Lumol user manual

Release 0.0.0

The lumol developers

Nov 13, 2019

Contents

Note: Lumol is actively developed and should be considered as alpha software.

As the code is likely to change so is this documentation.

Welcome to the Lumol user manual. In this book we teach you how to use Lumol to set up and run classical molecular simulations. We designed Lumol to be *flexible* and *extensible*; you are able to customize your simulation to suit your needs and use it as a platform to implement your own algorithms and customized potential functions in an *easy* way. You can use Lumol as a command line tool as well as a library in your own code.

1 Installation

Lumol is written in rust (*why?*), and you will need a Rust compiler to compile it. You can find rust installation instructions here, or use your system package manager. Lumol also depends on some C++ libraries, so you will need a C++ compiler and CMake to be installed.

Lumol is tested on Linux and OS X, and should build on Windows without any issue. You will need a C++11 capable compiler on Windows (MSVC > 15 or Mingw with gcc > 4.9). Be sure to pick the corresponding version of the Rust compiler.

When all the dependencies are installed on you system, you can install the the latest development version with:

```
cargo install --git https://github.com/lumol-org/lumol
```

This command will download and install lumol in `~/.cargo/bin/lumol`, where `~` is your home directory. You may want to add `~/.cargo/bin` to your PATH or move the `lumol` binary in another directory accessible in your PATH.

You can check that the installation worked by running

```
lumol --version
```

2 Tutorials

This section will teach you how to use Lumol to run basic simulations. You can and should re-use the input files shown in the examples to run your own simulations.

2.1 Monte Carlo simulation of Argon

So let's run out first simulation with Lumol. The easiest system to simulate is a Lennard-Jones fluid, which is a good model for noble gas fluids. Here we will simulate super-critical argon using the Metropolis Monte Carlo algorithm.

For this simulation, you will need the following files:

- the initial configuration `argon.xyz`
- the input file `argon.toml`

You can download both files at the following URL: https://lumol.org/lumol/latest/book/_downloads/argon.zip.

After extracting the archive, you can run the simulation with `lumol argon.toml`. The simulation should complete in a few seconds and produce two files: `energy.dat` and `trajectory.xyz`.

Input file anatomy

The input file is written using the TOML syntax, you can learn more about this syntax here. The file starts with a header declaring the version of the input file syntax used, here the version 1:

```
[input]
version = 1
```

Then, we declare which system that we want to simulate in the `systems` array. We define this system using an XYZ file for which we also have to provide the unit cell size.

```
[[systems]]
file = "argon.xyz"
cell = 21.65
```

We also need to define the interaction potential between the atoms in the system, which we'll do in the `system.potential.pairs` section. Here, we are using a Lennard-Jones potential with a cutoff distance of 10 Angstrom for all Argon (Ar) pairs.

```
[systems.potentials.pairs]
Ar-Ar = {type = "lj", sigma = "3.4 A", epsilon = "1.0 kJ/mol", cutoff = "10 A"}
```

Next, we define how we want to simulate our system. For this brief tutorial simulating `100000` steps should be enough. We also specify some output: the energy is written to `energy.dat` every 100 steps, and the trajectory is written to `trajectory.xyz` also every 100 steps.

```
[[simulations]]
nsteps = 100000
outputs = [
    {type = "Energy", file = "energy.dat", frequency = 100},
    {type = "Trajectory", file = "trajectory.xyz", frequency = 100}
]
```

Finally, we define how to propagate the system from one step to another. We are using a Monte Carlo simulation in this tutorial. We need to specify the temperature (set to 500 K) and the set of moves that we want to perform to change the systems. The only Monte Carlo move in this example is a translation for which we set the maximum amplitude (the furthest a particle can be translated in a single move) to 1 Angstrom.

```
[simulations.propagator]
type = "MonteCarlo"
temperature = "500 K"
moves = [
    {type = "Translate", delta = "1 A"},
]
```

Wrapping all this together, here is the complete input file:

```
[input]
version = 1

[[systems]]
file = "argon.xyz"
cell = 21.65

[systems.potentials.pairs]
Ar-Ar = {type = "lj", sigma = "3.4 A", epsilon = "1.0 kJ/mol", cutoff = "10 A"}

[[simulations]]
nsteps = 100000
outputs = [
    {type = "Energy", file = "energy.dat", frequency = 100},
    {type = "Trajectory", file = "trajectory.xyz", frequency = 100}
]

[simulations.propagator]
type = "MonteCarlo"
temperature = "500 K"
moves = [
    {type = "Translate", delta = "1 A"},
]
```

As mentioned above, you can start the simulation using

```
lumol argon.toml
```

So now we know how to run a simulation of a Lennard-Jones fluid. How about we add electrostatic interactions in the *next example*?

2.2 Hello Sodium Chloride

In this example, we will simulate a Sodium Chloride crystal using molecular dynamics to propagate the position throughout time. Sodium Chloride adds a challenge in simulations because each atom carries a charge. These charges interact with a Coulomb potential which goes to zero as $1/r$. The problem is that the cutoff scheme used for pair potentials in most molecular simulations can not be used for anything that goes to zero slower than $1/r^3$. So we need to use alternative methods to compute the potential for the interactions between pairs of charges.

For this simulation, you will need the following files:

- the initial configuration `nacl.xyz`
- the input file `nacl.toml`

You can download both files at the following URL: https://lumol.org/lumol/latest/book/_downloads/nacl.zip.

Again, you can run the simulation which should complete in a minute with `lumol nacl.toml`. This will perform a molecular dynamics simulation of a NaCl crystal using electrostatic interactions between atomic charges.

The input file commented

We start with the input version again:

```
[input]
version = 1
```

Then we load the system from the `nacl.xyz` file and define the unit cell.

```
[[systems]]
file = "nacl.xyz"
cell = 22.5608
```

Next, we define our potential. This section is way bigger than the one for our previous Lennard-Jones example:

```
[systems.potentials.global]
cutoff = "8 A"

[systems.potentials.charges]
Na = 1.0
Cl = -1.0

[systems.potentials.pairs]
Na-Na = {type = "lj", sigma = "2.497 A", epsilon = "0.07826 kcal/mol"}
Cl-Cl = {type = "lj", sigma = "4.612 A", epsilon = "0.02502 kcal/mol"}
Na-Cl = {type = "lj", sigma = "3.5545 A", epsilon = "0.04425 kcal/mol"}

[systems.potentials.coulomb]
wolf = {cutoff = "8 A"}
```

Let's break it down. First, we define some global values for the interactions: setting `systems.potentials.global.cutoff` will use the given cutoff for all pair interactions in the system. We set the charges of atoms in the `systems.potentials.charges` section.

```
[systems.potentials.global]
cutoff = "8 A"

[systems.potentials.charges]
Na = 1.0
Cl = -1.0
```

Then, we need to define the pair interactions for all possible pair combinations in the system, *i.e.* (Na, Na), (Cl, Cl), and (Na, Cl).

```
[systems.potentials.pairs]
Na-Na = {type = "lj", sigma = "2.497 A", epsilon = "0.07826 kcal/mol"}
Cl-Cl = {type = "lj", sigma = "4.612 A", epsilon = "0.02502 kcal/mol"}
Na-Cl = {type = "lj", sigma = "3.5545 A", epsilon = "0.04425 kcal/mol"}
```

Because our system contains charges, we need to use an electrostatic potential solver. Here we are going for the `Wolf` solver, with a cutoff of 8 A. Note that if we'd chose a cutoff different from the global one defined above, we would overwrite the global one *for the coulombic interactions*.

```
[systems.potentials.coulomb]
wolf = {cutoff = "8 A"}
```

We can now define the simulation and the outputs for this simulation. We are using a molecular dynamics simulation of the NaCl crystal with a timestep of 1 fs for integration (this will produce a NVE ensemble).

```
[[simulations]]
nsteps = 5000
outputs = [
    {type = "Trajectory", file = "trajectory.xyz", frequency = 10}
]

[simulations.propagator]
type = "MolecularDynamics"
timestep = "1 fs"
```

As before, run the simulation via

```
lumol nacl.toml
```

Until now, the force field we used for the system was defined in the same input file (in the `system.potential` section) as rest of the simulation settings. It can sometimes be interesting to separate the force field from the rest of the input, in particular when using the same force-field for multiple simulations. In the *next example*, we will do exactly this.

2.3 Molecular dynamics simulation of water

In this example, we will simulate a molecular liquid of high scientific interest: water. At the same time we will learn how we can reuse a potential definition between multiple simulations.

You will need three input files for this simulation:

- the initial configuration `water.xyz`;
- the simulation input `water.toml`;
- the force field (potential definitions) `water-fSCP.toml`.

You can download these files at the following URL: https://lumol.org/lumol/latest/book/_downloads/water.zip.

As usual, you can run the simulation with `lumol water.toml`. The simulation should finish in a few minutes.

