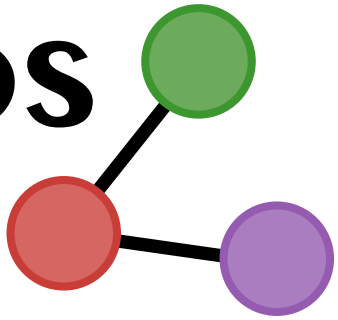
Jumos



Main Documentation

Release 0.1

Guillaume Fraux

May 26, 2015

Contents

Jumos is a package for molecular simulations using the Julia language. It aims at being as flexible as possible, and allowing the easy use and development of novel algorithm for each part of a simulation. Every algorithm, from potential computation, long range interactions, pair lists computing, outputs, *etc.* can be customised.

Jumos also includes code for trajectory analysis, either during the simulation run or by reading frames in a file.

> **Warning:** This package is in a very alpha stage, and still in heavy developement. Breaking changes can occurs in the API without any notice at any time.

This documentation is divided in two parts: first come the user manual, starting by some explanation about usal *algorithms in simulations* (page 3) and an *example* (page 2) of how we can use *Jumos* to run a molecular dynamic simulation. The second part is the developer documentation, exposing the internal of *Jumos*, and how we can use them to programm new algorithms.

1 User manual

1.1 Simulations

The simulation module contains code for molecular dynamic simulation. It extensively uses custom types, which are presented in this section.

The first section (*Usual simulation steps* (page 3)) describes how the code is organised, and is worth reading to understand the design of this module. Then *an example* (page 2) shows how to use *Jumos* to run simple simulations, and provides an example of the API usage. The *simulation* (page 5) part presents most of the API in a more formal maner.

The *Potentials* (page 6) section lists the available potentials for use in *Jumos*, and how you can easily use another potential.

Usage example

In *Jumos*, one run simulations by writting specials Julia scripts. A primer intoduction to the Julia language can be found here, if needed.

Lennard-Jones fluid

Here is a simple simulation script for running a simulation of a Lennard-Jones fluid at $300K$.

```
# Loading the Jumos module before anything else
using Jumos

# Molecular Dynamics with 1.0fs timestep
sim = MolecularDynamic(1.0)

# Create the simulation cell : cubic simulation cell with a width of 10A
set_cell(sim, (10.0,))

# Create the initial topology, positions and velocities
read_topology(sim, "lennard-jones.xyz")
read_positions(sim, "lennard-jones.xyz")
create_velocities(sim, 300)  # Initialize at 300K

# Add Lennard-Jones interactions between He atoms
add_interaction(sim, LennardJones(0.8, 2.0), "He")

out_trajectory = TrajectoryOutput("LJ-trajectory.xyz", 1)
add_output(sim, out_trajectory)
add_output(sim, EnergyOutput("LJ-energy.dat", 10))
```

```
21
22 run!(sim, 500)
23
24 # It is really easy to change some parameters if you bind them to variables
25 out_trajectory.frequency = 10
26
27 run!(sim, 5000)
28
29 # Simulation scripts are normal Julia scripts !
30 println("All done")
```

Each simulation script should start by the `using Jumos` directive. This imports the module and the exported names in the current scope.

Then, in this script, we create a molecular dynamics simulation with a timestep of 1.0 fs, and associated a cubic cell to this simulation. The topology and the original positions are read from the same file, as an `.xyz` file contains topological information (mainly the atomics names).

The only interaction — a *Lennard-Jones* (page 6) interaction — is also added to the simulation before the run. The next lines add some outputs to the simulation, namely a *trajectory* (page 10) and an *energy* (page 10) output. Finally, the simulation runs for a first 500 steps.

Any parameter can easily be changed during the simulation: here, at the line 25, we change the output frequency of the trajectory output, and then run the simulation for another 5000 steps.

Usual simulation steps

Note: All the capacities presented here may not be implemented (at least not yet) in *Jumos*. However, this page describes the exact steps involved in preparing and running a simulation with *Jumos*.

In every molecular dynamics simulation, the main steps are roughly the same. One shall start by setting the simulation : positions of the particles, velocities (either from a file or from a random gaussian distribution), timestep, atomic types, potential to use between the atoms, and so on.

Then we can run the simulation for a given number of steps `n_steps`. During the run, the three main steps are: computing the forces from the potentials; integrating the movement and outputing the quantities of interest : trajectories, energy, temperature, pressure, *etc.*

This is summarised in the figure *Usual steps in a molecular dynamic simulation* (page 4).

Setting the simulation

Here, one is building his simulation. The order of the steps doesn't matter: we should start by defining the simulation cell, and we have to define a topology before setting the initial positions, but we may setup the interactions at any time.

Get topology The topology of a simulation is a representation of the atoms, molecules, groups of molecules found in this simulation. It can be read from a file (`.pdb` files ,:file:.*lmp* LAMMPS topology, . . .) ; guessed from the initial configuration (two atoms are linked if they are closer than the sum of there Van der Waals radii) ; or built by hand.

Setup interaction The interactions describe how atoms should behave: they can be pair potentials, many-body potentials, bond potentials, torsion potentials, dihedral potentials or any other kinds of interactions.

Each of these potentials is associated with a force. This force is used in molecular dynamics to integrate Newton's equations of motions.

Initial positions The initial positions of all the atoms in the system can be read from a file, or defined by hand.

