A

A.1. Input/output modules

Modules to manage the input/output files that contain the atomic configurations (Table 1).

A.2. Cell generators

`lpmd` modules that create atomic cells automatically (Table 2).

A.3. Cellmanagers

Cellmanagers are encapsulated algorithms used by `lpmd` to divide the simulation cell in different ways, so that each atom computes the interactions only with its closest neighbors (Table 3).

A.4. Filters

Filters act over the simulation cell and are capable of selecting atoms in different ways. If they are not used with a modifier, they eliminate atoms (Table 4).

A.5. Modifiers

Modifiers apply various kinds of modifications to the simulation cell and the atoms within, including rescaling of positions or velocities, shearing, removing or randomly substituting atoms, etc. (Table 5).

A.6. Instantaneous properties

This modules calculate properties of the studied atomic sample. These are spatial properties independent of time, so they can be calculated in each simulation step. They can also be averaged in time. They can be calculated for a configuration (a single file) using `lpmd-analyzer` or during the simulation (Table 6).

A.7. Temporal properties

They calculate temporal properties of the system. These properties cannot be calculated during the simulation, as they depend

Table 1
Input/output modules used by `lpmd` and its utilities.

Module	Description
`dlpoly`	Reading/writing of `HISTORY` and `CONFIG` `dlpoly` files
`lpmd`	Own format of `lpmd`. Reading/writing support, 3 different levels (positions, velocities, accelerations) and tags
`vasp`	Reads `POSCAR` files. Used by the VASP software
`xyz`	Reads `xyz` files. Like `lpmd`, it has 3 different levels
`zlp`	Own format of `lpmd` to create compressed `lpmd` files using zlib. 3 different levels
`mol2`	Reading/writing of `mol2` files. Basic support
`pdb`	Reading/writing of `pdb` files. Basic support
`rawbinary`	Reading/writing of files in binary mode. Uses less hard-disk space and works faster

Table 2
Cell generator modules used by `lpmd` and its utilities.

Module	Description
`crystal3d`	Three-dimensional cells generator
`crystal2d`	Two-dimensional cells generator
`voronoi`	Nano-structured cells generator using the Voronoi method
`skewstart`	Cell generator using the skewstart method, developed by *K. Refson* for the MD program, **moldy**

Table 3
Modules that manage the interaction between lists of atoms.

Module	Description
`linkedcell`	Manages the interaction lists using the *Linked Cell* method
`minimumimage`	Manages the interaction lists using the *minimum image* method
`lcbinary`	Manages the interaction lists using the *Linked Cell* method using 1 atom per subcell
`verletlist`	Manages the interaction lists using the *Verlet List* method

Table 4
Modules that filter atoms in the simulation cell.

Module	Description
`box`	Selects atoms inside or outside a box of a given size
`element`	Selects atoms by their atomic symbol
`index`	Selects atoms by their index in the simulation cell
`sphere`	Selects atoms inside or outside a sphere of a given radius and center
`tag`	Selects atoms by their tag (label)

Table 5
Modifiers modules of the system.

Module	Description
berendsen	Temperature scaling of the cell using Berendsen's algorithm
cellscaling	Volume scaling of the cell
displace	Displace atoms
moleculecm	Creates diatomic molecules from bonded atoms
propertycolor	Color atoms by property
quenchedmd	Structural minimization using *Quenched MD* (energy minima searching)
randomatom	Random elimination/selection of atoms
replicate	Replicates the original cell
rotate	Atom rotation
setcolor	Assign some specific color to atoms
settag	Assign a tag to the atoms
setvelocity	Assign velocity to the atoms
shear	Applies shearing to the simulation cell
temperature	Assign temperature to groups of atoms
tempscaling	Use simple velocity rescaling
undopbc	Undo periodic boundary conditions

Table 6
Instantaneous properties provided by `lpmd`.

Module	Description
angdist	Calculates the angular distribution of the sample
cna	Makes a *Common Neighbor Analysis* [14] of the sample
cordnumfunc	Calculates the *coordination number function* of the sample
cordnum	Calculates the coordination number in histogram form
densityprofile	Generates a density profile of the sample
gdr	Calculates the pair distribution function of the sample
localpressure	Generates a local pressures profile
pairdistances	Search for pairs of atoms closer than a given distance
sitecoord	Calculates the coordination number of individual atoms
tempprofile	Generates a temperature histogram of the sample
veldist	Generates a velocity profile of the sample

Table 7
Temporal properties provided by `lpmd`.