The input files commented

`water.toml` file

The main input file is pretty similar to the previous examples, with two novelties:

- The `guess_bonds = true` entry tells lumol to try to guess the bonds from the distances in the XYZ file. This is needed because we want to simulate a molecule but there is no bonding information inside the XYZ format. If we were to use a PDB file with connectivity instead, this would not be needed;
- The `potentials = "water-fSCP.toml"` tells lumol to read the potentials from the `water-fSCP.toml` file. Using this type of input for the potentials we can reuse the same potential for multiple simulations and also easily change the potential used in the simulation.

```
[input]
version = 1

[[systems]]
file = "water.xyz"
```

(continues on next page)

(continued from previous page)

```
guess_bonds = true
cell = 28.0
potentials = "water-fSCP.toml"

[[simulations]]
nsteps = 5000
outputs = [
    {type = "Trajectory", file = "trajectory.xyz", frequency = 10}
]

[simulations.propagator]
type = "MolecularDynamics"
timestep = "1 fs"
```

`water-fSCP.toml` file

`water-fSCP.toml` is a standalone potential input file. It contains the same data as a potential definition inside a `[[systems]]` section, but without the `system.potential` prefix on all section names.

In this input, we start by defining which version of the input format we are using.

```
[input]
version = 1
```

Then, we can define some global input data: the pair potential cutoff and the atomic charges.

```
[global]
cutoff = "14 A"

[charges]
O = -0.82
H = 0.41
```

The pair potential section contains the usual declarations for pairs, with a few additional options.

```
[pairs]
O-O = {type = "lj", sigma = "3.16 A", epsilon = "0.155 kcal/mol"}
```

We can add a `restriction` to restrict a specific pair interaction to some kind of pairs. Here, we will only account for H-H pairs inside the same molecule (`"intra-molecular"` interactions).

```
H-H = {type = "harmonic", k = "79.8 kcal/mol/A^2", x0 = "1.633 A", restriction =
↪"intra-molecular"}
```

We can also define a non-interacting pair interaction! This is useful when some atoms do not interact in the model we use.

```
H-O = {type = "null"}
```

Next, the bonds and angles are defined. These are interactions used between bonded particles (or angles formed by two bonds and dihedral angles formed by three bonds). This section follows the same pattern as the `[pairs]` section.

```
[bonds]
O-H = {type = "harmonic", k = "1054.2 kcal/mol/A^2", x0 = "1.0 A"}

[angles]
H-O-H = {type = "harmonic", k = "75.9 kcal/mol/rad^2", x0 = "109.5 deg"}
```

Finally we specify how to compute the electrostatic interaction, this time using the Ewald summation method. We can restrict the coulombic interactions to only apply between particles not in the same molecule by using `restriction = "inter-molecular"`.

```
[coulomb]
ewald = {cutoff = "8.5 A", kmax = 3}
restriction = "inter-molecular"
```

The simulation is run via

```
lumol water.toml
```

3 Lumol concepts

In order to run a molecular simulation, you need two things: a description of the system you are simulating and a set of algorithms used to propagate the simulation. This section presents how the `System` and the `Simulation` objects work in Lumol. The `System` contains the full description of the system, and the `Simulation` contains the algorithms used to propagate the system.

3.1 System

A `System` contains all the data about the physical system we are simulating. It contains four types of data:

- A list of **Particles**, which are physical objects with a position, a velocity, a mass and a name;
- A list of **Molecules** containing information about how the particles are bounded together;
- An **UnitCell**, *i.e.* the bounding box of the simulation.
- **Interactions**, sometimes called a force field.

Particles

Particles are the basic building blocks of a system. They can be atoms or more complex: coarse-grained sites, dummy atoms, anisotropic particles ...

They have a name, a mass, a position, a velocity, and most importantly a particle kind, defined by the name of the particle. All particles with the same name share the same kind: all `H` are the same, as well as all `C`, *etc.*

Molecules

When particles are bonded together, they form molecules. A `Molecule` contains the list of its bonds; molecules make for the *molecular* part in *molecular simulation*.

Unit cells

Lumol knows about three types of unit cells:

- Infinite cells do not have any boundaries and can be used to simulate systems in vacuum;
- Orthorhombic cells have up to three independent lengths whereas all angles of the cell are set to 90°;
- Triclinic cells have 6 independent parameters: 3 lengths and 3 angles.

Orthorhombic and Triclinic cells are used in combination with periodic boundary conditions to simulate infinite systems.

Interactions

Interactions are potentials acting on or between particles. Lumol provides functions for various potential types:

- Non-bonding pair potentials;
- Bonds potentials in molecules;
- Angles potentials in molecules;
- Dihedral angles potentials in molecules;
- Long-ranges coulombic potentials (Ewald and Wolf methods);
- Arbitrary external potential applying on the whole system at once.

3.2 Simulation

A simulation in Lumol always contains the same steps:

1. Setup the system and initialize the algorithms using system specific information;
2. Propagate the system for one step using a Propagator;
3. Compute the physical properties of the system and output them to a file;
4. Check if the simulation is finished (either the required number of steps has been done, or a convergence criterion is reached) and return the updated system. If the simulation is not finished, go back to (2)

Propagators

Propagators are at the heart of a Simulation. They have the responsibility to update the system at each simulation step. Currently, three propagators exists: a molecular dynamics one, a Monte Carlo one and a minimizer, for energy minimization.

Output algorithms

Output algorithms have the responsibility to compute and output statistical data about the simulated system. Temperature, energy, radial distribution functions are some examples of output algorithms.

3.3 Units

The unit system used internally by Lumol is the following:

- Angstrom (`A`) for distances;
- Femtosecond (`fs`) for time;
- Unified atomic mass unit (`u` or `Da`) for mass;
- Kelvin (`K`) for temperature;
- Number of particles for quantity of matter;
- Radian (`rad`) for angles;

Any other internal unit is derived from this set:

- The internal unit of energy is `u A^2 fs^-2`;
- The internal unit of force is `u A fs^-2`;
- The internal unit of pressure is `u A^-1 fs^-2`;
- *etc.*

For convenience Lumol provides convertion facilities for any value in these internal unit to any others units. The following table lists available units that can be converted:

Quantity	Accepted units
Distance	A, Å, nm, pm, fm, m, bohr
Time	fs, ps, ns
Mass	u, Da, kDa, g, kg
Matter	mol
Angle	rad, deg
Energy	J, kJ, kcal, eV, H, Ry
Force	N
Pressure	Pa, kPa, MPa, bar, atm

In the input files, the units are specified as strings, and must be spelled exactly as in the above table. They can be combined with other units using `*` for multiplication, `/` for division, and `^` for exponentiation. Parentheses can be used to group sub-units together. Some valid unit strings are `kcal/mol`, `(J / mol) * A^-2`, and `m*fs^-1`.

4 Input file reference

This section describes how to use input files to run your simulations with Lumol. An input file contains all information that you need to run a simulation and it is usually organized in four main sections: **input**, **log**, **systems** and **simulations**.

- The **input** section contains metadata about the input itself (i.e. a version number).
- The **log** sections explains different methods about how Lumol reports information about your simulation such as warnings and errors.
- The **systems** section contains information about the initial configuration, the interactions between atoms and the simulation cell.
- The **simulations** section defines how your system will propagate. You can generally choose between molecular dynamics (MD), Monte-Carlo (MC) and energy minimization.

Lumol's input files use the TOML format, a simple and minimalist configuration format based on `key = value` pairs. You can read an introduction to the TOML format here.

Example:

```
# an example input file for a Monte Carlo simulation

# input section
[input]
version = 1

# log section
[log]
target = "lumol.log"

# systems section
[[systems]]
file = "data/ethane.xyz"
guess_bonds = true
cell = 100.0

[systems.potentials.global]
cutoff = "14.0 A"
tail_correction = true
```

(continues on next page)

(continued from previous page)

```
[systems.potentials.pairs.C-C]
type = "lj"
sigma = "3.750 A"
epsilon = "0.814 kJ/mol"
restriction = "InterMolecular"

[systems.potentials.bonds]
C-C = {type = "null"}

# simulations section
[[simulations]]
nsteps = 1_000_000
outputs = [
    {type = "Energy", file = "ethane_ener.dat", frequency = 500},
    {type = "Properties", file = "ethane_prp.dat", frequency = 500}
]

[simulations.propagator]
type = "MonteCarlo"
temperature = "217.0 K"
update_frequency = 500

moves = [
    {type = "Translate", delta = "20 A", frequency = 50, target_acceptance = 0.5},
    {type = "Rotate", delta = "20 deg", frequency = 50, target_acceptance = 0.5},
    {type = "Resize", pressure = "5.98 bar", delta = "5 A^3", frequency = 2,␣
↪target_acceptance = 0.5},
]
```

4.1 `[input]` section

All input files must contain an `[input]` section looking like this:

```
[input]
version = 1
```

Introducing a `version` key helps us to make changes to the input file format while keeping compatibility with previous formats. Please note that Lumol is not in version 1.0 yet and we currently cannot guarantee compatibility for input files.

4.2 `[log]` section

Lumol sends various logging messages while running a simulation. Some of them are informational messages (`charge was set to 1.2 for 132 particles`), others are warnings or error message (`infinite energy!`) and some are for debugging purposes.