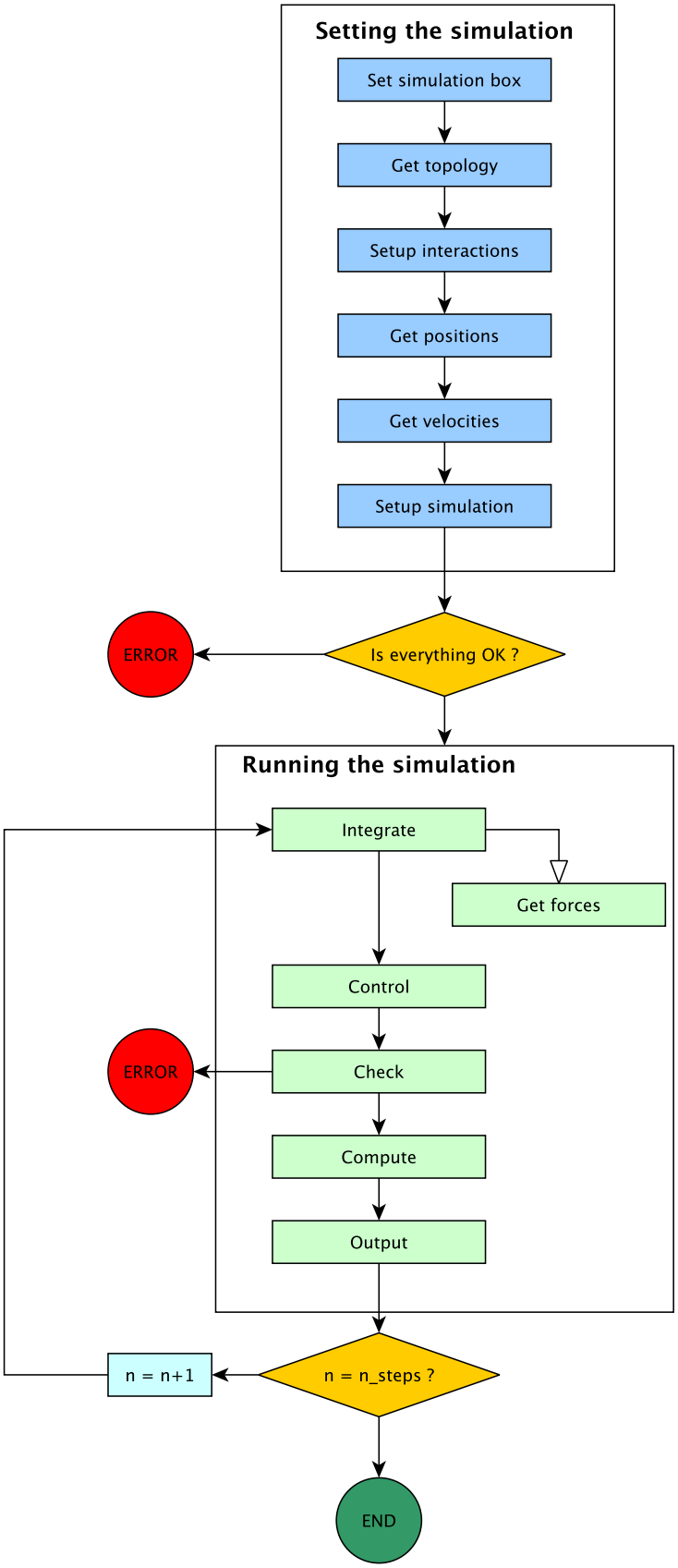


Fig. 1: Usual steps in a molecular dynamic simulation

A molecular dynamics simulation is usually built around two main parts, these parts being composed of various steps. The first part sets the configuration, defining all the information needed for the simulation. The second part is the dynamics calculation, effectivelly running the simulation.

Initial velocities We will also need the initial velocities to start the time integration in molecular dynamics. They can be defined either in a file (to restart previous dynamic), or randomly assigned. In the later case, it is better to use a Maxwell-Boltzmann distribution at the temperature T of the simulation.

Setup simulation We will have to set other values in order to run a simulation, depending on the kind of simulation: the timestep of integration dt, the type of thermostat or barostat to use.

Running the simulation

The order of the steps in a simulation run is fixed, and can not be changed.

Integrate Using the forces, one can now integrate Newton's equations of motions. Numerous algorithms exist, such as velocity-Verlet, Beeman algorithm, multi-timestep RESPA algorithm. At the end, postions and velocities are updated to the new step.

Get forces During the integration steps, we will need forces acting on the atoms. This forces computing step is the most time consuming aspect of the calculation. Various ways to do this computation exists, depending on the type of potentials (short range or long range, bonding or not bonding), and some tricks can speed up this computation (pairs list, short range potential truncation).

The algorithm used for integration have to call this step when and as many times as needed.

Control If the time integration does not force the value of external parameters, like temperature or pressure or volume, this step can enforce them. Controls examples are the Berendsen thermostat, velocity rescaling, particle wrapping to the cell.

Check During the run, one can check for simulation consistency. For example, the number of particles in the cell may have be constant, the global momentum should be null, and so on.

Compute This step allows for computing and averaging physical quantities, such as total energy, kinetic energy, temperature, pressure, magnetic moment, and any other physical quantities.

Output In order to exploit the computation results, we have to do save them to a file. during this step, we can output computed values and the trajectory.

Creating simulations

The Simulation type

In *Jumos*, simulations are first-class citizen, *i.e.* objects bound to variables. The main type for simulations is `MolecularDynamic`, which can be constructed in two ways:

`MolecularDynamic`(*timestep*)
: Creates an empty molecular dynamic simulation using a Velocity-Verlet integrator with the specified timestep.

  Without any *thermostat* (page 13) or *barostat* (page 14), this performs a NVE integration of the system.

`MolecularDynamic`(*::Integrator*)
: Creates an empty simulation with the specified *integrator* (page 13).

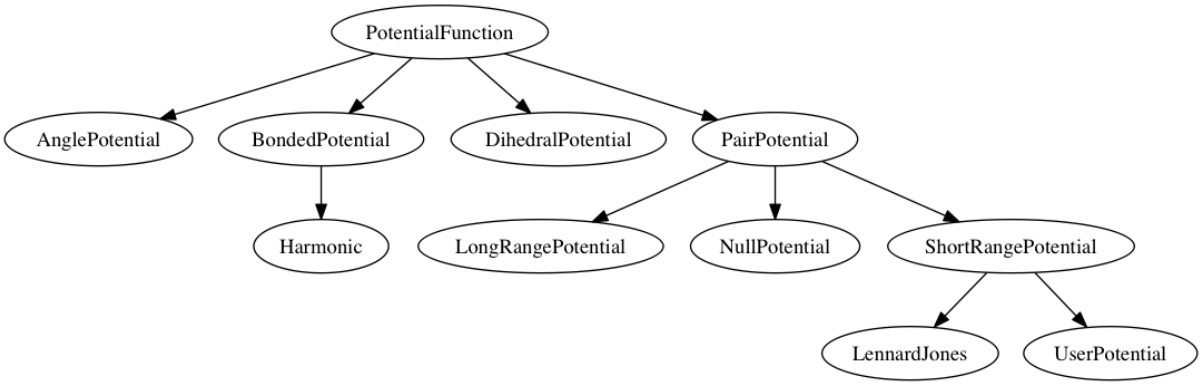
Default algorithms

Default algorithms for molecular dynamic are presented in the following table:

Simulation step	Default algorithms
Integration	*Velocity-Verlet* (page 13)
Forces computation	*Naive computation* (page 23)
Control	*Wrap particles in the box* (page 14)
Check	*All positions are defined* (page 13)
Compute	*None*
Output	*None*

Potentials

In order to compute the energy or the forces acting on a particle, two informations are needed : a potential energy function, and a computation algorithm. The potential function is a description of the variations of the potential with the particles positions, and a potential computation is a way to compute the values of this potential function. The following image shows the all the potentials functions currently implemented in *Jumos*.



