molsim tutorial

for the 1.x.x series

J.S. Hansen

January 2023

1 Introduction

molsim is a GNU Octave/Matlab toolbox for molecular dynamics simulations. molsim supports simulations of

- simple Lennard-Jones systems,
- molecular systems with bond, angle, and torsion potentials,
- confined flow systems, eg., Couette and Poiseuille flows,
- charged systems using shifted force and Wolf methods,
- dissipative particle dynamics systems,
- different ensembles,
- and more . . .

molsim is basically a wrapper for the seplib library, which is a light-weight flexible molecular dynamics simulation library mainly written in ISO-C99. With version series 1.x.x. seplib offers both shared memory parallelization for CPUs and GPU simulations using Nvidia's CUDA toolkit; this is supported by molsim. seplib is also developed and maintained by this author, and the underlying algorithms are based on the books by Allen and Tildesley [1], Frenkel and Smit [2], Rapaport [3], and Sadus [4].

In this text

```
>>
```

symbolizes the GNU Octave/Matlab command prompt. This

```
$
```

symbolizes the shell prompt.

Example scripts and functions to simulate different systems can be found under the package `tests` directory. It is highly recommended that the user's project starts from one of these `.m`-files, and the user then makes the necessary changes.

2 Installation

2.1 GNU Octave

GNU Octave's package manager offers a very easy installation. From the command prompt type (one single line)

```
>> pkg install "https://github.com/jesperschmidthansen/molsim/archive/ \
                                                  refs/tags/v<version>.tar.gz"
```

to install the package. `<version>` can be for example `0.9.2`. Check contact to `molsim` by

```
>> molsim('hello')
Hello.
```

You can also download the `tar.gz` file manually from

```
https://github.com/jesperschmidthansen/molsim
```

and save it in some directory of your choice. From this directory enter GNU Octave and type

```
>> pkg install molsim-<version>.tar.gz
```

2.2 Matlab

From

```
https://github.com/jesperschmidthansen/seplib/
```

download and save the current release `seplib-<version>.tar.gz` in a directory of your choice. Unpack, configure and build the library

```
$ tar zxvf seplib-<version>.tar.gz
$ cd seplib
$ ./configure
$ make
$ cd octave
```

To build the mex-file enter Matlab

```
$ matlab -nodesktop
```

and run the script `buildmex`, that is,

```
>> buildmex
```

Depending on the system this will build a `molsim.mex<archtype>` file; `<archtype>` being your computer architecture. Copy this file to a directory in your Matlab search path.

Note: Matlab compatibility is not guarantied. molsim will only be tested against very limited Matlab versions.

3 The interface strategy

This tutorial is not meant to introduce molecular dynamics; such introductions can be found in the books listed in the reference list. In brief, the basic idea is to solve the classical equation of motion of an ensemble of interacting particles. In the simplest form this means solving (numerically) Newton's second law

$$\frac{\mathrm{d}\mathbf{r}_i}{\mathrm{d}t} = \mathbf{v}_i \ , \quad \frac{\mathrm{d}\mathbf{p}_i}{\mathrm{d}t} = \mathbf{f}_i \ , \tag{1}$$

where $\mathbf{r}_i, \mathbf{v}_i, \mathbf{p}_i$ and $\mathbf{f}_i$ are the particle position, velocity, momentum and force acting on the particle, respectively. In a standard simulation we solve this set of differential equations by (i) evaluating the forces acting on the particles, and (ii) from this integrate forward in time. The following pseudo code lists the basic idea

Listing 0

```
Set simulation parameters
Set initial configuration $\mathbf{r},\mathbf{p}$

do (as many times as we want)
   $\mathbf{f} \leftarrow$ calcforce($\mathbf{r}$)
   $\mathbf{r},\mathbf{p} \leftarrow$ integrate($\mathbf{f}$,$\mathbf{p}$)
done
```

—

The molsim interface seeks to emulate this, and give the user coding flexibility and accessibility to the simulation quantities. In general, the `molsim` interface is of the form

```
molsim(<action>, <specifier>, <arguments>);
```

The action can be any particular action the user wishes to perform, for example, `calcforce`, `integrate`, and so on. The action is specified by the second argument; say, `lj` specifies that action `calcforce` should apply the Lennard-Jones interaction. The specifier arguments are given in the final input and can be a scalar, string, vector, or a sequence of these.