By default, Lumol prints all the informational, warning and error messages to the standard terminal output. This allows to run the code and redirect the output to a specific file in the usual UNIX way: `lumol input.toml > simulation.log`.

Lumol also offers more detailed configuration for logging output, for example if you only want to print errors and warnings, and redirect everything else to a file. This configuration happens in the `[log]` section of the input file. This section can contain either a single output `target`, or multiple `targets`.

Example

```
# Single log target
[log]
```

(continues on next page)

(continued from previous page)

```
target = "lumol.log"

# Multiple log targets
[log]
targets = [
    {target = "lumol.log"},
    {target = "<stdout>", level = "warning"}
]

# Multiple log targets, alternative syntax
[[log.targets]]
target = "lumol.log"

[[log.targets]]
target = "<stdout>"
level = "warning"
```

For multiple targets, the `targets` key must be an array of tables (either indicated by two brackets, i.e. `[[log.targets]]` or by defining the table `targets = [{target = ...}, {target = ...}, ...]`), containing individual target configuration with the same syntax as a single target.

The only required key is the `target` key, identifying where to write the messages. It is interpreted as path to a file, except for the two special cases of `<stdout>` and `<stderr>` that are used to write messages to the standard terminal output stream or error stream respectively.

```
# Write all messages to the standard error stream
[log]
target = "<stderr>"
```

Optional keys are `level` and `append`. The `level` key controls which messages are sent to the specified output and defaults to `info`. Available levels are the following:

- `error`: only error messages;
- `warning`: error and warning messages;
- `info`: error, warnings and informational messages;

The `debug` (debug messages) and `trace` (very verbose debug) are also available, but mainly intended for developers.

The `append` key is a boolean value (`true` or `false`) only used for files, and controls whether to overwrite the file or append new messages at the end of an existing file. It defaults to `false`, meaning that the file is overwritten by every simulation run.

```
[log]
# Use multiple targets
targets = [
    # Print warnings to the standard output stream
    {target = "<stdout>", level = "warning"},
    # Save all messages to the 'lumol.log' file
    {target = "lumol.log"},
    # Save debug messages to 'debug.log', keeping the file across simulation
    # runs.
    {target = "debug.log", level = "debug", append = true},
]
```

4.3 `[[systems]]` section

Let's talk about how you can set up your system. The system contains information about:

- the configuration, i.e. the positions (and velocities) of your atoms;

- which atoms are connected (bonded) to form molecules;
- how atoms will interact with each other;
- and the simulation cell (i.e. volume).

All these details are listed after the `[[systems]]` keyword.

Setting the initial configuration

A convenient way to get initial atom positions is by reading them from a file using the `file` key:

```
[[systems]]
file = "data/water.pdb"
```

Lumol will read the file to build the system accordingly. If the file is a trajectory containing multiple steps, only the first frame is used. Under the hood, we utilize chemfiles to parse the data and thus we can read in plenty different file formats. All possible formats are listed in the chemfiles documentation.

From the file, we will read in the unit cell, the atomic positions, the atomic masses, and use the atomic types as particles names. We will also read the list of bonds from the topology.

Initializing velocities

For molecular dynamics (MD) simulations you need initial positions and initial velocities of all atoms in your system. If no velocities are present within the read in configuration you can use the `velocities` key to initialize the velocities in the following way:

```
[[systems]]
file = "data/water.xyz"
topology = "topology.pdb"
velocities = {init = "300 K"}
```

where the `init` key will take the temperature as *string*. The velocities will be initialized from a Boltzmann distribution at the given temperature. Monte Carlo simulations will not make any use of velocities since transition probabilities (i.e. how the system evolves) are based on the positions (and the underlying interactions) only.

Setting the simulation cell

To set up the (initial) simulation cell you can use the *cell* key. This key is only needed if the configuration file does not contain this information (for example XYZ file), or if you want to override the cell from the file.

We offer three different ways to set the cell:

- `cell = <length>` creates a cubic unit cell with the given side length. `<length>` should be a numeric value (no quotation marks) in Angstrom.
- `cell = []` creates an infinite unit cell, without boundaries. This can be used when periodic boundary conditions are undesirable, for example to simulate aggregates in the void;
- `cell = [<a>, <b>, <c>]` creates an orthorhombic unit cell. You should provide the lengths of the cell, `<a>`, `<b>`, and `<c>` as numeric values in Angstrom.
- `cell = [<a>, <b>, <c>, <alpha>, <beta>, <gamma>]` creates a triclinic unit cell with the given side lengths and angles. `<a>`, `<b>`, and `<c>` should be numeric values in Angstrom and `<alpha>`, `<beta>`, and `<gamma>` numeric values in degree.

Note: In an TOML array, all values have to have the same type. `cell = [24, 24, 76]` will work since we use all integer values, while `cell = [24., 24., 76]` will throw an error.

4.4 Interactions input

Interactions describe the energies between atoms - or more general - between interaction sites. These energies arise due to covalent bonding of atoms to form molecules, short-range van der Waals or long-range Coulombic energies. To describe interactions we use *potentials* which are functions that take a set of parameters and geometrical coordinates (for example distances or angles) as input and yield energies and forces. A set of potential functions and parameters to describe energies and forces of a molecule is also often called *force field*. We will use the terms interactions and force field interchangeably.

To be more specific, we distinguish between the following contributions:

- `pairs` are van der Waals interactions between pairs of atoms;
- `bonds` describe the energy between bonded atoms;
- `angles` and `dihedrals` describe energy contributions due to bending and twisting of bonded atoms;
- `coulomb` and `charges` describe long-range contributions due to electrostatic interactions;
- the `global` section describes additional parameter that apply to all the energy contributions.

Information about interactions for `pairs`, `bonds`, `angles` and `dihedrals` are organized as TOML tables. The `coulomb` section contains information about the treatment of long-range electrostatic interactions and the `charges` section defines the partial charges of the atoms.

Organizing interactions

You can specify interactions between atoms in two ways: either inside the main input file or in a separate file. For example take a look at the input file for the flexible SPC water model where we put all interactions directly into our main input file:

```toml
# system configuration: initial positions, topology and cell
[[systems]]
file = "data/water.xyz"
topology = "topology.pdb"
cell = 40

# intermolecular potentials
[systems.potentials.pairs]
O-O = {type = "lj", sigma = "3.165 A", epsilon = "0.155 kcal/mol"}
H-H = {type = "null"}
O-H = {type = "null"}

# intramolecular potentials
[systems.potentials.bonds]
O-H = {type = "harmonic", k = "1059.162 kcal/mol/A^2", x0 = "1.012 A"}

[systems.potentials.angles]
H-O-H = {type = "harmonic", k = "75.90 kcal/mol/deg^2", x0 = "113.24 deg"}

# ... additional interactions omitted
```

As you can see, there is a lot of bracket notation going on here. First, in `[systems.potentials.xxx]`, the `potentials` key is actually a nested table of `systems` indicated by the dot notation. Accordingly, `pairs`, `bonds`, `angles`, etc. are nested tables of `potentials`. Second, `{type = "harmonic", k = "75.90 kcal/mol/deg", x0 = "113.24 deg"}` is the notation for an inline table.

Input files can get very big and hard to read when you simulate complex systems with a large number of different atoms. In these scenarios it may be better to define a separate input file for your interactions like so:

```toml
[[systems]]
file = "data/water.xyz"
topology = "topology.pdb"
```

(continues on next page)

(continued from previous page)

```
cell = 40
potentials = "water.toml"
```

Here, the `potentials` key contains a string that is interpreted as the path to another input file containing only definitions of interactions. This way, you can build your own library of force field files.

Non-bonded interactions

Cutoff treatment

When computing the energy and forces for non-bonded pair interactions, Lumol uses a cutoff radius r_c. This means that the force and energy associated with any pair at a distance bigger than r_c will be zero. We can use two different cutoff schemes, presented in the following section.