We can see these potentials are classified as four main categories: pair potentials, bond potentials, angles potentials (between 3 atoms) and dihedral potentials (between 4 atoms).

The only implemented pair potentials are short-range potentials. Short-range pair potentials go to zero faster than the $1/r^3$ function, and long-range pair potentials go to zero at the same speed or more slowly than $1/r^3$. A typical example of long-range pair potential is the Coulomb potential between charged particles.

Potential functions

Short-range pair potential Short-range pair potential are subtypes of *PairPotential*. They only depends on the distance between the two particles they are acting on. They should have two main properties:

- They should go to zero when the distance goes to infinity;
- They should go to zero faster than the $1/r^3$ function.

Lennard-Jones potential A Lennard-Jones potential is defined by the following expression:

$$V(r) = 4\epsilon \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right)$$

`LennardJones` (*epsilon*, *sigma*)

Creates a Lennard-Jones potential with $\sigma =$ sigma, and $\epsilon =$ epsilon. `sigma` should be in angstroms, and `epsilon` in kJ/mol.

Typical values for Argon are: $\sigma = 3.35\ A, \epsilon = 0.96\ kJ/mol$

Null potential This potential is a potential equal to zero everywhere. It can be used to define "interactions" between non interacting particles.

`NullPotential``()`
Creates a null potential instance.

Bonded potential Bonded potentials acts between two particles, but does no go to zero with an infinite distance. *A contrario*, they goes to infinity as the two particles goes apart of the equilibirum distance.

Harmonic An harmonic potential have the following expression:

$$V(r) = \frac{1}{2}k(\vec{r} - \vec{r}_0)^2 - D_0$$

D_0 is the depth of the potential well.

`Harmonic`(*k*, *r0*, *depth=0.0*)
Creates an harmonic potential with a spring constant of *k* (in $kJ.mol^{-1}.A^{-2}$), an equilibrium distance r_0 (in angstroms); and a well's depth of D_0 (in kJ/mol).

Adding interactions to a simulation

Before runnning a simulation, we should define interactions between all the pairs of atomic types. The `add_interaction` function should be used for that.

`add_interaction`(*sim*, *potential*, *atoms*[, *computation=:auto*, *kwargs...*])
Adds an interaction between the `atoms` in the simulation `sim`. The energy and the forces for this interaction will be computed using the potential function `potential`.

`atoms` can be a string, an integer, or a tuple of strings and integers. The strings should be the atomic names, and the integers the atomic type in the simulation's topology. For example, if a simulation has the three following atomic types: `["C", "H", "O"]` with respective index 1, 2 and 3; then the atomic pair `("H", "O")` can be accessed with either `(2, 3)`, `(2, "O")`, `("H", 3)` or `("H", "O")`.

The keyword argument `computation` determines which computation is used for the given potential function. The default is to use CutoffComputation with short-range pair potentials, and DirectComputation with the other ones.

All the other keyword arguments will be used to create the potential computation algorithm.

Defining a new potential

User potential The easier way to define a new potential is to create `UserPotential` instances, providing potential and force functions. To add a potential, for example an harmonic potential, we have to define two functions, a potential function and a force function. These functions should take a `Float64` value (the distance) and return a `Float64` (the value of the potential or the force at this distance).

`UserPotential`(*potential*, *force*)
Creates an UserPotential instance, using the `potential` and `force` functions.

`potential` and `force` should take a `Float64` parameter and return a `Float64` value.

`UserPotential`(*potential*)
Creates an UserPotential instance by automatically computing the force function using a finite difference method, as provided by the Calculus package.

Here is an example of the user potential usage:

```
# potential function
f(x) = 6*(x-3.)^2 - .5
# force function
g(x) = -12.*x + 36.

# Create a potential instance
my_harmonic_potential = UserPotential(f, g)

# One can also create a potential whithout providing a funtion for the force,
# at the cost of a less effective computation.
my_harmonic_2 = UserPotential(f)

force(my_harmonic_2, 3.3) == force(my_harmonic_potential, 3.3)
# false

isapprox(force(my_harmonic_2, 3.3), force(my_harmonic_potential, 3.3))
# true
```

Subtyping PotentialFunction A more efficient way to use custom potential is to subtype the either `PairPotential`, `BondedPotential`, `AnglePotential` or `DihedralPotential`, according to the new potential from.

For example, we are goning to define a Lennard-Jones potential using an other function:

$$V(r) = \frac{A}{r^{12}} - \frac{B}{r^6}$$

This is obviously a `PairPotential`, so we are going to subtype this potential function.

To define a new potential, two functions are needed: *call* and *force*. It is necessary to import these two functions in the current scope before extending them. Potentials should be declared as `immutable`, to allow some optimisations in the computations.

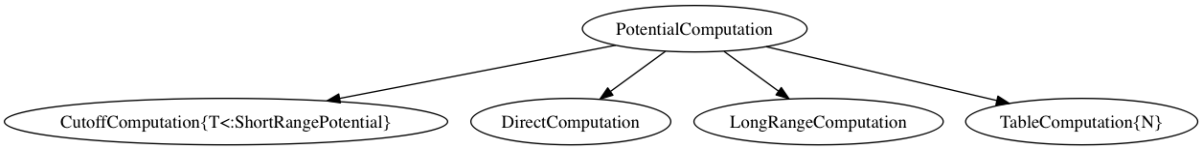
```
# import the functions to extend
import Base: call
import Jumos: force

immutable LennardJones2 <: PairPotential
    A::Float64
    B::Float64
end

# potential function
function call(pot::LennardJones2, r::Real)
    return pot.A/(r^12) - pot.B/(r^6)
end

# force function
function force(pot::LennardJones2, r::Real)
    return 12 * pot.A/(r^13) - 6 * pot.B/(r^7)
end
```

The above example can the be used like this:

```
sim = MolecularDynamic(1.0)

add_interaction(sim, LennardJones2(4.5, 5.3), ("He", "He"))

pot = LennardJones2(4.5, 5.3)

pot(3.3) # value of the potential at r=3.3
force(pot, 3.3) # value of the force at r=3.3
```

Potential computation

As stated at the begiging of this potentials section, we need two informations to compute interactions between particles: a potential function, and a potential computation. The potential compuatation algorithms come in four flavors:



- The `DirectComputation` is only a small wrapper on the top of the potential functions, and directly calls the potential function methods for energy and force evaluations.
- The `CutoffComputation` is used for short range potentials. All interactions at a longer distance than the cutoff distance are set to zero. The default cutoff is 12 A, and this can be changed by passing a `cutoff` keyword argument to the `add_interaction` function. With this computation, the energy is shifted so that their is a continuity in the energy at the cutoff distance.
- The `TableComputation` use table lookup to extrapolate the potential energy and the forces at a given point. This saves computation time at the cost of accuracy. This algorithm is parametrized by an integer, the size of the undelying array. Increases in this size will result in more accuracy, at the cost of more memory usage. The default size is 2000 points — which corresponds to roughly 15kb. `TableComputation` has also a maximum distance for computations, `rmax`. For any bigger distances, the `TableComputation` will returns a null energy and null forces. So `TableComputation` can only be used if you are sure that the particles will never be at a greater distance than `rmax`.
- The `LongRangeComputation` is not implemented yet.