Module	Description
dispvol	Calculates the volume inside which the atoms diffuse
msd	Calculates the mean square displacement
vacf	Calculates the velocity autocorrelation function

on past and future instants. They can be calculated using `lpmd-analyzer` taking as input the output files of a previous simulation (Table 7).

A.8. Integrators

They are components used to propagate the state of the system from one state to the next. They are not limited to the integration of Newton's equation, for instance the `metropolis` *plug-in* updates the system using a Monte Carlo algorithm (Table 8).

Appendix B. Pair potentials

Plug-ins providing different kinds of pair interatomic interactions between the atoms in the system being simulated (Table 9).

B.1. Metallic potentials

They provide potential models for atomic interactions best suited for metallic systems (Table 10).

Table 8
Integrators provided by `lpmd`.

Module	Description
beeman	Beeman integration algorithm
euler	Euler integration algorithm
hardspheres	Algorithm specialized in dynamics of hard spheres
leapfrog	Leapfrog integration algorithm
metropolis	Metropolis Monte Carlo method, used, for example, for structural minimization
nullintegrator	It does not move the atoms. Mainly used for testing and benchmarks
velocityverlet	Velocity Verlet integration algorithm
verlet	Verlet integration algorithm

Table 9
Interatomic potentials provided by `lpmd`.

Module	Description
buckingham	Atomic interaction with Buckingham potential
harmonic	Atomic interaction with harmonic potential
lennardjones	Atomic interaction with Lennard-Jones potential
morse	Atomic interaction with Morse potential
nullpairpotential	Null atomic interaction. Mainly used for testing and benchmarks
tabulatedpair	Atomic interaction read from a data table

Table 10
Interatomic metallic potentials provided by `lpmd`.

Module	Description
finnissinclair	Atomic interaction with Finnis–Sinclair potential
gupta	Atomic interaction with Gupta potential
nullmetalpotential	Null atomic interaction. Mainly used for testing and benchmarks
suttonchen	Atomic interaction with Sutton–Chen potential

Table 11
Visualizers available in `lpmd`.

Module	Description
average	Average data visualizer during the simulation
lpvisual	Graphic Molecular Dynamics visualizer based in OpenGL
monitor	Instantaneous data visualizer during the simulation
printatoms	Displays the position of the atoms on the screen

B.2. Visualizers

Used to obtain a picture or video of the simulation, or to display information in real time (Table 11).

References

[1] L. Verlet, Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules, Phys. Rev. Z 159 (1967) 98.
[2] H. Rafii-Tabar, A.P. Sutton, Long-range Finnis–Sinclair potentials for f.c.c. metallic alloys, Philos. Mag. Lett. 63 (1991) 217–224.
[3] G. Gutiérrez, B. Johansson, Molecular dynamics study of structural properties of amorphous Al_2O_3, Phys. Rev. B 65 (2002) 104202.
[4] J. Peralta, G. Gutiérrez, J. Rogan, Structural and vibrational properties of amorphous GeO_2: A molecular dynamics study, J. Phys. Condens. Matter 20 (2008) 145215.
[5] Timothy C. Germann, Kai Kadau, Trillion-atom molecular dynamics becomes a reality, Internat. J. Modern Phys. C 19 (2008) 1315–1319.
[6] P. Walsh, A. Omeltchenko, R.K. Kalia, A. Nakano, P. Vashishta, S. Saini, Nanoindentation of silicon nitride: A multi-million atom molecular dynamics study, Appl. Phys. Lett. 82 (2003) 118.
[7] J. Karl Johnson, John A. Zollweg, Keith E. Gubbins, The Lennard-Jones equation of state revisited, Mol. Phys. 78 (1993) 591–618.
[8] M.S. Daw, M.I. Baskes, Embedded-atom method: Derivation and application to impurities, surfaces, and other defects in metals, Phys. Rev. B 29 (1984) 6443–6453.
[9] A.P. Sutton, J. Chen, Philos. Mag. Lett. 61 (1990) 139.

[10] W.C. Swope, H.C. Andersen, P.H. Berens, K.R. Wilson, A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters, J. Chem. Phys. 76 (1982) 637.

[11] K. Refson, MOLDY, a general–purpose molecular dynamics code, available free for academic purpose, http://www.earth.ox.ac.uk/~keith/moldy.html.

[12] W. Smith, DLPOLY 3, release 3.06, http://www.cse.scitech.ac.uk/ccg/software/DL_POLY, 2006.

[13] M. Allen, D. Tildesley, Computer Simulation of Liquids, Oxford University Press, USA, 1989.

[14] J.D. Honeycutt, H.C. Andersen, Molecular dynamics study of melting and freezing of small Lennard-Jones clusters, J. Phys. Chem. 91 (1987) 4950–4963.