4 First quick example: The Lennard-Jones liquid

Listing 1 shows a script simulating a standard Lennard-Jones (LJ) system in the micro-canonical ensemble, where number of particles, volume, and total mechanical energy is conserved.

Listing 1

```
% Specify the LJ parameters
cutoff = 2.5; epsilon = 1.0; sigma = 1.0; aw=1.0;

% Set init. position and velocities 10x10x10 particles
% in box with lengths 12x12x12. Velocities set to default.
% Configuration stored in start.xyz.
molsim('set', 'lattice', [10 10 10], [12 12 12]);

% Load the configuration file
molsim('load', 'xyz', 'start.xyz');
```

```
% Main mol. simulation loop - 10 thousand time steps
for n=1:10000

  % Reset forces etc
  molsim('reset');

  % Calculate force between particles of type A (default type)
  molsim('calcforce', 'lj', 'AA', cutoff, sigma, epsilon, aw);

  % Integrate forward in time - use leapfrog alogrithm
  molsim('integrate', 'leapfrog');

end

% Free memory allocated
molsim('clear');
```

—

I hope the code is, mostly, clear. Notice, that in action `calcforce` the first argument to the specifier `'lj'`, namely, `'AA'` tells the program that this force acts between particles of type `A`. This type is the default type; a very important point I highlight

The default particle type is A

In Listing 1 no information is printed or saved, and, admitted, not very useful. Inside the main loop the user can call the print action, for example,

```
if rem(n,100)==0
  molsim('print');
end
```

to print current iteration number, potential energy per particle, kinetic energy per particle, total energy per particle, kinetic temperature, and total momentum to screen every 100 time steps.

Information can also be stored into variables for further analysis. For example, to get the system energies and pressure

```
energies = molsim('get', 'energies'); # vector with ekin and epot
press = molsim('get', 'pressure');
```

and particle positions and velocities

```
x = molsim('get', 'positions');
v = molsim('get', 'velocities');
```

In the reference sheet (see Appendix) you can find the list of specifiers for the `get` action.

Important note For molecular systems the pressure is calculated using the molecular pressure tensor. In general this is different from the atomic pressure. The user must enable this calculation using the `set` action with specifier `molstresscalc`. For example, to calculate the (molecular) pressure every ten time steps use

```
molsim('set', 'molstresscalc', 10);
```

before the main loop. Then in the main loop retrieve the pressure by

```
if rem(n,10)==0
  [press_atomic, press_mol]=molsim('get', 'pressure');
end
```

4.1 NVT and NPT simulations

Often you will not perform simulations in the micro-canonical ensemble, but under a desired temperature and/or pressure. To perform simulations under specific temperatures you can choose one of three different methods:

Relaxation: Using this method call the action `'thermostat'` with specifier `'relax'` *after* the integration step, for example, to thermostat particles of type `A` such that their kinetic temperature is 2.2 use

```
molsim('thermostat', 'relax', 'A', 2.2, 0.01);
```

The last argument is the relaxation parameter; the higher value the faster relaxation. Notice that too large values make the system unrealistically stiff; the best value is optimized via trail-and-error.

Nosé-Hoover: There is also a Nosé-Hoover thermostat available. This is called by

```
molsim('thermostat', 'nosehoover','A' ,2.2, 0.1);
```

The last argument is here the inverse thermostat mass. Again, you should choose this parameter with care. Impotantly, this thermostat is called *before* the leap-frog integrator call.