Which computation for which potential ? Not all computation algorithms are suitable for all potential functions. The usable associations are in the table below.

Function	DirectComputation	CutoffComputation	TableComputation
ShortRangePotential	✔	✔	✔
BondedPotential	✔	✘	✔
AnglePotential	✔	✘	✔
DihedralPotential	✔	✘	✔

Using non-default computation By default, the computation algorithm is automatically determined by the potential function type. `ShortRangePotential` are computed with `CutoffComputation`, and all other potentials are computed by `DirectComputation`. If we want to use another computation algorithm, this can be done by providing a `computation` keyword to the `add_interaction` function. The following values are allowed:

- `:direct` to use a `DirectComputation`;
- `:cutoff` to use a `CutoffComputation`. The cutoff can be specified with the `cutoff` keyword argument;
- `:table` ` to use a `TableComputation`. The table size can be specified with the `numpoints` keyword argument, and the maximum distance with the `rmax` keyword argument.

Here is an example of how we can use these keywords.

```
sim = MolecularDynamic(1.0) # Creating a simulation

# ...

# Use default computation, i.e. CutoffComputation with 12A cutoff.
add_interaction(sim, LennardJones(0.4, 3.3))
```

```
# Use another cutoff distance
add_interaction(sim, LennardJones(0.4, 3.3), cutoff=7.5)

# Use direct computation
add_interaction(sim, LennardJones(0.4, 3.3), computation=:direct)

# Use table computation with 3000 points, and a maximum distance of 10A
add_interaction(sim, LennardJones(0.4, 3.3),
                computation=:table, numpoints=3000, rmax=10.0)
```

Exporting values of interest

While running a simulation, some basic analysis can be performed and writen to a data file. Further analysis can be differed by writing the trajectory to a file, and running existing tools on these trajectories.

In *Jumos*, outputs are subtypes of the `BaseOutput` type, and can be added to a simulation by using the *`add_output()`* (page 15) function.

`add_output` (*sim*, *output*)
Add an output to the simulation. If the output is already present, a warning is issued. Usage example:

```
sim = MolecularDynamic(1.0)

# Direct addition
add_output(sim, TrajectoryOutput("mytraj.xyz"))

# Binding to a variable
out = TrajectoryOutput("mytraj-2.xyz", 5)
add_output(sim, out)
```

Each output is by default writen to its file at every simulation step. To speed-up the simulation run and remove useless information, a *write frequency* can be used as the last parameter of each output constructor. If this frequency is set to `n`, the values will be writen only every *n* simulaton steps. This frequency can also be changed dynamically:

```
# Frequency is set to 1 by default
traj_out = TrajectoryOutput("mytraj.xyz")

add_output(sim, traj_out)
run!(sim, 300) # 300 steps will be writen

# Set frequency to 50
traj_out.frequency = 50
run!(sim, 500) # 10 steps will be writen
```

Existing outputs

Trajectory output The first think one might want to save in a simulation run is the trajectory of the system. Such trajectory can be used for visualisation, storage and further analysis. The `TrajectoryOutput` provide a way to write this trajectory to a file.

`TrajectoryOutput` (*filename*[, *frequency*])
This construct a `TrajectoryOutput` which can be used to write a trajectory to a file. The trajectory format is guessed from the `filename` extension. This format must have write capacities, see *the list* (page 17) of supported formats in *Jumos*.

Energy output The energy output write to a file the values of energy and temperature for the current step of the simulation. Values writen are the current step, the kinetic energy, the potential energy, the total energy and the temperature.

`EnergyOutput` (*filename*[, *frequency*])
This construct a `EnergyOutput` which can be used to the energy evolution to a file.

Defining a new output

Adding a new output with custom values, can be done either by using a custom output or by *subtyping* (page 24) the `BaseOutput` type to define a new output. The the former wayis to be prefered when adding a *one-shot* output, and the latter when adding an output which will be re-used.

Custom output The `CustomOutput` type provide a way to build specific output. The data to be writen should be *computed* (page 11) before the output by adding the specific algorithms to the current simulation. These computation algorithm set a value in the `MolecularDynamic.data` dictionairy, which can be accessed during the output step. See the *computation algorithms* (page 11) page for a list of keys.

`CustomOutput` (*filename*, *values*[, *frequency; header="# header string"*])
This create a `CustomOutput` to be writen to the file `filename`. The `values` is a vector of symbols, these symbols being the keys of the `MolecularDynamic.data` dictionairy. The `header` string will be writen on the top of the output file.

Usage example:

```
sim = MolecularDynamic(1.0)

# TemperatureCompute register a :temperature key
add_compute(sim, TemperatureCompute())

temperature_output = CustomOutput("Sim-Temp.dat", [:temperature])
add_output(sim, temperature_output)
```

Computing values of interest

To compute physical values from a simulation, we can use algorithms represented by subtypes of `Basecompute` and associate these algorithms to a simulation.

Users don't usualy need to use these compute algorithms directly, as the output algorithms (see *Exporting values of interest* (page 10)) set the needed computations by themself.

Computed values can have various usages: they may be used in *outputs* (page 10), or in *controls* (page 13). The data is shared between algorithms using the `MolecularDynamic.data` field. This field is a dictionnary associating symbols and any kind of value.

This page of documentation presents the implemented computations. Each computation can be associated with a specific *simulation* (page 5) using the `add_compute` function.

`add_compute` (*::MolecularDynamic*, *::BaseCompute*)
This function registers a computation for a given simulation. Example usage:

```
sim = MolecularDynamic() # Create a simulation
# ...

# Do not forget the parentheses to instanciate the computation
add_compute(sim, MyCompute())

run!(sim, 10)

# You can access the last computed value in the sim.data dictionnary
sim.data[:my_compute]
```

You can also call directly any instance of `MyCompute`:

```
sim = MolecularDynamic() # Create a simulation
# ...

compute = MyCompute() # Instanciate the compute
value = compute(sim) # Compute the value
```

The following paragraphs sums up the implemented computations, giving for each algorithm the return value (for direct calling), and the associated keys in `MolecularDynamic.data`.