In the potentials input file, the cutoff should be specified for all the pairs. It can be specified once for all the pairs in the `global` section, and then overridden for specific interactions:

```
# Use a simple cutoff with a radius of 10 A for all pair interactions
[global]
cutoff = "10 A"

[pairs]
A-A = {type = "lj", sigma = "3 A", epsilon = "123 kJ/mol"}
B-B = {type = "lj", sigma = "3 A", epsilon = "123 kJ/mol"}

# Except for this one, use a shifted cutoff with a radius of 8 A
[pairs.A-B]
type = "lj"
sigma = "3 A"
epsilon = "123 kJ/mol"
cutoff = {shifted = "8 A"}
```

Simple cutoff (potential truncation)

This scheme just sets the potential $U(r)$ to zero for any distance bigger than rc:

$$V(r) = \begin{cases} U(r) & r <= rc \\ \\ 0 & r > rc \end{cases}$$

To use this potential truncation, we specify a string containing the cutoff distance as the `cutoff` value.

```
[global]
cutoff = "10 A"

[pairs]
O-O = {type = "lj", x0 = "3 A", k = "5.9 kJ/mol/A^2", cutoff = "8 A"}
```

Truncation with energy shift

The potential U can be additionally shifted to make sure it is continuous at $r = rc$. This is important for molecular dynamics, where a discontinuity means an infinite force in the integration.

$$V(r) = \begin{cases} U(r) - U(rc) & r <= rc \\ \\ 0 & r > rc \end{cases}$$

In the input, this uses a table containing the shifted value, which must be a string containing the cutoff radius.

```
[global]
cutoff = {shifted = "8 A"}

[pairs]
O-O = {type = "lj", x0 = "3 A", k = "5.9 kJ/mol/A^2", cutoff = {shifted = "10 A"}}
```

Tail correction

Tail corrections (also called long range corrections) are a way to account for the error we introduce by cutting off the potential. If the simulated system is homogeneous and isotropic beyond the the cutoff (if the pair distribution function $g(r)$ is 1 after the cutoff) we have an expression for the corrections we can evaluate. For a potential $V(r)$ with associated forces $\vec{f}(r)$, the missing energy and virial (which is used to compute instantaneous pressure) expressions are at density ρ:

$$U_{\text{tail}} = 2\pi\rho \int_{rc}^{+\infty} r^2 V(r)\, dr,$$

$$P_{\text{tail}} = 2\pi\rho^2 \int_{rc}^{+\infty} r^2 \vec{r} \cdot \vec{f}(r)\, dr.$$

In the input, these additional energetic and pressure terms are controlled by the `tail_correction` keyword, which can be placed either in the `[global]` section, or in any specific `[pairs]` section.

```
# Use tail corrections for every pair interaction
[global]
tail_correction = true

# Except for this one.
[pairs]
O-O = {type = "lj", x0 = "3 A", k = "5.9 kJ/mol/A^2", tail_correction = false}
```

Potentials computation

The same potential function (Lennard-Jones, Harmonic, *etc.*) can be computed with different methods: directly, by shifting at the cutoff distance, using a table interpolation, *etc.* This is the purpose of computation. The default way is to use the mathematical function corresponding to a potential to compute it. To use a different type of computation, the `computation` keyword can be used in the `[pairs]` section.

Table interpolation

Another way to compute a potential is to compute it on a regularly spaced grid, and then to interpolate values for points in the grid. In some cases, this can be faster than recomputing the function every time.

This can be done with the `table` computation, which does a linear interpolation in regularly spaced values in the `[0, max)` segment. You need to provide the `max` value, and the number of points `n` for the interpolation:

```
[pairs.O-O]
type = "lj"
sigma = "3 A"
epsilon = "123 kJ/mol"
computation = {table = {max = "8 A", n = 5000}}
```

Electrostatic interactions

When some particles in a system are charged, they interact with a Coulomb potential:

$$V(x) = \frac{Z_i Z_j}{4\pi\epsilon r_{ij}},$$

where $Z_{i,j}$ are the net charges of the particles, r_{ij} the distance between them and ϵ the dielectric permittivity of the current medium (usually the one of void). Because this potential goes to zero at infinity slower than $1/r^3$, it can not be computed in periodic simulations using a cutoff distance. This section present the available solvers for electrostatic interactions.

In the input files, electrostatic interactions are specified with two sections: the `[charges]` section sets the values of the charges of the atoms in the system, and the `[coulomb]` section sets the solver to use for the interaction.

Charge section

Charges for the particles in the system are set in a `[charges]` section in the potential input file. This section should contain multiple `name = <charge>` entries, one for each charged particle in the system.

```
# Some salt here
[charges]
Na = 1
Cl = -1
```

Ewald solver

Ewald's idea to compute electrostatic interactions is to split the interaction into a short-range term which can be handled with a cutoff scheme; and a long range term that can be computed using a Fourier transform. For more information about the Ewald summation and its variants, see [Frenkel2002].

The `[coulomb]` section for using an Ewald solver looks like this in the input file:

```
[coulomb]
ewald = {cutoff = "9 A", accuracy = 1e-5}
```

The `cutoff` parameter specifies the cutoff distance for the short-range and long-range interactions splitting. The `accuracy` parameter is used to request a relative relative error in forces, and should be smaller than 1. The `accuracy` is used to set the other Ewald parameters (alpha and kmax).

It is also possible to manually set the Ewald parameters:

```
[coulomb]
ewald = {cutoff = "9 A", kmax = 7, alpha = "0.33451 A^-1"}
```

The `kmax` parameter gives the number of points to use in the reciprocal space (the long-range part of interactions). Usually 7-8 is a good value for pure water, for a very periodic charges distribution (like a crystal) a lower value, such as 5 is sufficient, and for more heterogeneous system, higher values of `kmax` are needed. The `alpha` parameter specifies the width of the charges spreading used to smooth the distribution in reciprocal space. A good value of `alpha` is one that satisfies $\exp\left(-\alpha\frac{L}{2}\right) << 1$. If only `kmax` is provided in the input file, the default value of π/cutoff is used for `alpha`.

Wolf solver

The Wolf summation method is another method for computing electrostatic interactions presented in [Wolf1999]. This method replaces the expensive computation in reciprocal space from Ewald by a corrective term, and can be expressed as a converging sum over the charged pairs in the system.

It is accessible using the `wolf` keyword in the input files:

```toml
[coulomb]
wolf = {cutoff = "11 A"}
```

The only parameter is a `cutoff`, which - as a rule of thumb - should be larger than the corresponding cutoff from Ewald summation. For example, `cutoff = "11 A"` should be suitable for pure water.

[Frenkel2002] Frenkel, D. & Smith, B. *Understanding molecular simulation.* (Academic press, 2002).

[Wolf1999] Wolf, D., Keblinski, P., Phillpot, S. R. & Eggebrecht, J. *Exact method for the simulation of Coulombic systems by spherically truncated, pairwise 1/r summation.* The Journal of Chemical Physics **110**, 8254 (1999).

Available potentials

This section is a list of all the available potentials in Lumol, with the associated parameters. All potentials have to provide additional parameters in there definition, as a TOML table.

Using inline tables is the easiest way to do so:

```toml
# Additional parameters here are 'sigma' and 'epsilon'.
[pairs]
A-B = {type = "lj", sigma = "3 A", epsilon = "123 kJ/mol"}
```

Another option is to use a separated TOML table, for example when there are too many parameters to fit on a line

```toml
[pairs.A-B]
type = "lj"
sigma = "3 A"
epsilon = "123 kJ/mol"
```

The same potential can be used for either pairs (at distance r); or for angles (at angle ϕ). In all the formulas, the x parameter represents either a distance or an angle.

Null potential

This potential is always 0, for all values of x. It should be used to remove interactions between atoms in a pair/bond/angle/dihedral that are present in the system but should not be interacting.

This potential can be used by specifying the `null` key with an empty table `{}` as value.

```toml
[pairs]
O-O = {type = "null", }
```

Lennard-Jones potential

The Lennard-Jones potential is a classical potential for pair interactions expressed as:

$$V(x) = 4\epsilon \left[\left(\frac{\sigma}{x}\right)^{12} - \left(\frac{\sigma}{x}\right)^{6} \right].$$

The Lennard-Jones potential is defined using the `lj` key. The parameters are `sigma` (σ) and `epsilon` (ϵ), which should be provided as strings.

```toml
[pairs]
O-O = {type = "lj", sigma = "3.16 A", epsilon = "0.155 kcal/mol"}
```

Buckingham potential

The Buckingham potential is a potential for pair interactions expressed as:

$$V(x) = A \exp(-r/\rho) - \frac{C}{r^6}.$$

The potential type keyword is `buckingham`, and the parameters `A`, `rho` (ρ) and `C` should be provided as strings.

```
[pairs]
C-C = {type = "buckingham", A = "40 kJ/mol", C = "120e-6 kJ/mol/A^6", rho = "3.0 A
↪"}
```

Born-Mayer-Huggins potential

The Born-Mayer-Huggins potential is a potential for pair interactions, used in particular for halide alkali. Its expression is:

$$V(x) = A \exp\left(\frac{\sigma - r}{\rho}\right) - \frac{C}{r^6} + \frac{D}{r^8}.$$

The potential type keyword is `born`, and the parameters `A`, `C`, `D`, `sigma` (σ) and `rho` (ρ) should be provided as strings.

```
[pairs.Li-Li]
type = "born"
A = "40 kJ/mol"
C = "120e-6 kJ/mol/A^6"
D = "23e-6 kJ/mol/A^8"
rho = "3.0 A"
sigma = "2.2 A"
```

Harmonic potential

The Harmonic potential is usually used for intramolecular interactions such as bonds, angles or dihedrals. It is expressed as:

$$V(x) = \frac{1}{2} k \left(x - x_0\right)^2$$

The potential type keyword is `harmonic`, and the parameters are `k` and `x0`, provided as strings.

```
[bonds]
O-H = {type = "harmonic", k = "1054.2 kcal/mol/A^2", x0 = "1.0 A"}

[angles]
H-O-H = {type = "harmonic", k = "75.9 kcal/mol/rad^2", x0 = "109.5 deg"}
```