GJF-Langevin integrator: This method *replaces* the leap-frog integrator, by

```
molsim('integrate', 'langevin', 2.2, 0.5);
```

The last argument is the thermostate coupling parameter, α. Typically $0.01 \leq \alpha \leq 1.0$ See Ref. [6].

To simulate at pressure, say 0.9, you call the action `'barostat'` *after* the integration step,

```
molsim('barostat', 'relax', 0.9, 0.01, 'iso');
```

The choice of relaxation parameter, here 0.01, is again a matter of the specific system. The last argument tells the barostat to do an isotropic compression. If this is left out the barostat works by changing the system box length in the z-direction only (anisotropic scaling); this is practical when doing sampling as two directions are fixed. You can use the barostat and the thermostat in the same simulation mimicking an NPT system. For the expert: The barostat is based on the atomic pressure, a molecular pressure barostat is planned for future releases.

5 The molsim force field

`molsim` supports simulations of more complicated systems. In general, the molsim force field is defined from the potential function

$$U(\mathbf{r}_i, r_{ij}, \ldots) = U_{\text{lattice}} + U_{\text{vWaals}} + U_{\text{coulomb}} + U_{\text{bonds}} + U_{\text{angles}} + U_{\text{torsion}} \quad (2)$$

The first term allows for simulations of fictitious fixed crystal arrangements, where the particles/atoms are tethered around a pre-set lattice site. This is particularly useful for systems with walls. The potential function is a harmonic spring type

$$U_{\text{lattice}} = \sum_{\text{sites}} \frac{1}{2} k_0 (\mathbf{r}_i - \mathbf{r}_0)^2 \ , \quad (3)$$

where k_0 is the spring constant, $\mathbf{r}_i$ is the position of particle/atom i, and $\mathbf{r}_0$ is the virtual lattice site. Using this requires that the virtual crystal sites are set: use `molsim('set','virtualsites');` to set the current positions as crystal sites. The force from this potential is calculated by

```
molsim('calcforce', 'lattice', <part. type>, <k0>);
```

where `<part. type>` is the particle type and `<k0>` is the force constant.

The short-range van der Waals pair interaction is given via the 12-6 Lennard-Jones potential

$$U_{\text{vWaals}} = \sum_{i,j \text{ pairs}} 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - a_w \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] . \tag{4}$$

Here r_{ij} is the particle distance, ϵ and σ define the characteristic energy and length scales. The parameter a_w determines the weight of the attractive second term in the potential function. The force call is

```
molsim('calcforce', 'lj', <pair>, <cutoff>, <sigma>, <eps>, <aw>);
```

where `<cutoff>` is the maximum interaction length (or cut-off). This must be less than or equal to the maximum system interaction length which is by default 2.5. The system maximum interaction length can be set via the `set` action

```
molsim('set', 'cutoff', <value>);
```

The Coulomb potential is

$$U_{\text{coulomb}} = \sum_{i,j \text{ pairs}} \frac{q_i q_j}{r_{ij}} . \tag{5}$$

Currently this long-range interaction is evaluated using approximate shifted-force or Wolf methods; this can be specified. Note: these two algorithms do not apply to confined systems. The call is

```
molsim('calcforce', 'coulomb', <method>, <cutoff>, <optWolf>);
```

`<method>` can take value `'sf'` or `'wolf'`, and `<optWolf>` is the Wolf screening parameter which must be specified if the Wolf method is chosen. Again, the cut-off must be less than or equal to the maximum system interaction length.

Bonds are modelled via the harmonic spring potential

$$U_{\text{bonds}} = \sum_{\text{bonds}} \frac{1}{2} k_s (r_{ij} - l_0)^2 . \tag{6}$$

k_s is the spring constant and l_0 is the zero force bond length. Currently molsim does not support rigid bonds. To calculate the force from bonds use

```
molsim('calcforce', 'bond', <type>, <ks>);
```

`<type>` specifies the specific bond type. Bond, angle, and torsion angle types are specified through integers, see example script later.