Energy related values

type `TemperatureCompute`
Computes the temperature of the simulation. All the masses have to be set.

Key: `:temperature`

Return value: The current frame temperature.

type `EnergyCompute`
Computes the potential, kinetic and total energy of the current simulation step.

Keys: `:E_kinetic`, `:E_potential`, `:E_total`

Return value: A tuple containing the kinetic, potential and total energy.

```
energy = EnergyCompute()
sim = MolecularDynamic()

# unpacking the tuple
E_kinetic, E_potential, E_total = energy(sim)

# accessing the tuple values
E = energy(sim)

E_kinetic = E[1]
E_potential = E[2]
E_total = E[3]
```

Volume

type `VolumeCompute`
Computes the volume of the current *unit cell* (page 19).

Key: `:volume`

Return value: The current cell volume

Pressure

type `PressureCompute`
TODO

Key:

Return value:

Selecting the algorithms

Integrator: running the simulation

An integrator is an algorithm responsible for updating the positions and the velocities of the current *frame* (page 20) of the *simulation* (page 5).

Verlet integrators Verlet integrators are based on Taylor expensions of Newton's second law. They provide a simple way to integrate the movement, and conserve the energy if a sufficently small timestep is used. Assuming the absence of barostat and thermostat, they provide a NVE integration.

type `Verlet`
: The Verlet algorithm is described here for example. The main constructor for this integrator is `Verlet(timestep)`, where `timestep` is the timestep in femtosecond.

type `VelocityVerlet`
: The velocity-Verlet algorithm is descibed extensively in the literature, for example in this webpages. The main constructor for this integrator is `VelocityVerlet(timestep)`, where `timestep` is the integration timestep in femtosecond. This is the default integration algorithm in *Jumos*.

Checking the simulation consistency

Molecular dynamic is usually a *garbage in, garbage out* set of algorithms. The numeric and physical issues are not caught by the algorithm themselves, and the physical (and chemical) consistency of the simulation should be checked often.

In *Jumos*, this is achieved by the `Check` algorithms, which are presented in this section. Checking algorithms can be added to a simulation by using the *`add_check()`* (page 15) function.

Existing checks

type `GlobalVelocityIsNull`
: This algorithm checks if the global velocity (the total moment of inertia) is null for the current simulation. The absolute tolerance is $10^{-5}\ A/fs$.

type `TotalForceIsNull`
: This algorithm checks if the sum of the forces is null for the current simulation. The absolute tolerance is $10^{-5}\ uma \cdot A/fs^2$.

type `AllPositionsAreDefined`
: This algorithm checks is all the positions and all the velocities are defined numbers, *i.e.* if all the values are not infinity or the `NaN` (not a number) values.

  This algorithm is used by default by all the molecular dynamic simulation.

Controlling the simulation

While running a simulation, we often want to have control over some simulation parameters: the temperature, the pressure, ... This is the goal of the *Control* algorithms.

Such algorithms are subtypes of `BaseControl`, and can be added to a simulation using the *`add_control()`* (page 15) function:

Controlling the temperature: Thermostats Various algorithms are available to control the temperature of a simulation and perform pseudo NVT simulations. The following thermostating algorithms are currently implemented:

type `VelocityRescaleThermostat`

The velocity rescale algorithm controls the temperature by rescaling all the velocities when the temperature differs exceedingly from the desired temperature.

The constructor takes two parameters: the desired temperature and a tolerance interval. If the absolute difference between the current temperature and the desired temperature is larger than the tolerance, this algorithm rescales the velocities.

```
sim = MolecularDynamic(2.0)

# This sets the temperature to 300K, with a tolerance of 50K
thermostat = VelocityRescaleThermostat(300, 50)

add_control(sim, thermostat)
```

type `BerendsenThermostat`

The berendsen thermostat sets the simulation temperature by exponentially relaxing to a desired temperature. A more complete description of this algorithm can be found in the original article [1].

The constructor takes as parameters the desired temperature, and the coupling parameter, expressed in simulation timestep units. A coupling parameter of 100, will give a coupling time of 150 fs if the simulation timestep is 1.5 fs, and a coupling time of 200 fs if the timestep is 2.0 fs.

`BerendsenThermostat` (*T*[, *coupling*])

Creates a Berendsen thermostat at the temperature `T` with a coupling parameter of `coupling`. The default values for `coupling` is 100.

```
sim = MolecularDynamic(2.0)

# This sets the temperature to 300K
thermostat = BerendsenThermostat(300)

add_control(sim, thermostat)
```

Controlling the pressure: Barostats

type `BerendsenBarostat`

TODO

Other controls

type `WrapParticles`

This control wraps the positions of all the particles inside the *unit cell* (page 19).

This control is present by default in the molecular dynamic simulations.

Functions for algorithms selection

The six following functions are used to to select specific algorithms for the simulation. They allow to add and change all the algorithms, even in the middle of a run.

`set_integrator` (*sim*, *integrator*)

Sets the simulation integrator to `integrator`.

Usage example:

```
# Creates the integrator directly in the function
set_integrator(sim, Verlet(2.5))

# Binds the integrator to a variable if you want to change a parameter
integrator = Verlet(0.5)
set_integrator(sim, integrator)
```

[1] H.J.C. Berendsen, *et al.* J. Chem Phys **81**, 3684 (1984); doi: 10.1063/1.448118

```
run!(sim, 300)   # Run with a 0.5 fs timestep
integrator.timestep = 1.5
run!(sim, 3000)  # Run with a 1.5 fs timestep
```

set_forces_computation (*sim*, *forces_computer*)
Sets the simulation algorithm for forces computation to `forces_computer`.

add_check (*sim*, *check*)
Adds a *check* (page 13) to the simulation check list and issues a warning if the check is already present.

Usage example:

```
# Note the parentheses, needed to instanciate the new check.
add_check(sim, AllPositionsAreDefined())
```

add_control (*sim*, *control*)
Adds a *control* (page 13) algorithm to the simulation list. If the control algorithm is already present, a warning is issued.

Usage example:

```
add_control(sim, RescaleVelocities(300, 100))
```

add_compute (*sim*, *compute*)
Adds a *compute* (page 11) algorithm to the simulation list. If the algorithm is already present, a warning is issued.

Usage example:

```
# Note the parentheses, needed to instanciate the new compute algorithm.
add_compute(sim, EnergyCompute())
```

add_output (*sim*, *output*)
Adds an *output* (page 10) algorithm to the simulation list. If the algorithm is already present, a warning is issued.

Usage example:

```
add_output(sim, TrajectoryOutput("mytraj.xyz"))
```

1.2 Trajectories

When running molecular simulations, the trajectory of the system is commonly saved to the disk in prevision of future analysis or new simulations run. The *Trajectories* module offers facilities to read and write this files.

