ALE Molecular Dynamics Documentation

Anton Olsson
Daniel Stannelind
Daniel Spegel-Lexne
Martin Clason
Gustaf Åhlgren

Jan 12, 2022

CONTENTS:

This is the documentation for the molecular dynamics program Ale. Ale is designed to be used in investigating hypothetical materials and get a peak into their properties. Ale is built upon the ASE (Atomic Simulation Environment) and ASAP (As Soon As Possible) is used to speed up the calculations. Ale also supports importing openKIM interatomic potentials to further enhance the relevance of the simulation results.

As of now the program has been mostly tested on single element materials but does support multi element simulation in arbitrary cubic configurations. For more info consider reading the limitations.

CHAPTER

ONE

MAIN FUNCTIONS

1.1 Ale simulate

Ale simulate runs the molecular dynamics simulation using classical mechanics to evaluate the equations of motion for the system that is simulated. This does not limit the program to use only classical interatomic potentials in the simulations and it's highly encouraged to use the built in support to import interatomic potentials from openKIM which comes in a variety of forms and is in most cases specific to the system that is simulated, which yields the best prediction of properties.

1.2 Ale analyze

Ale analyze is the module that calculates material properties from the behavior of the atoms in the molecular dynamics simulation and stores them in a line .json file for easier analysis in Ale visualize or in a third party software.

1.3 Ale visualize

Ale visualize allows the user to plot properties across the duration of the simulation or to create scatter plots of two properties in a large set of materials to compare large sets of materials or material categories.

1.4 Ale multi

Ale multi allows the user to generate a combination of simulation setups and run them in parallel, either on a multi-core pc or on a supercomputer. This can be useful for investigating large sets of materials or for researching differences between material categories.

1.4.1 Installation

1.4.1.1 Install dependencies

Using conda

Conda can be used to create an environment suitable for ale to run in. This environment could be called `my-md-env` for example. This oneliner could be executed to create the environment and install the packages in one go:

```
$ conda create -c conda-forge -n my-md-env python=3 ase asap3 kimpy kim-api openkim-
↪models Cython numpy scipy matplotlib mpi4py pytest openmpi
```

Followed by:

```
$ conda activate my-md-env
```

Alternatively, the `requirements.txt` could be used instead:

```
$ conda create -c conda-forge -n my-md-env python=3
$ conda activate my-md-env
$ conda install -c conda-forge --file requirements.txt
```

Using pip

Pip could be used instead but conda has been used during development in this project.

Install ASE:

```
$ pip install ase
```

Install ASAP:

```
pip install asap3
```

Other packages will need to be installed. See contents of `requirements.txt`.

1.4.1.2 Install the software

To be able to run ale in the terminal in your current environment, download this git repository and navigate into it. Then run:

```
$ python -m pip install .
```

This will read the script `setup.py` and pip will install `ale` as a command line tool. It will also install its dependencies but for this to work your environment must have installed `kim-api`, `openkim-models` and `openmpi`. These can be installed with e.g. conda.

IMPORTANT: To develop without having to reinstall ale all the time you can instead run:

```
$ python -m pip install --no-deps -e .
```

This will install `ale` without dependencies and in editable mode so the source code can be edited without having to reinstall ale for the changes to take effect.

To test if ale is installed correctly you can now run:

```
$ ale -h
```

If it shows the help message the installation worked!

1.4.2 Running the software

Run ale help for ale and its various subcommands:

```
$ ale -h
$ ale simulate -h
$ ale analyze -h
$ ale visualize -h
$ ale multi -h
```

Run ale (both simulation and analyzation) using the default config file `config.yaml`:

```
$ ale
```

Without asap and with a special config (`--config` can also be used):

```
$ ale --no-asap -c my_config.yaml
```

All ale modules can be handed an output directory as a command line argument using the `-d` or `--dir` flag together with the path to the output directory relative to the current working directory. The output files are named the symbol(s) (the elements) in the simulation with either `.traj` or `.json` depending on the data. The names of the output file with the flag `-t` or `--traj` to name the trajectory output files and the flags `-o` or `--out` to name the output data files (`.json`).

To only run a simulation:

```
$ ale simulate -c my_config.yaml
```

It's also possible to only run analyzation and calculate properties from existing output files. The user doesn't have to specify a trajectory file if `ale analyze` is started from a directory where both the config file and the corresponding trajectory file are located. The command looks as follows:

```
$ ale analyze -c my_config.yaml
```

However a safer approach is to specify both the config file and the corresponding trajectory file. Then the command is:

```
$ ale analyze -c my_config.yaml -t symbol.traj
```

It's also possible to visualize things like results from the simulations and the analysis. In the config file the user can specify which quantities that should be plotted and in which directory the files containing properties for the scatter plot are located. The config file is specified with the flag `-c`. The directory for the output files can be specified with the flag `-d`. The directory containing the properties for the scatter can be specified with the flag `-s`. To run visualization:

```
$ ale visualize -c my_config.yaml -d out_dir -s scatter_dir
```

1.4.2.1 Running the software without installing it as a package

If you for some reason want to run the code without having to install it as a package with `pip` you can do the following. Make sure you're in the project directory, that all dependencies are installed and run it as a python module with the following command:

```
$ python -m ale
```

This line can be followed by the arguments, e.g. `python -m ale -h`, as usual.

It's probably better to try to install it as a package using `pip` though. That way it will be possible to run `ale` from any directory and the dependencies should automatically be installed by pip.

Running `ale multi` currently requires `ale` to be installed with `pip`.