The angle potential is the cosine squared potential

$$U_{\text{angles}} = \frac{1}{2} \sum_{\text{angles}} k_\theta (\cos(\theta) - \cos(\theta_0))^2 \,, \tag{7}$$

where k_θ is the force amplitude, and θ_0 the zero-force angle. See Fig. 1 for the angle definition. The call is

```
molsim('calcforce', 'angle', <type>, <a0>, <ka>);
```

where `<type>` specifies the angle type, `<a0>` the zero force angle, θ_0, and `<ka>` the force constant, k_θ.

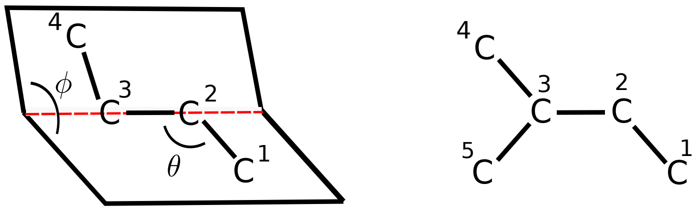

Figure 1: Illustration of the angle and torsion angle.

Finally, the torsion angle potential is the Ryckaert-Belleman potential

$$U_{\text{torsion}} = \sum_{\text{angles}} \sum_{n=0}^{5} c_n \cos^n(\pi - \phi) \,. \tag{8}$$

Here c_n are the six Ryckaert-Belleman coefficients, and ϕ is the torsion angle, see Fig. 1. Two illustrative examples are when only $c_1 \neq 0$:

1. If $c_1 > 0$ then the minimum energy torsion angle is $\phi = 0$; this is illustrated in the right-hand figure of a planar molecule with the torsion angle defined by the 1-2-3-5 bonds.

2. If $c_1 < 0$ then the minimum energy torsion angle is $\phi = \pi$; this is the torsion angle defined by the 1-2-3-4 bonds.

To calculate the force from this interaction potential use

```
molsim('calcforce', 'torsion', <type>, <RB-coef>);
```

`<RB-coef>` is an array of length six specifying the Ryckert-Belleman coefficients. Note, the torsion angle is often referred to as the dihedral angle.

6 Molecular systems: Toluene

This example shows how to setup a simulation of liquid toluene. The model of the molecule is a so-called united atomic unit (UAU) model. This means that each carbon group is represented by a single Lennard-Jones particle, thus, the toluene molecule is composed of seven identical Lennard-Jones particles, six forming the phenyl ring structure (particle indices 2-7) and one representing the methyl group (index 1). We exclude the molecular dipole moment, i.e., we do not apply any charges to the system. The model molecule is shown in Fig. 2.

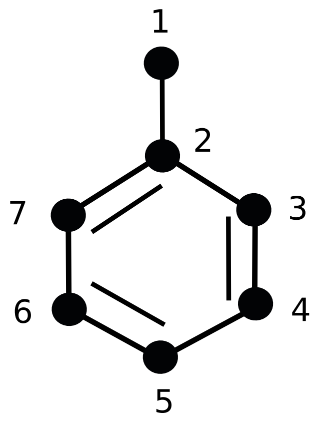

Figure 2: United atomic unit representation of toluene.

To define the molecule geometry (or topology) we need different intramolecular interactions, i.e., bond, angle, and torsion angle potentials. Lennard-Jones interactions between carbon groups in the same molecule are excluded. The model is further simplified by using only two different bond types (with different zero force bond length, but the same force constant), and one angle type. There are two different torsion angles, eg., 1-2-3-4 form one type of torsion angle, $\phi = \pi$, whereas 7-2-3-4 form a torsion angle with $\phi = 0$. We define the molecular model in two files; one with extension `.xyz` giving the carbon groups' positions for one molecule and one with extension `.top` defining bonds and angles in the molecule. You can find examples of these two files for different molecules under the `resources` directory.