Reading and writing trajectories

One can read or write *frames* (page 20) from a trajectory. In order to do so, more information is needed : namely an *unit cell* (page 19) and a *topology* (page 18). Both are optional, but allow for better computations. Some file formats already contain this kind of informations so there is no need to provide it.

Trajectories can exist in differents formats: text formats like the XYZ format, or binary formats. In *Jumos*, the format of a trajectory file is automatically determined based on the file extension.

Base types

The two basic types for reading and writing trajectories in files are respectively the `Reader` and the `Writer` parametrised types. For each specific format, there is a `FormatWriter` and/or `FormatReader` subtype implementing the basic operations.

Usage

The following functions are defined for the interactions with trajectory files.

`opentraj` (*filename*[, *mode="r"*, *topology=""*, *kwargs...*])
: Opens a trajectory file for reading or writing. The `filename` extension determines the *format* (page 17) of the trajectory.

  The `mode` argument can be `"r"` for reading or `"w"` for writing.

  The `topology` argument can be the path to a *Topology* (page 18) file, if you want to use atomic names with trajectories files in which there is no topological informations.

  All the keyword arguments `kwargs` are passed to the specific constructors.

`Reader` (*filename*[, *kwargs...*])
: Creates a `Reader` object, by passing the keywords arguments `kwargs` to the specific constructor. This is equivalent to use the *opentraj* function with `"r"` mode.

`Writer` (*filename*[, *kwargs...*])
: Creates a `Writer` object, by passing the keywords arguments `kwargs` to the specific constructor. This is equivalent to use the *opentraj* function with `"w"` mode.

`eachframe` (*::Reader [range::Range, start=first_step]*)
: This function creates an [interator] interface to a `Reader`, allowing for constructions like `for frame in eachframe(reader)`.

`read_next_frame!` (*::Reader*, *frame*)
: Reads the next frame from `Reader`, and stores it into `frame`. Raises an error in case of failure, and returns `true` if there are other frames to read, `false` otherwise.

  This function can be used in constructions such as `while read_next_frame!(traj)`.

`read_frame!` (*::Reader*, *step*, *frame*)
: Reads a frame at the step `step` from the `Reader`, and stores it into `frame`. Raises an error in the case of failure and returns `true` if there is a frame after the step `step`, `false` otherwise.

`write` (*::Writer*, *frame*)
: Writes the *Frame* (page 20) `frame` to the file associated with the `Writer`.

`close` (*trajectory_file*)
: Closes the file associated with a `Reader` or a `Writer`.

Reading frames from a file Here is an example of how you can read frames from a file. In the `Reader` constructor, the `cell` keyword argument will be used to construct an *UnitCell* (page 19).

```
traj_reader = Reader("filename.xyz", cell=[10., 10., 10.])

for frame in eachframe(traj_reader)
    # Do stuff here
end

close(traj_reader)
```

Writing frames in a file Here is an example of how you can write frames to a file. This example converts a trajectory from a file format to another. The `topology` keyword is used to read a *Topology* (page 18) from a file.

```
traj_reader = Reader("filename-in.nc", topology="topology.xyz")
traj_writer = Writer("filename-out.xyz")

for frame in eachframe(traj_reader)
    write(traj_writer, frame)
end
```

```
close(traj_writer)
close(traj_reader)
```

Supported formats

The following table summarizes the formats supported by *Jumos*, giving the reading and writing capacities of *Jumos*, as well as the presence or absence of the unit cell and the topology information in the files. The last column indicates the accepted keywords.

Format	Extension	Read	Write	Cell	Topology	Keywords
XYZ	`.xyz`	✔	✔	✘	✔	cell
Amber NetCDF	`.nc`	✔	✘	✔	✘	topology

Readind and writing topologies

Topologies can also be represented and stored in files. Some functions allow to read directly these files, but there is usally no need to use them directely.

Supported formats for topology

Topology reading supports the formats in the following table.

Format	Reading ?	Writing ?
XYZ	✔	✔
LAMMPS data file	✔	✘

If you want to write a toplogy to a file, the best way for now is to create a frame with this topology, and write this frame to an XYZ file.

1.3 Analysis

> **Warning:** This part of Jumos is not documented because it's going to be merged with the compute algorithms.

Base Histogram type

Radial distribution function

Density profile

1.4 Internal units

Jumos uses a set of internal units, converting back and forth to these units when needed. Conversion from SI units is always supported. Parenthesis indicate planned conversion that is not implemented yet.

Quantity	Internal unit	Supported conversions
Distances	Ångtröm (A)	(bohr)
Time	Femptosecond (fs)	
Velocities	Ångtröm/Femptosecond (A/fs)	
Mass	Unified atomic mass (u or Da)	g/mol
Temperature	Kelvin (K)	
Energy	Kilo-Joule/Mole (kJ/mol)	(eV), (Ry), ($kcal/mol$)
Force	Kilo-Joule/(Mole-Ångtröm) $kJ/(molA)$	
Pressure	bar	(atm)
Charge	Multiples of $e = 1.602176487\ 10^{-19}C$	

2 Developer documentation

2.1 Developer documentation

Atoms

This module creates the link between human-readable and computer-readable information about the studied system. Humans prefer to use string labels for molecules and atoms, whereas a computer only uses integers.

Atom type

An `Atom` instance is a representation of atomic information. Think of an `Atom` as a cell in an augmented periodic table. You can access the following fields :

- `name` : the atom name;
- `symbol` : the atom chemical type;
- `mass` : the atom mass;
- `special::Dict{String, Any}` : all the other values: charge, dipolar moment ...;

The atom name and symbol may not be the same: if there are two kinds of hydrogen atoms in a simulation, they may have the names **H1** and **H2** ; and share the same symbol **H**.

Topology

A `Topology` instance stores all the information about the system : atomic types, atomic composition of the system, bonds, angles, dihedral agnles and molecules.

Atoms of the system can be accesed using integer indexing. The following example shows a few operations available on atoms:

```
# topology is a Topology with 10 atoms

atom = topology[3]   # Get a specific atom
println(atom.name)   # Get the atom name

atom.name = "H2"     # Set the atom name
topology[5] = atom   # Set the 5th atom of the topology
```

Topology functions

`size` (*::Topology*)

This function returns the number of atoms in the topology.

atomic_masses (*::Topology*)

This function returns a `Vector{Float64}` containing the masses of all the atoms in the system. If no mass was provided, it uses the `ATOMIC_MASSES` dictionnary to guess the values. If no value is found, the mass is set to 0.0. All the values are in *internal units* (page 17).