Cosine-Harmonic potential

This potential is usually used for angles and dihedral angles interactions, because it presents a 2π periodicity. It is expressed as:

$$V(x) = \frac{1}{2} k \left(\cos x - \cos x_0\right)^2$$

The potential type keyword is `cosine-harmonic`, and the parameters `k` and `x0` should be provided as strings.

```
[angles]
H-C-H = {type = "cosine-harmonic", k = "67 kJ/mol", x0 = "120 deg"}
```

Torsion potential

This potential is usually used for dihedral interactions. It is expressed as:

$$V(x) = k\,(1 + \cos(nx - \delta))$$

The potential type keyword is `torsion`, and the parameters `k` and `delta` (δ) should be provided as strings, and `n` should be provided as an integer.

```
[dihedrals]
C-C-C-C = {type = "torsion", k = "40 kJ/mol", delta = "120 deg", n: 4}
```

Morse potential

This potential is usually used for intramolecular interaction such as bonds, angles or dihedrals. It is a better approximation for the vibrational structure of the molecule than the Harmonic potential. It is expressed as:

$$V(x) = \text{depth} \times (1 - \exp(-A(x - x_0))^2$$

The potential type keyword is `morse`, and the parameters `A`, `x0` and `depth` should be provided as strings.

```
[pairs]
A-B = {type = "morse", depth = "40 kJ/mol", A = "2.0 A^-1", x0 = "1.3 A"}
```

For angles and dihedral angles, `x0` and `A` should be provided in angle units:

```
[pairs]
A-B = {type = "morse", depth = "40 kJ/mol", A = "2.0 rad^-1", x0 = "109.7 deg"}
```

Gaussian potential

This potential is usually used to describe energy wells and is expressed as:

$$V(r) = -A\exp(-Br^2)$$

The potential type keyword is `gaussian`, and the parameters `A` (well depth) and `B` (well width) should be provided as strings. `B` has to be positive.

```
[pairs]
A-B = {type = "gaussian", A = "8.0 kJ/mol", B = "0.2 A^-2"}
```

Mie potential

The Mie potential is a classical potential for pair interactions expressed as:

$$V(r) = \frac{n}{n-m}\left(\frac{n}{m}\right)^{m/(n-m)} \epsilon \left[\left(\frac{\sigma}{r}\right)^n - \left(\frac{\sigma}{r}\right)^m\right]$$

The potential type keyword is `mie` and the parameters are `sigma` (σ), the particle diameter, and `epsilon` (ϵ), the energetic parameter, which should be provided as strings (with units). The repulsive exponent `n` and the attractive exponent `m` should be provided as numbers. The repulsive exponent `n` has to be larger than the attractive exponent `m`.

```
[pairs]
A-B = {type = "mie", sigma = "3 A", epsilon = "5.9 kJ/mol", n = 12.0, m = 6.0}
```

Restrictions

Some force fields define additional restrictions concerning which particles should interact together and which ones should not. For example, sometimes bonded particles should not interact through electrostatic potential, or some interactions should only be taken in account for particles not in the same molecule. The way to specify this is to use restrictions. Restrictions can be used in two places: in the `[pairs]` section, and in the `[coulomb]` section. In both cases, they are specified with the `restriction` keyword, and one of the possible values.

```
[pairs]
O-O = {type = "lj", sigma = "3 A", epsilon = "123 kJ/mol", restriction = {scale14␣
↪= 0.5}}

[coulomb]
ewald = {cutoff = "8 A", kmax = 6}
restriction = "intermolecular"
```

The possible values for `restriction` are:

- `"intramolecular"` or `"intra-molecular"` to act only on particles that are in the same molecule;
- `"intermolecular"` or `"inter-molecular"` to act only on particles that are **NOT** in the same molecule;
- `"exclude12"` to exclude particles directly bonded together;
- `"exclude13"` to exclude particles directly bonded together or forming an angle;
- `"exclude14"` to exclude particles directly bonded together; forming an angle or a dihedral angle;
- `{scale14 = <scaling>}` works like `exclude13`, *i.e.* intramolecular interactions between three neighboring particles (connected by two bonds) will not be computed. Additionally, interactions between the first and the forth (hence the `14` in `scale14`) particle will be computed, but using scaled energies and forces. This simply means that the energies and forces are multiplied (linear scaling) by the given scaling factor, which must be between 0 and 1.

4.5 `[[simulations]]` section

The way to propagate the system is defined in the `[[simulations]]` section of the input file. This section always contains at least two keys: `nsteps` specify the number of steps in the simulation, and the `simulations.propagator` table specify which propagator to use.

Here is an example of NPT molecular dynamics:

```
[[simulations]]
nsteps = 1_000_000

[simulations.propagator]
type = "MolecularDynamics"
timestep = "1 fs"
integrator = {type = "BerendsenBarostat", pressure = "100 bar", timestep = 1000}
thermostat = {type = "Berendsen", temperature = "400 K", timestep = 100}
```

Three propagators are currently implemented:

- A *Minimization* propagator, to minimize energy of a system before running another propagator;
- A *Molecular dynamics* propagator;
- A *Monte Carlo* propagator;

Specifying output information

Additionally, a simulation can also output the evolution of the system properties. Which properties are needed is given in the `outputs` array:

Example

```
[[simulations]]
nsteps = 1_000_000
outputs = [
    {type = "Trajectory", file = "filename.xyz", frequency = 100},
    {type = "Energy", file = "energy.dat", frequency = 200},
    {type = "Custom", file = "custom.dat", template = "{vx[3] / mass[3]}"},
]

[simulations.propagator]
...
```

This array is an array of tables, containing three keys:

- the `type` of output
- the `file` to write the output to
- the `frequency` of the output.

The `file` is the path where the output will be written to. The frequency is a number and the output will be written every `frequency` steps to the file. Except for the `Trajectory` output, all files are formatted with header lines starting with a `#`, and containing information about the quantities and the units used for the output followed by multiple lines containing the step and associated quantities. The available outputs are the following:

- The `Energy` output will write the potential, kinetic and total energy;
- The `Cell` output will write the unit cell parameters, lengths and angles;
- The `Properties` output will write the volume, the instant pressure (computed from the virial equation) and the instant temperature of the system;
- The `Stress` output will write all the components of the stress tensor (computed from the virial equation);
- The `Trajectory` output should be used to write a trajectory. The format of the trajectory will be guessed from the `file` extension. Supported formats are documented in chemfiles documentation.
- The `Custom` output is the most powerful one, taking an user-provided template string and using it to output data. The template should be given as a string with the `template` key in the TOML input file.

Here are some examples of custom output templates:

- A constant string is reproduced as it is: `some data`;
- Anything in braces is replaced by the corresponding values: `{pressure} {volume}` will write the pressure and volume;
- Mathematical operators are allowed in braces: `{pressure / volume}` will print an entry containing the quotient of pressure and volume. You can use `+`, `-`, `/`, `*`, `^` for exponentiation and parentheses;
- Some properties are arrays of atomic properties: `{x[0] + y[20]}` will print the sum of the x position of the 0'th atom and the y position of the 20'th position;
- Finally, all the properties are given in the internal units but one can specify a different unit: `x[0] / nm`.

Here is a list of all properties available to custom outputs:

- Atomic properties: *x*, *y* and *z* for cartesian coordinates, *vx*, *vy* and *vz* for cartesian components of the velocity, *mass* for the atomic mass, *charge* for the atomic charge.
- Physical properties: *pressure*, *volume*, *temperature*, *natoms*, stress tensor components: *stress.xx*, *stress.yy*, *stress.zz*, *stress.xy*, *stress.xz*, *stress.yz*, simulation *step*.

- Unit Cell properties: *cell.a*, *cell.b*, *cell.c* are the unit cell vector lengths; *cell.alpha*, *cell.beta* and *cell.gamma* are the unit cell angles.

Minimization

You can run a minimization by setting the propagator `type` to `Minimization`. The unique needed key is the `minimizer` algorithm to use for this simulation; you can also optionally set the criteria for minimization convergence.

```
[simulations.propagator]
type = "Minimization"
minimizer = {type = "SteepestDescent"}
criteria = {energy = "1e-5 kJ/mol", force2 = "1e-5 kJ^2/mol^2/A^2"}
```

The single minimization algorithm implemented is the steepest descent algorithm, that updates the coordinates of the atom following the energy gradient.

The minimization stops when the energy difference between the previous and the current step is lower than the `energy` criterion, or when the maximal squared norm of the atomic force is lower than the `force2` criterion.

Molecular dynamics

A molecular dynamics simulation is started by setting the propagator `type` to `"MolecularDynamics"`. The only needed key is the `timestep`, which is the time step to use in the integration of forces and velocities to positions.

```
[[simulations]]
nsteps = 1_000_000

[simulations.propagator]
type = "MolecularDynamics"
timestep = "1 fs"
integrator = {type = "BerendsenBarostat", pressure = "100 bar", timestep = 1000}
thermostat = {type = "Berendsen", temperature = "400 K", timestep = 100}
```

Other options are the `integrator` key to use another integration scheme, the `thermostat` key to set a thermostat, and the `controls` key to add some additional control algorithm to the simulation.