To setup the entire system (i.e. the ensemble of molecules) we copy the single molecule `.xyz` and `.top` to the current directory and use the `set` action; for example to simulate 500 molecules

```
>> molsim('set', 'molconfig', 'toluene.xyz', 'toluene.top', ...
   500, 0.05, 42)
```

The two last arguments are the molecular number density (keep very low initially and compress the system afterwards), and a seed for the random number generator. This generates a system `start.xyz` file and `start.top` file that can be loaded by your program.

We now only need the parameter values for the interaction potentials and we will simply use what is available in the literature [5] and convert them into MD reduced units. Listing 2 shows the resulting script

Listing 2

```
% Simulation parameters
% (Corresponds to 298.15 K, 862 kg/m^3)
temp0 = 4.969; dens0 = 1.96; dt  = 0.001; nloops = 200000;

% Intra-molecular parameters
bondlength_0 = 0.4; bondlength_1 = 0.38; springconstant = 48910;
bondangle = 2.09; angleconstant = 1173;

torsionparam_0 = [0.0, 133.0, 0.0, 0.0, 0.0];
torsionparam_1 = [0.0, -133.0, 0.0, 0.0, 0.0];

% Load positions, set temp, remove intra-molecular pair-interaction etc
molsim('load', 'xyz', 'start.xyz');
molsim('load', 'top', 'start.top');

molsim('set','timestep', dt);
molsim('set', 'temperature', temp0);
molsim('set', 'exclusion', 'molecule');

% Main loop
for n=1:nloops
  molsim('reset')

  molsim('calcforce', 'lj', 'CC', 2.5, 1.0, 1.0, 1.0);

  molsim('calcforce', 'bond', 0, bondlength_0, springconstant);
  molsim('calcforce', 'bond', 1, bondlength_1, springconstant);
  molsim('calcforce', 'angle', 0, bondangle, angleconstant);

  molsim('calcforce', 'torsion', 0, torsionparam_0);
  molsim('calcforce', 'torsion', 1, torsionparam_1);

  molsim('thermostate', 'nosehoover', 'C', temp0, 0.01);
  molsim('integrate', 'leapfrog');

  molsim('compress', dens0);

end
```

—

Notice that

- `molsim('set', 'exclusion', 'molecule');` ensures that van der Waals and the Coulomb interactions are excluded if the particles are in same molecule. Exclusion can also be set for only bonded particles using the 'bond' argument.
- For the `bond`, `angle`, and `torsion` specifiers the first argument pertains to the bond/angle/torsion type.

7 Sampling

The user can access the system configuration through the `get` action and from this perform data analysis via GNU Octave's or Matlab's built-in tools. molsim also offers some run-time data sampling. The different samplers are initialized before the main loop using the `sample`-action

```
molsim('sample', <sample specifier>, <arguments>);
```

For example, to sample the stress autocorrelation function with 200 sample points and over a sample time span window of 5.0 we write

```
molsim('sample', 'sacf', 200, 5.0);
```

The actual sampling is carried out by the specifier `do`; inside the main loop there must be one call

```
molsim('sample', 'do');
```

Typically this call is done just after the integration. Check the reference sheet for the list of available samplers.

8 Parallelization

molsim offers two types of shared memory parallelization, namely,

- Loop parallelization
- Task-block parallelization

We here only document the first type[1]; the task-block type will be included in later versions of the tutorial. To use loop parallelization simply call the `set` action with specifier `omp`

```
molsim('set', 'omp', <nthreads>);
```

where `nthreads` is the number of threads you wish to use. Typically, do not use more threads than the number of cpu-cores[2]. The call is placed anywhere before the main loop. Be aware that depending on your hardware and the particular system the parallelization efficiency quickly drops as a function of the number of threads. To explore this we define the speed-up and efficiency by