Periodic table information

The `Atoms` module also defines two dictonaries that store information about atoms:

- `ATOMIC_MASSES` is a `Dict{String, Float64}` associating atoms symbols and atomic masses, in *internal units* (page 17) ;
- `VDW_RADIUS` is a `Dict{String, Integer}` associating atoms symbols and Van der Waals radii, in *internal units* (page 17).

Universe

The Universe module defines base data types used in the other modules.

Array3D

3-dimensionals vectors are very common in molecular simulations. The `Array3D` type implements arrays of this kind of vectors, provinding all the usual operations between its components.

If `A` is an `Array3D` and `i` an integer, `A[i]` is a 3-dimensional vector implementing `+, -` between vector, `.+, .-, .*, */` between vectors and scalars; `dot` and `cross` products, and the `unit!` function, normalizing its argument.

Simulation Cell

A simulation cell (`UnitCell` type) is the virtual container in which all the particles of a simulation move. There are three different types of simulation cells :

- Infinite cells (`InfiniteCell`) do not have any boundaries. Any move is allowed inside these cells;
- Orthorombic cells (`OrthorombicCell`) have up to three independent lenghts; all the angles of the cell are set to 90° ($\pi/2$ radians)
- Triclinic cells (`TriclinicCell`) have 6 independent parameters: 3 lenghts and 3 angles.

Creating simulation cell

UnitCell (*Lx*[, *Ly*, *Lz*, *alpha*, *beta*, *gamma*, *celltype*])

Creates an unit cell. If no `celltype` parameter is given, this function tries to guess the cell type using the following behavior: if all the angles are equals to $\pi/2$, then the cell is an `OrthorombicCell`; else, it is a `TriclinicCell`.

If no value is given for `alpha, beta, gamma`, they are set to $\pi/2$. If no value is given for `Ly, Lz`, they are set to be equal to `Lx`. This creates a cubic cell. If no value is given for `Lx`, a cell with lenghts of $0A$ and $\pi/2$ angles is constructed.

```
julia> UnitCell() # Without parameters
OrthorombicCell
    Lenghts: 0.0, 0.0, 0.0

julia> UnitCell(10.) # With one lenght
OrthorombicCell
    Lenghts: 10.0, 10.0, 10.0
```

```
julia> UnitCell(10., 12, 15) # With three lenghts
OrthorombicCell
    Lenghts: 10.0, 12.0, 15.0

julia> UnitCell(10, 10, 10, pi/2, pi/3, pi/5) # With lenghts and angles
TriclinicCell
    Lenghts: 10.0, 10.0, 10.0
    Angles: 1.5707963267948966, 1.0471975511965976, 0.6283185307179586

julia> UnitCell(InfiniteCell) # With type
InfiniteCell

julia> UnitCell(10., 12, 15, TriclinicCell) # with lenghts and type
TriclinicCell
    Lenghts: 10.0, 12.0, 15.0
    Angles: 1.5707963267948966, 1.5707963267948966, 1.5707963267948966
```

`UnitCell` (*u::Vector*[, *v::Vector*, *celltype*])

If the size matches, this function expands the vectors and returns the corresponding cell.

```
julia> u = [10, 20, 30]
3-element Array{Int64,1}:
 10
 20
 30

julia> UnitCell(u)
OrthorombicCell
    Lenghts: 10.0, 20.0, 30.0
```

Indexing simulation cell You can access the cell size and angles either directly, or by integer indexing.

`getindex` (*b::UnitCell*, *i::Int*)

Calling `b[i]` will return the corresponding length or angle : for `i in [1:3]`, you get the i^{th} lenght, and for `i in [4:6]`, you get [avoid get] the angles.

In case of intense use of such indexing, direct field access should be more efficient. The internal fields of a cell are : the three lenghts `x, y, z`, and the three angles `alpha, beta, gamma`.

Boundary conditions and cells Only fully periodic boundary conditions are implemented for now. This means that if a particle crosses the boundary at some step, it will be wrapped up and will appear at the opposite boundary.

Distances and cells The distance between two particles depends on the cell type. In all cases, the minimal image convention is used: the distance between two particles is the minimal distance between all the images of theses particles. This is explicited in the *Periodic boundary conditions and distances computations* (page 21) part of this documentation.

Frame

A `Frame` object holds the data from one step of a simulation. It is defined as

```
type Frame
    step::Integer
    cell::UnitCell
    topology::Topology
    positions::Array3D
    velocities::Array3D
end
```

i.e. it contains information about the current step, the current *cell* (page 19) shape, the current *topology* (page 18), the current positions, and possibly the current velocities. If there is no velocity information, the velocities `Array3D` is a 0-sized array.

Creating frames There are two ways to create frames: either explicitly or implicity. Explicit creation uses one of the constructors below. Implicit creation occurs while reading frames from a stored trajectory or from running a simulation.

The Frame type have the following constructors:

`Frame`(*::Topology*)
: Creates a frame given a topology. The arrays are pre-allocated to store data according to the topology.

`Frame``()`
: Creates an empty frame, with a 0-atoms topology.

Reading and writing frames from files The main goal of the `Trajectories` module is to read or write frames from or to files. See this module *documentation* (page 15) for more information.

Periodic boundary conditions and distances computations

The `PBC` module offers utilities for distance computations using periodic boundary conditions.

Minimal images

These functions take a vector and wrap it inside a *cell* (page 19), by finding the minimal vector fitting inside the cell.

`minimal_image`(*vect*, *cell::UnitCell*)
: Wraps the `vect` vector inside the `cell` unit cell. *vect* can be a `Vector` (*i.e.* a 1D array), or a view from an *Array3D* (page 19). This function returns the wrapped vector.

`minimal_image!`(*vect*, *cell::UnitCell*)
: Wraps `vect` inside of `cell`, and stores the result in `vect`.

`minimal_image!`(*A::Matrix*, *cell::UnitCell*)
: If `A` is a 3xN Array, wraps each one of the columns in the `cell`. The result is stored in `A`. If `A` is not a 3xN array, this throws an error.

Distances

Distances are computed using periodic boundary conditions, by wrapping the $\vec{r}_i - \vec{r}_j$ vector in the cell before computing its norm.

Within one Frame This set of functions computes distances within one frame.

`distance`(*ref::Frame*, *i::Integer*, *j::Integer*)
: Computes the distance between particles `i` and `j` in the frame.

`distance_array`(*ref::Frame*[, *result*])
: Computes all the distances between particles in `frame`. The `result` array can be passed as a pre-allocated storage for the $N \times N$ distances matrix. `result[i, j]` will be the distance between the particle i and the particle j.

`distance3d`(*ref::Frame*, *i::Integer*, *j::Integer*)
: Computes the $\vec{r}_i - \vec{r}_j$ vector and wraps it in the cell. This function returns a 3D vector.