Integrators

Integrators are algorithms that propagate the forces acting on the particles to compute their motions. The simplest ones performs an NVE integration, but some integrators allow to work in different ensembles. All NVE integrators can be turned into NVT integrators by adding a *thermostat* to the simulation. In the input, if the `integrator` key is absent, the default integrator is a Velocity-Verlet integrator.

Velocity-Verlet integrator

Velocity-Verlet is the most common NVE integrator for molecular dynamics. See this page for more information about the algorithm.

In the input, it can be specified by using the `VelocityVerlet` integrator type:

```
[simulations.propagator]
type = "MolecularDynamics"
timestep = "1 fs"
integrator = {type = "VelocityVerlet"}
```

Verlet integrator

Verlet algorithm is another simple NVE integrator. See this page for more information. Most of the time, the Velocity-Verlet algorithm is preferable, since it produces more precise velocities.

In the input, it can be specified by using the `Verlet` integrator type:

```
[simulations.propagator]
type = "MolecularDynamics"
timestep = "1 fs"
integrator = {type = "Verlet"}
```

Leap-Frog integrator

The Leap-Frog algorithm is a third NVE integrator. See this page for details about the algorithm.

In the input, it can be specified by using the `LeapFrog` integrator type:

```
[simulations.propagator]
type = "MolecularDynamics"
timestep = "1 fs"
integrator = {type = "LeapFrog"}
```

Berendsen barostat

The Berendsen barostat integrator algorithm use the Berendsen barostat with a Velocity-Verlet integrator to achieve NPT integration. It must be use together with a thermostat, preferentially the Berendsen thermostat. See this page for more information about the algorithm.

This algorithm exists in two versions: an isotropic one and an anisotropic one. The isotropic version of the barostat scale all the cell parameter by the same value using the scalar pressure. The anisotropic version scale the different cell parameters by different values, using the stress tensor instead.

In the input, the isotropic barostat can be specified by using the `BerendsenBarostat` integrator type:

```
[simulations.propagator]
type = "MolecularDynamics"
timestep = "1 fs"
integrator = {type = "BerendsenBarostat", pressure = "100 bar", timestep = 1000}
thermostat = {type = "Berendsen", temperature = "400 K", timestep = 100}
```

The `pressure` key specify the target pressure for the simulation, and the `timestep` is the relaxation time step of the barostat.

The anisotropic version of the Berendsen barostat can be specified by using the `AnisoBerendsenBarostat` integrator type:

```
[simulations.propagator]
type = "MolecularDynamics"
timestep = "1 fs"
integrator = {type = "AnisoBerendsenBarostat", pressure = "100 bar", timestep =␣
↪1000}
thermostat = {type = "Berendsen", temperature = "400 K", timestep = 100}
```

The `pressure` key specify the target hydrostatic pressure for the simulation, and the `timestep` is the relaxation time step of the barostat.

In both cases, the barostat time step is expressed in fraction of the main integration time step. Using a main time step of 2 fs and a barostat time step of 1000 will yield an effective relaxation time of 2000 fs or 2 ps.

Thermostats

Thermostats are algorithms used to maintain the temperature of a system at a given value. They are specified in the input by the `thermostat` key.

Canonical Sampling through Velocity Rescaling (CSVR)

The CSVR thermostat implement the algorithm in [Bussi2012]. This algorithm provides a cheap, efficient and correct thermostat, sampling the right dstribution in the canonical (NVT) ensemble. In the input, it is declared with the `CSVR` thermostat type, a target `temperature` value, and a `timestep`. The time step control the relaxation rate of this thermostat, and is expressed in fraction of the main integration time step.

```
[simulations.propagator]
type = "MolecularDynamics"
timestep = "1 fs"
thermostat = {type = "CSVR", temperature = "400 K", timestep = 100}
```

Berendsen thermostat

The Berendsen thermostat is described here, and provide a simple exponential relaxation of the temperature to a target value. In the input, it is declared with the `Berendsen` thermostat type, a target `temperature` value, and a `timestep`. The time step is expressed in fraction of the main integration time step.

```
[simulations.propagator]
type = "MolecularDynamics"
timestep = "1 fs"
thermostat = {type = "Berendsen", temperature = "400 K", timestep = 100}
```

Rescaling thermostat

A rescaling thermostat is the simplest thermostat algorithm possible: it just rescale all the velocities to set the temperature to the wanted value. It can be useful for equilibration as it converges quickly. In the input, it is specified by the `Rescale` thermostat type, a target `temperature` value, and a `tolerance` value. The tolerance value is optional, and is used to let the system fluctuate around the wanted temperature: while the instant temperature is inside the `[temperature - tolerance : temperature + tolerance]` range, no rescale happen.

```
[simulations.propagator]
type = "MolecularDynamics"
timestep = "1 fs"
thermostat = {type = "Rescale", temperature = "250 K", tolerance = "10 K"}
```

Controls

Control algorithm are supplementary steps that modify the system to ensure some invariant, or apply some constraint. They are specified in the `controls` array, by giving a control `type`. The `every` key specifies that the algorithm should only be run every `n` step of the simulation (optional, defaults to 1).

```
[simulations.propagator]
type = "MolecularDynamics"
timestep = "1 fs"
controls = [
    # Remove global rotation of the system every 4 timestep
    {type = "RemoveRotation", every = 4}
]
```

- The `RemoveTranslation` control removes the global system rotation;
- The `RemoveRotation` control removes the global system translation.
- The `Rewrap` control rewraps all molecules' centers of mass to lie within the unit cell. Individual atoms in a molecule may still lie outside of the cell.

Monte Carlo

- Needed keys:
  - `type = "MonteCarlo"`
  - `temperature` (string): System temperature. The string contains the temperature with unit.
- Optional keys:
  - `update_frequency` (positive integer): After this number of steps of a move, `delta` values for this move are updated. Updates use statistics of a moves' acceptance ratio so it is recommended to choose a sufficiently high number (>100).

If you want to perform a Monte Carlo simulation, you have to set the propagator `type` to `"MonteCarlo"`. Every Monte Carlo simulations needs a `temperature` and a set of `moves` (this set can consist of a single move).

You can think of a "move" as a specific instruction to generate a new trial configuration. For example: "translate a single molecule in the system", "rotate a molecule", or "change the cell size". If a move is accepted (based on an acceptance criterion), the trial configuration becomes the new configuration. If the move is rejected, the system stays in its current configuration. You can add multiple moves so that the ensemble of your choice is sampled.

Here is a list of all moves that are currently implemented in Lumol:

- *Translate*: Change the center of mass position of a molecule.
- *Rotate*: Perform a rotation of a molecule about its center of mass.
- *Resize*: Change the size of the simulation cell.

Currently, all Monte Carlo simulations are carried out using Metropolis acceptance criteria.

You can add all necessary information after the `[simulations.propagator]` label.

Naturally, Monte Carlo simulations are carried out at constant temperature which is set using the `temperature` key.

Different from Molecular Dynamics, Monte Carlo simulations don't carry information about the velocities of particles. As a consequence we cannot access temperature from the kinetic energy.

Example

A sample input for a Monte Carlo simulation (in the NPT ensemble) can look like so:

```
[simulations.propagator]
type = "MonteCarlo"
temperature = "500 K"

moves = [
    {type = "Translate", delta = "1 A", frequency = 2},
    {type = "Rotate", delta = "20 deg", molecule = "CO2.xyz"},
    {type = "Resize", pressure = "10 bar", delta = "3 A^3", frequency = 0.001},
]
```

Moves

All `moves` are specified as inline tables. You can add a move using the `type` key with the name of the move.

`moves` accept an optional `frequency` parameter. During a Monte Carlo simulation it is very important that a move is selected randomly from the whole set. You can increase the chance to pick a certain move (compared to all other moves) by assigning a high `frequency` to it. If you don't specify a `frequency`, it is set to one.

Lumol normalizes frequencies after all moves are added. The easiest way to handle frequencies is to use relative values. We will explain this below in the given examples.

Some moves can be specified to act on a single molecule or particle type. These moves accept a `molecule` key whose value is a path to a configuration file that can be read by chemfiles.

You can add the same move multiple times. For example, you can assign different amplitudes for different species in a mixture to make sampling more efficient. You can also use the `molecule` type to freeze a species by assigning a move for all but the frozen species.

If you specify a molecule, it will be selected with the following algorithm:

- Read the first frame of the file;
- If the file does not contain any bonding information, try to guess the bonds;
- Use the first molecule of the frame.

`moves` that use a displacement (`delta`) can be added with the `target_acceptance` key. After a specific number of times a move was called (`update_frequency`), we compute the acceptance ratio for the current `delta` value, *i.e.* how often the move was accepted versus how often a move was attempted. If the current acceptance is far away from the `target_acceptance`, we compute a new value of `delta` based on the current acceptance. A `target_acceptance` can only be used in conjunction with the `update\_frequency` key that specifies the frequency between updates.

Sometimes a given acceptance value cannot be achieved. Either due to limits of the adjusted `delta` value (it makes no sense to rotate a particle by more than 180° or to translate it by multiple values of the cutoff range) or due to the nature of the system.