$$\text{speedup} = \frac{t_\text{single}}{t_\text{parallel}} \quad \text{and} \quad \text{efficiency} = \frac{t_\text{single}}{t_\text{parallel} N_\text{threads}}\,, \tag{9}$$

where t_single is the single thread execution time and t_parallel the parallel execution time. The speed-up is plotted in Fig. 3 for a simple Lennard-Jones liquid simulation. The efficiency is also plotted in Fig. 3. The efficiency quickly drops, even on this multi-core machine, and adding more threads (using more cpu cores) is not always optimal. In the `tests`-directory you can find `molsim_runparallel.m` that times the execution time for a given system size, density and number of threads; use the `help` command for its usage.

9 CUDA support

10 Analysis tools

`molsim` ships with a series of m-functions that can be used for further system analysis after the simulation. Some of the functions depend on the system's configuration at different times; in order to store these configurations the user can use the `save` action

[1] Yep - I'm lazy

[2] not even with hyper-threading

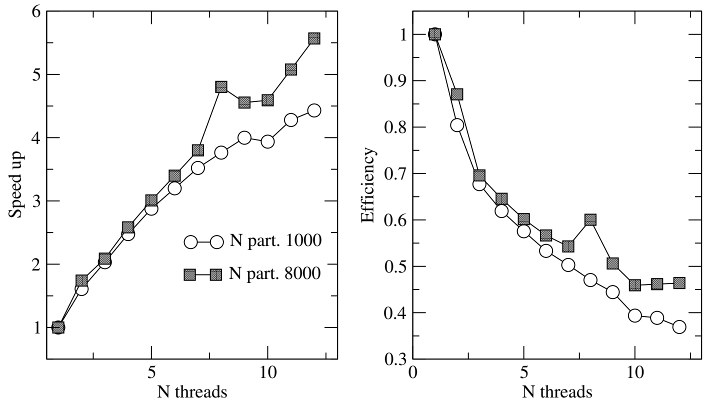

Figure 3: Left: Speed-up as function of number of threads. Right: Efficiency as function of number of threads. The test machine is a 72-core machine.

```
molsim('save', <types>, <filename>);
```

`<types>` enables one to specify the type of particles for which the configuration should be stored, and `<filename>` the filename where the configuration is stored (typically with suffix `xyz`).

```
fileid = 0;

for n=1:nloops
   ...

   if rem(n,100)==0
      molsim('save', 'AB', sprintf('molsim-%d.xyz', fileid));
      fileid++;
   end

   ...
end
```

Important note The analysis tools are only supported by GNU Octave

```
molsim_readxyz
molsim_calcrdf
molsim_calcmsd
```

References

[1] M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids*, (1989).

[2] D. Frenkel and B. Smit, *Understanding Molecular Simulation*, (1996).

[3] D. C. Rapaport, *The Art of Molecular Dynamics Simulation*, (1995).

[4] R. J. Sadus, *Molecular Simulation of Fluids. Theory, Algorithms and Object-Orientation*, (1999).

[5] J.S. Hansen *Where is the hydrodynamic limit?* Mol. Sim., 47:1391 (2021).

[6] *A simple and effective Verlet-type algorithm for simulating Langevin dynamics* Mol. Phys., 111:983 (2013).

Reference sheet

Action	Specifier	Arguments	Output	CUDA
`load`	`xyz`	file name		X
	`top`	file name		X
`save`		1: type names 2: file name		X
`set`	`timestep`	time step (0.005)		X
	`temperature`	temperature (1.0)		X
	`cutoff`	Max. cut-off (2.5)		X
	`omp`	No. of threads		
	`exclusion`	'bond' or 'molecule'		X
	`temperaturerelax`	relaxation time (0.01)		
	`compressionfactor`	compress factor (0.9999)		
	`types`	particle types (vector string)		
	`mass`	particle mass (vector)		
	`skin`	buffer-skin neighblist		X
	`charges`	atom charges (vector)		
	`lattice`	1: array $[N_x, N_y, N_z]$ 2: array $[L_x, L_y, L_z]$		
	`molconfig`	1: xyz file 2: top file 3: No. molecules 4: Crystal density 5: Random seed		
	`virtualsites`			
	`molstresscalc`	iterations between calculation		