Between two Frames This set of functions computes distances within two frames, either computing the how much a single particle moved between two frames or the distance between the position of a particle i in a reference frame and a particle j in a specific configuration frame.

`distance`(*ref::Frame*, *conf::Frame*, *i::Integer*, *j::Integer*)
Computes the distance between the position of the particle i in `ref`, and the position of the particle j in `conf`

`distance`(*ref::Frame*, *conf::Frame*, *i::Integer*)
Computes the distance between the position of the same particle i in `ref` and `conf`.

`distance3d`(*ref::Frame*, *conf::Frame*, *i::Integer*)
Wraps the `ref[i] - conf[i]` vector in the `ref` unit cell.

`distance3d`(*ref::Frame*, *conf::Frame*, *i::Integer*, *j::Integer*)
Wraps the `ref[i] - conf[i]` vector in the `ref` unit cell.

Interaction with others units systems

Jumos uses it's own unit system, and do not track the units in the code. All the interaction with units is based on the SIUnits package. We can convert from and to internal representation using the following functions :

`internal`(*value::SIQuantity*)
Converts a value with unit to its internal representation.

```
julia> internal(2m)          # Distance
    1.9999999999999996e10

julia> internal(3kg*m/s^2)   # Force
    7.17017208413002e-14
```

`with_unit`(*value::Number*, *target_unit::SIUnit*)
Converts an internal value to its value in the International System. You shall note that units are not tracked in the code, so you can convert a position to a pressure. And all the results are returned in the main SI unit, without considering any power-of-ten prefix.

This may leads to strange results like:

```
julia> with_unit(45, mJ)
    188280.0 kg m²/s²

julia> with_unit(45, J)
    188280.0 kg m²/s²
```

This behaviour will be corrected in future versions.

Developping new algorithms

Writing a new integrator

To create a new integrator, you have to subtype the `BaseIntegrator` type, and provide the `call` method, with the following signature: `call(::BaseIntegrator, ::MolecularDynamic)`.

The integrator is responsible for calling the `get_forces!(::MolecularDynamic)` function if it need the `MolecularDynamic.forces` field to be updated. It should update the two *Array3D* (page 19): `MolecularDynamic.frame.positions` and `MolecularDynamic.frame.velocities` with appropriate values.

The `MolecularDynamic.masses` field is a `Vector{Float64}` containing the particles masses, in the same order than in the current simulation *frame* (page 20). Any other required information should be stored in the new `BaseIntegrator` subtype.

Computing the forces

Naive force computation The `NaiveForceComputer` algorithm computes the forces by iterating over all the pairs of atoms, and calling the appropriate interaction potential. This algorithm is the default in *Jumos*.

New algorithm for forces computations To create a new force computation algorithm, you have to subtype `BaseForcesComputer`, and provide the method `call(::BaseForcesComputer, forces::Array3D, frame::Frame, interactions)`.

This method should fill the forces array with the forces acting on each particles: `forces[i]` should be the 3D vector of forces acting on the atom `i`. In order to do this, the algorithm can use the `frame.posistions` and `frame.velocities`. `interactions` is a dictionary associating tuples of integers (the atoms types) to *potential* (page 6). The `get_potential` function can be useful to get a the potential function acting on two particles.

`get_potential` (*interactions*, *topology*, *i*, *j*)
Returns the potential between the atom i and the atom j in the topology.

Due to the internal unit system, forces returned by the potentials are in $kJ/(mol \cdot A)$, and should be in $uma \cdot A/fs^2$ for being used with the newton equations. The conversion can be handled by the unexported `Simulations.force_array_to_internal!` function, converting the values of an Array3D from $kJ/(mol \cdot A)$ to $uma \cdot A/fs^2$.

Adding a new check

Adding a new check algorithm is as simple as subtyping `BaseCheck` and extending the `call(::BaseCheck, ::MolecularDynamic)` method. This method should throw an exception of type `CheckError` if the checked condition is not fullfiled.

type `CheckError`
Customs exception providing a specific error message for simulation checking.

```
julia> throw(CheckError("This is a message"))
ERROR: Error in simulation :
This is a message
in __text at no file (repeats 3 times)
```

Adding new controls

To add a new type of control to a simulation, the main way is to subtype `BaseControl`, and provide two specialised methods: `call(::BaseControl, ::MolecularDynamic)` and the optional `setup(::BaseControl, ::MolecularDynamic)`. The `call` method should contain the algorithm implementation, and the `setup` method is called once at each simulation start. It should be used to add add some *computation algorithm* (page 11) to the simulation.

Computing values

To add a new compute algorithm (`MyCompute`), we have to subtype `BaseCompute` and provide specialised implementation for the `call` function; with the following signature:

`call` (*::MyCompute*, *::MolecularDynamic*)
This function can set a `MolecularDynamic.data` entry with any kind of key to store the computed value.

Outputing values

An other way to create a custom output is to subtype `BaseOutput`. The subtyped type must have two integer fields: `current` and `frequency`, and the constructor should initialize `current` to 0. The `write` function should also be overloaded for the signature `write(::BaseOutput, ::Dict)`. The dictionairy parameter contains all the values set up by the *computation algorithms* (page 11), and a special key `:frame` refering to the current simulation *frame* (page 20).

`BaseOutput` subtypes can also define a `setup(::BaseOutput, ::MolecularDynamic)` function to do some setup job, like adding the needed computations to the simulation.

As an example, let's build a custom output writing the `x` position of the first atom of the simulation at each step. This position will be taken from the frame, so no specific computation algorithm is needed here. But this position will be writen in bohr, so some conversion from Angstroms will be needed.

```
# File FirstX.jl

using Jumos

import Base.write
import Jumos.setup

type FirstX <: BaseOutput
    file::IOStream
    current::Integer
    frequency::Integer
end

# Default values constructor
function FirstX(filename, frequency=1)
    file = open(filename, "w")
    return FirstX(file, 0, frequency)
end

function write(out::FirstX, context::Dict)
    frame = context[:frame]
    x = frame.positions[1][1]
    x = x/0.529 # Converting to bohr
    write(out.file, "$x \n")
end

# Not needed here
# function setup(::FirstX, ::MolecularDynamic)
```

This type can be used like this:

```
using Jumos
require("FirstX.jl")

sim = MolecularDynamic(1.0)
# ...

add_output(sim, FirstX("The-first-x-file.dat"))
```

3 Installation

To install, simply run `Pkg.clone(https://github.com/Luthaf/Jumos.jl)` at julia prompt. You may also want to run `Pkg.test("Jumos")` to run the tests.

Only 0.4 julia prerelease version is supported, because *Jumos* makes use of features from the 0.4 version.

Index