To summarize, using an adjustable displacement, we can increase the efficiency of our simulation, but strictly speaking we violate detailed balance and therefore the Markov chain. To make sure you get correct results from your simulations, we recommend to use adjustable displacements *only for equilibration runs*. You can then take the resulting values for `delta` and use them for a production run, where no further adjustments are made.

Example

```
# Equilibration of a protein in water.
[simulations.propagator]
type = "MonteCarlo"
temperature = "300 K"
# we update the maximum displacement `delta` after a move was called 500 times
update_frequency = 500

moves = [
    # we have much more water in the system so we want to move it more often
    # hence we set the `frequency = 100`
    # after 500 calls to this translation move, we adjust `delta` to get to␣
↪approximately 50% acceptance
    {type = "Translate", delta = "2 A", molecule = "H2O.xyz", frequency = 100,␣
↪target_acceptance = 0.5},
    # the single protein will be displaced only by a small distance `delta = "0.05␣
↪A"` during the whole run
    {type = "Translate", delta = "0.05 A", molecule = "protein.pdb", frequency = 1}
↪,
]
```

Translate

The `Translate` move changes the position of a single, randomly selected molecule by adding a random displacement vector to its center of mass.

- Needed keys:
  - `type = "Translate"`
  - `delta` (string): Maximum amplitude for displacement.
- Optional keys:
  - `frequency` (float): Move frequency.
  - `molecule` (string): Select only the specified molecule type. The string contains the path to the configuration file of the molecule.
  - `target_acceptance` (float): The target acceptance for this move. Value has to be greater than zero and smaller than one. Can only be used in conjunction with `update_frequency`.

If the `molecule` key is used, the move will only apply to one molecule type. If not, the move will apply to all molecule types in the system. The `delta` key is the maximum magnitude of the translation vector. The conjugated string contains the value with unit of distance.

Example

```
[simulations.propagator]
type = "MonteCarlo"
temperature = "500 K"
moves = [
    # Define a translation for all molecules in the system, including He.
    {type = "Translate", delta = "1 A", frequency = 2},
    # For He, pick a larger displacement with half the frequency of the
    # first move. Now there is a 66% chance to pick *any* molecule
    # and translate it by up to 1 A. There is a 33% chance to pick He (and only He)
    # and translate it by up to 10 A.
    {type = "Translate", delta = "10 A", molecule = "He.xyz"},
]
```

Rotate

The `Rotate` move randomly rotates a single molecule around its center of mass.

- Needed keys:
  - `type = "Rotate"`
  - `delta` (string): Maximum angle for rotation.
- Optional keys:
  - `frequency` (float): Move frequency.
  - `molecule` (string): Select only the specified molecule type. The string contains the path to the configuration file of the molecule.
  - `target_acceptance` (float): The target acceptance for this move. Value has to be greater than zero and smaller than one. Can only be used in conjunction with `update_frequency`.

If the `molecule` key is used, the move will only apply to one molecule type. If not, the move will apply to all molecules in the system. The `delta` key is the maximum angle. The conjugated string contains the value and the unit of either radians or degrees (`rad` or `deg`).

Example

```
[simulations.propagator]
type = "MonteCarlo"
temperature = "500 K"
moves = [
    {type = "Rotate", delta = "3 deg", frequency = 2},
]
```

Resize

The `Resize` move can be used to isotropically change the systems' volume.

- Needed keys:
  - `type = "Resize"`
  - `pressure` (string): Target pressure.
  - `delta` (string): Amplitude.
- Optional keys:
  - `frequency` (float): Move frequency.
  - `target_acceptance` (float): The target acceptance for this move. Value has to be greater than zero and smaller than one. Can only be used in conjunction with `update_frequency`.

For a given `pressure`, the volume will fluctuate during the simulation. We can use this move to sample an isobaric-isothermal ensemble. The `delta` key sets the maximum amplitude of the volume change in units of cubic length.

By changing the volume, we effectively change all (center of mass) positions at once. This makes `Resize` moves computationally expensive and we recommend to use a comparatively low value for the `frequency`. As a rule of thumb, for a system containing N particles, every $N + 1$'th move should be a `Resize` move, since a single volume change is approximately as expensive as N particle translations or rotations.

Example

```
# Simulation of 500 molecules.
[simulations.propagator]
type = "MonteCarlo"
temperature = "500 K"
moves = [
    {type = "Translate", delta = "1 A", frequency = 250},
    {type = "Rotate", delta = "20 deg", frequency = 250},
    {type = "Resize", pressure = "10 bar", delta = "3 A^3", frequency = 1},
]
```

As mentioned above, frequencies are normalized. In this example, 501 moves consist of 250 translations, 250 rotations and a single resizing of the cell *on average* (remember, moves are picked at random with their respective frequency). Setting up a move set like we did in this example is very convenient and in literature you'll often find the term "cycle" (here, 1 cycle = 501 moves) to describe such a set of moves and respective frequencies.

5 Modifying lumol

This chapter discusses ways to extend or modify Lumol.

5.1 Adding a potential

In this tutorial we will teach you how to use your own potential function with Lumol. This tutorial consists of two parts. In the first part, we will implement the logic in a separate project using Lumol as an external library. The second part describes the neccessary steps to implement the logic into the core package of Lumol called `lumol-core`.

A potential is a function that describes the energy and force between interaction sites. In Lumol we differentiate between two types of potentials: First, there are potential functions that need the global state of the system, *i.e.* all positions, as input. The Ewald summation which is used to compute electrostatic interactions is an example for this kind of potential. We call them `GlobalPotentials`. And second, we have potentials that take a single geometric parameter as input. This geometric parameter can be for example a distance or an angle. Typical examples are Van-der-Waals potentials (*e.g.* Lennard-Jones) and potential functions describing covalent bonds

(*e.g.* harmonic potential, cosine potential, torsion, *etc.*). There are plenty of potentials falling into this category, hence in Lumol we simply call them `Potentials`.

Note: There are two kind of potentials. A `GlobalPotential` takes the global systems' state as input and a `Potential` takes a single scalar value (distance, angle, ...) as input.

In this tutorial we will focus on the implementation of the latter. More specific, we will implement a potential to compute van-der-Waals interactions between pairs of particles. We will make liberal use of the API documentation of both the Rust standard library as well as the Lumol API. Please note that we will point to other references, such as the Rust book, concerning general Rust concepts to keep this tutorial brief. If you have questions concerning Rust or Lumol, please don't hesitate to file an issue on github or join the discussion on our gitter.

The tutorial is structured as follows: First we will have a look at how `Potentials` are represented (what data structures are used) and what functionalities we have to implement. Then, we describe how we add those functionalities. We will write a small simulation program that makes use of our newly created potential. In the second part we talk about implementing the potential into Lumol's core. Rust offers beautiful utilities to add documentation inside the code. We will have a look at the documentation of currently implemented potentials to guide us. We will then add some tests to make sure that our implementation is correct. This concludes the bulk the work, but to make our new potential function accessible to all Lumol users we will also add a parsing function. Doing so, our potential can be conveniently specified from an input file. We conclude the tutorial by adding a short documentation to the user manual.

In this tutorial, we will implement a potential to describe non-bonded pair interactions, namely the Mie potential.

We have a lot to do. Ready? Let's go.

Note: We recommend you read the chapters concerning structs and traits in the rust book.

Traits used for potentials

As mentioned above, potential functions can be used to model all kinds of interactions between particles such as bonded and non-bonded interactions. In Lumol, `Potential` is a trait. To further distinguish between bonded interactions (bond lengths, angles and dihedrals) and non-bonded interactions, we use another trait (often called marker traits). Your possible options to further specialise a `Potential` are

- PairPotential for non-bonded two body interactions;
- BondPotential for covalent bonds interactions;
- AnglePotential for covalent angles interactions;
- DihedralPotential for covalent dihedral angles interactions.

For our Mie potential implementation, we will have to implement both the `Potential` as well as the `PairPotential` traits. If we wanted to implement a function that can be used as non-bonded as well as bond-length potential, we'd have to implement `Potential`, `PairPotential` as well as `BondPotential`. "Implementing a trait" means that we will define a struct for which we will add functions to satisfy the traits' requirements.

Let's start by having a look at the documentation for `Potential`: open the API documentation. As you can see from the trait, a `Potential` defines two functions, `energy` and `force` (we ignore the `Sync + Send` statement for now):

```
pub trait Potential: Sync + Send {
    fn energy(&self, x: f64) -> f64;
    fn force(&self, x: f64) -> f64;
}
```

`energy` will compute the interaction energy between atoms as a function of `x`. The force is defined as the negative derivative of the energy function with respect to `x`.

Both functions take a single, scalar argument and return a single scalar value. In our case `x` stands for the distance between two interaction sites. Note that only the function definitions – without a function body – are specified. We will have to implement these functions for our potential.

Implementation of the potential

We start by creating a new package using `cargo`:

```
cargo new potential
cd potential
```

Open `Cargo.toml` and add the lines

```
[dependencies]
lumol = {git = "https://github.com/lumol-org/lumol"}
```

to add the `lumol` crate as a dependency to the package. To test if everything works, run `cargo build` and check if an error occurs.