Action	Specifier	Arguments	Output	CUDA
`get`	`numbpart`		scalar	X
	`box`		3-vector	
	`energies`		2-vector (kin, pot)	X
	`pressure`		scalar(s) (p, pmol)	X*
	`velocities`		$N_{\text{part.}} \times$ 3-matrix	X
	`positions`		$N_{\text{part.}} \times$ 3-matrix	X
	`forces`		$N_{\text{part.}} \times$ 3-matrix	
	`types`		$N_{\text{part.}}$-string	X
	`mass`		$N_{\text{part.}}$-vector	X
	`charges`		$N_{\text{part.}}$-vector	X
	`molpositions`		$N_{\text{mol}} \times$ 3-matrix	
	`molvelocities`		$N_{\text{mol}} \times$ 3-matrix	
	`indices`	1: mol. index	N_{uau}-vector	
	`bondlengths`		N_{nbonds}-vector	
	`angles`		N_{nangles}-vector	
	`torsions`		N_{ntorsion}-vector	

Action	Specifier	Arguments	Output	CUDA
calcforce	lj	1: part. types 2: cutoff 3: σ 4: ϵ 5: a_w		X
	coulomb	1: method 2: cutoff 3: opt. Wolf param.		X
	bond	1: type 2: bond length 3: spring constant		X
	angle	1: type 2: zero angle force 3: force constant		X
	torsion	1: type 2: tors. param		X
	lattice	1: part. type 2: spring constant		
	dpd	1: part. types 2: cutoff 3: rep. parameter 4: σ		

Action	Specifier	Arguments	Output	CUDA
`integrate`	`leapfrog`			X
	`dpd`	λ		
	`langevin`	1:target temperature 2:α (0.01-1.0)		
`thermostat`	`relax`	1: type 2: target temperature 3: relax time		
	`nosehoover`	1: type (optional) 2: target temperature 3: thermostate mass (10-50)		X
`barostat`	`relax`	1: target pressure 2: relax time 3: `'iso'` (optional)		
`compress`		1: target density/length 2: opt for length compression, the direction		
`add`	`force`	1: force vector 2: direction (0,1,2)		
	`tolattice`	1: dx (scalar) 2: direction (0,1,2)		
`clear`				

Action	Specifier	Arguments	Output
`sample`	`vacf` or `mvacf`	1: sample vector length 2: sample time span	
	`sacf` or `msacf`	1: sample vector length 2: sample time span	
	`msd`	1: sample vector length 2: sample time span 3: no. wavevectors 4: particle type	
	`radial`	1: sample vector length 2: step between samples 3: particle types	
	`hydrocorrelations` or `mhydrocorrelations`	1: sample vector length 2: sample time span 3: no. wavevectors Wavevector direction is y-direction Transverse direction is x-direction	
	`profiles` or `mprofiles`	1: particle type 2: sample vector length 3: sample interval	
	`do`		
`reset`			
`print`			
`convert`		1: σ 2: ϵ/k_B 3: m	2× structs

Action	Specifier	Arguments	Output
`task`	`lj`	1: part. types	
		2: cutoff	
		3: σ	
		4: ϵ	
		5: a_w	
		6: block id	
	`bond`	1: type	
		2: bond length	
		3: spring constant	
		4: block id	
	`angle`	1: type	
		2: zero angle force	
		3: force constant	
		4: block id	
	`torsion`	1: type	
		2: tors. param	
		3: block id	
	`coulomb`	1: cutoff	
		3: block id	
	`do`	no. blocks	