Defining the struct

For the first part of the tutorial, the complete code will be written into the `lib.rs` file.

The energy function of the Mie potential reads

$$u(x) = \varepsilon \frac{n}{n-m} \left(\frac{n}{m}\right)^{\frac{m}{n-m}} \left[\left(\frac{\sigma}{x}\right)^n - \left(\frac{\sigma}{x}\right)^m\right]$$

where x denotes the distance between two interaction sites i, j, with $x = x_{ij} = |\mathbf{r}_j - \mathbf{r}_i|$. The parameters of the potential are

- n, m the repulsive and attractive exponents, respectively,
- ε the energetic paramater,
- σ the particle diameter or structural parameter.

We start by defining the `struct` for our potential. Add the following lines to `lib.rs`:

```
use lumol::energy::{Potential, PairPotential};

#[derive(Clone, Copy)]
pub struct Mie {
    /// Distance constant
    sigma: f64,
    /// Exponent of repulsive contribution
    n: f64,
    /// Exponent of attractive contribution
    m: f64,
    /// Energetic prefactor computed from the exponents and epsilon
    prefactor: f64,
}
```

In the first two lines we define our imports from `Lumol`, following with our `Mie` structure. Notice that we don't store the `epsilon` value, instead we store an energetic prefactor that will make it easier to compute the potential.

$$\text{prefactor} = \varepsilon \frac{n}{n-m} \left(\frac{n}{m}\right)^{m/(n-m)}$$

Next, we implement a constructor function. That's usefull in this case since we want to compute the prefactor of the potential once before we start our simulation.

In Rust we typically use `new` for the constructors' name.

```
impl Mie {
    pub fn new(sigma: f64, epsilon: f64, n: f64, m: f64) -> Mie {
        if m >= n {
            panic!("The repulsive exponent n has to be larger than the attractive␣
↪exponent m")
        };
        let prefactor = n / (n - m) * (n / m).powf(m / (n - m)) * epsilon;
        return Mie {
            sigma: sigma,
            n: n,
            m: m,
            prefactor: prefactor,
        }
    }
}
```

Our function takes the parameter set as input, computes the prefactor and returns a `Mie` struct. Notice that it panics, for `n` smaller than or equal to `m`. The next step is to implement the `Potential` trait for `Mie`.

Implementing `Potential`

Add the following lines below the structs implementation.

```
impl Potential for Mie {
    fn energy(&self, r: f64) -> f64 {
        let sigma_r = self.sigma / r;
        let repulsive = f64::powf(sigma_r, self.n);
        let attractive = f64::powf(sigma_r, self.m);
        return self.prefactor * (repulsive - attractive);
    }

    fn force(&self, r: f64) -> f64 {
        let sigma_r = self.sigma / r;
        let repulsive = f64::powf(sigma_r, self.n);
        let attractive = f64::powf(sigma_r, self.m);
        return -self.prefactor * (self.n * repulsive - self.m * attractive) / r;
    }
}
```

`energy` is the implementation of the Mie potential equation shown above. `force` is the negative derivative of the energy with respect to the distance, `r`. To be more precise, the vectorial force can readily be computed by multiplying the result of `force` with the connection vector $\vec{r}$.

The next step is to make our `Potential` usable in Lumol's algorithms to compute non-bonded energies and forces. Therefore, we have to implement the `PairPotential` trait.

Implementing `PairPotential`

Let's inspect the documentation for `PairPotential`.

```
pub trait PairPotential: Potential + BoxClonePair {
    fn tail_energy(&self, cutoff: f64) -> f64;
    fn tail_virial(&self, cutoff: f64) -> f64;

    fn virial(&self, r: &Vector3D) -> Matrix3 { ... }
}
```

First, we can see that `PairPotential` enforces the implementation of `Potential` which is denoted by `pub trait PairPotential: Potential ...` (we ignore `BoxClonePair` for now, as it is automatically

implemented for us if we implement `PairPotential` manually). That makes sense from a didactive point of view since we said that `PairPotential` is a "specialization" of `Potential` and furthermore, we can make use of all functions that we had to implement for `Potential`.

There are three functions in the `PairPotential` trait. The first two functions start with `tail_`. These are functions to compute long range or tail corrections. Often, we introduce a cutoff distance into our potential beyond which we set the energy to zero. Doing so we intoduce an error which we can account for using a tail correction. We need two of these corrections, one for the energy, `tail_energy`, and one for the pressure (which uses `tail_virial` under the hood). For a beautiful derivation of tail corrections for truncated potentials, see here.

The third function, `virial`, already has its body implemented – we don't have to write an implementation for our potential.

We will omit the derivation of the formulae for tail corrections here but they are computed by solving these equations

$$\text{tail energy} = \int_{r_c}^{\infty} u(r) r^2 \mathrm{d}r$$

$$\text{tail virial} = \int_{r_c}^{\infty} \frac{\partial u(r)}{\partial r} r^3 \mathrm{d}r$$

The implementation looks like so

```
impl PairPotential for Mie {
    fn tail_energy(&self, cutoff: f64) -> f64 {
        if self.m < 3.0 {
            return 0.0;
        };
        let sigma_rc = self.sigma / cutoff;
        let n_3 = self.n - 3.0;
        let m_3 = self.m - 3.0;
        let repulsive = f64::powf(sigma_rc, n_3);
        let attractive = f64::powf(sigma_rc, m_3);
        return -self.prefactor * self.sigma.powi(3) * (repulsive / n_3 -␣
↪attractive / m_3);
    }

    fn tail_virial(&self, cutoff: f64) -> f64 {
        if self.m < 3.0 {
            return 0.0;
        };
        let sigma_rc = self.sigma / cutoff;
        let n_3 = self.n - 3.0;
        let m_3 = self.m - 3.0;
        let repulsive = f64::powf(sigma_rc, n_3);
        let attractive = f64::powf(sigma_rc, m_3);
        return -self.prefactor * self.sigma.powi(3) * (repulsive * self.n / n_3 -␣
↪attractive * self.m / m_3);
    }
}
```

Note that we cannot correct every kind of energy function. In fact, the potential has to be a *short ranged* potential. For our Mie potential, both the exponents have to be larger than 3.0 else our potential will be *long ranged* and the integral that has to be solved to compute the tail corrections diverges. We return zero in that case.

Running a simulation

That concludes the first part. To test your new and shiny potential, you can run a small simulation. You'll find a minimal Monte Carlo simulation example in the `tutorials/potential` directory of the main lumol repository where you will also find the `src/lib.rs` file we created in this tutorial. You can then run the simulation via

```
cargo run --release
```

Fantastic! You implemented a new potential and ran a simulation with it!

If you want to share your implementation with other Lumol users only some small additional steps are neccessary. We will talk about them in the next part of this tutorial (which is not yet written).

6 Frequently Asked Questions

Here are some commons questions about Lumol. If you have more questions, please contact us on Gitter to ask it, so that we can add it here!

6.1 What kind of simulation can I run with Lumol?

You should be able to run any kind of classical simulation, the only limit being the number of atoms fitting in memory.

6.2 Why should I use Lumol?

If any of these statement is true for you, you should consider using Lumol:

- You need to use a specific potential that is not yet available in other codes, or develop your own potential. Adding a new potential in Lumol is very simple and take less than 20 lines of code;
- You are developing new simulation algorithms, for example more efficient free-energy computations or better parallel scaling of Coulomb computations. Lumol allow you to write the specific algorithm, and reuse all the other part of the simulation engine;

Other nice goodies include:

- Nicely formatted and easy to read input files;
- *(and more to come . . .)*

6.3 Why should I not use Lumol?

Here are some reasons for you not to use Lumol:

- You need to get the fastest code for your simulations because you are working with a lot of atoms. Lumol is relatively young and is not yet fully optimized;
- You need to run your simulation on a cluster. Lumol can run on multiple cores (think OpenMP), but not yet on multiple nodes (think MPI).

6.4 How can I build the initial configuration?

Lumol does not provide tools for building the initial simulation configuration. There are already a lot of very good tools around, that you can use. Examples include VMD, packmol, and many others. Because Lumol uses chemfiles to read initial configuration, any format supported by chemfiles can be used.

6.5 Is the code parallel?

Lumol can run in parallel on a single computer, using the multiple cores of the processor (this is shared memory parallelism, like OpenMP). It is not yet possible to run Lumol on multiple nodes in a cluster (message passing parallelism, like MPI).

6.6 Why is Lumol written in Rust?

Rust is a language created by Mozilla, and was released in 1.0 version in may 2015. It is a modern language, that provides the same access to the bare metal performances as C or C++, but prevents some programmer mistakes leading to crashes and corruptions.

This allow to build better software faster, because the programmer does not need to spend as much time debugging the code. At the same time, it also allow to check at compile-time that a code is data-race free, and allow to build parallel programs more easily.

6.7 Is there any graphical interface to Lumol?

Not yet. But because Lumol is built as a library implementing all the simulation algorithms, it should be relatively easy to create a graphical interface around it. If you are interested in a graphical user interface (using it or building it), please contact us!