1.4.2.2 Running ale multi

Ale multi needs two config files to be run, one base config file and one multi config file, which specifies which simulations to generate from the multi config with the config file as the template or base. Ale multi requires the user to specify an output directory to store the output files in. To run ale multi, `multi_config.yaml` is the multi config file and `out_dir` is the name of the output directory for the generated files:

```
ale multi multi_config.yaml -c base_config.yaml out_dir
```

1.4.2.3 Running tests

To run the unit tests and integrations tests with pytest run:

```
$ make test
```

1.4.2.4 Building documentation

To build the documentation either navigate to docs and run sphinx own Makefile or trigger the build from the base directory by running:

```
$ make html-doc
```

To view the docs you then open the file `docs/_build/html/index.html`.

1.4.3 The config files

1.4.3.1 Single run config file:

The stand-alone config file contains the following fields and an entire simulation and analysis can be created from this file.

Simulate:

The user can specify a `latticeconstant` to use for the simulation but is this is left empty Ale will compute a lattice constant with the `guess_latticeconstant` as the initial value. If `guess_latticeconstant` is left empty Ale has a fallback value of 4 Å.

```
latticeconstant: <double [Å]>
guess_latticeconstant: <double>
```

The field `cell` specifies the unit cell structure. The user can specify fcc or bcc bravais lattices but other lattices has to be specified with a base matrix. Ale only support single parameter cubic lattices.

```
cell: <string or matrix>
```

The field `scaled_positions` is the instruction on where to position different species of atoms relative to the origin of the unit cell. The length scale is normalized to the unit cell size (to the lattice constant). This parameter is only important for multi element systems and `[[0,0,0]]` should be used when simulating single element systems.

```
scaled_positions: <array of 3D positions>
```

The field `symbol` specifies the element or a string of elements to be simulated such as `"Au"` for gold or `"CuK"` for a copper and potassium alloy. These elements are then placed at the `scaled_positions` in order.

```
symbol: <string>
```

To use periodic boundary conditions enter `True` in the `pbc` field.

```
pbc: <bool>
```

The field `size` specifies how many times to repeat the unit cell to a super cell in each dimension (e.g. 2 -> 2^3 = 8 times as large).

```
size: <int>
```

This field specifies which ensemble to simulate. Ale currently only supports `"NVT"` or `"NVE"` ensembles.

```
ensemble: <string>
```

This field specifies how hard the thermostat should correct the temperature. The friction number is usually `1E-4` to `1E-2`.

```
NVT_friction: <double>
```

This field specifies the initial temperature of the simulation. If the temperature should remain close to the specified value enter `"NVT"` in the `ensemble` field.

```
temperature_K: <double [K]>
```

This field specifies whether Ale should check that the simulated system has reached equilibrium before writing to the output trajectory file. This check either terminates when equilibrium is obtained or when the check timeout is reached.

```
checkForEquilibrium: <bool>
```

This field specifies which interatomic potential to use. The recommended potentials are those found in the openKIM library and these are designated with `"openKIM:<potential_name>"`, in this case sigma and epsilon aren't needed. Atomic number, sigma, epsilon (model parameters) and cutoff is only used if the built in Lennard Jones potential is used, this potential is specified with `"LJ"`.

```
potential: <string>
sigma: <double [Å]>
epsilon: <double [Å]>
cutoff: <double [Å]>
atomic_number: <int [unit charge]>
```

This field specifies the timestep for the simulation in femtoseconds.

```
dt: <int [fs]>
```

This field specifies the number of timesteps that should be taken in the simulation. In case the equilibrium check is enabled this is the number of iterations after equilibrium is reached or equilibrium timeout has occurred. In the case the equilibrium check isn't enabled this is the total number of iterations in the simulation.

```
iterations: <int>
```

This field specifies how many timesteps that will be taken between each save of the simulation state to the trajectory file(s).

```
interval: <int>
```

This field specifies if Ale should calculate the cohesive energy of the system which is done after the system has reached equilibrium, or equilibrium timeout as long as the equilibrium check is enabled.

```
calculateCohesiveEnergy: <bool>
```

This field specifies how many iterations the cohesive energy calculation should run for at most.

```
max_iterations_coh_E:
```

Analyze:

This field specifies a list of properties that Ale will calculate in the analyse step.

```
output:
  - <yaml list of strings>
```

The properties that can be calculated are:

```
- Temperature
- Volume
- Specific Heat Capacity
- Density
- Instant Pressure
- Average Pressure
- MSD # Mean Square Displacement
- Self Diffusion Coefficient
- Self Diffusion Coefficient Array
- Lindemann criterion
- Optimal Lattice Constant
- Optimal Lattice Volume
- Bulk Modulus
- Debye Temperature
- Transversal Sound Wave Velocity
- Longitudinal Sound Wave Velocity
- Shear Modulus
- Cohesive Energy
```

Visualize:

This field specifies which visualizations to perform when `ale visualize` is run.

```
visualize:
  - <yaml list of strings>
```

The visualizations that can be performed are:

```
- Lattice
- Temperature
- MSD
- Scatter
```

`Lattice` will use ASE gui to show initial positions of all atoms in the simulated lattice.

`Temperature` plots temperature over time for the simulation.

`MSD` plots the mean square displacement over time for the simulation.

`Scatter` makes scatter plots which have further options described below.

These fields specify which two properties that will be plotted in a scatter plot with d1 on one axis and d2 on the other.

```
scatter_type_d1: <string>
scatter_type_d2: <string>
```

The properties that can be shown in a scatterplot are (not that the chosen data must be available for the scatterplot to work):

```
Temperature
Volume
Specific Heat Capacity
Density
Average Pressure
Self Diffusion Coefficient
Lindemann criterion
Optimal Lattice Constant
Optimal Lattice Volume
Bulk Modulus
Debye Temperature
Transversal Sound Wave Velocity
Longitudinal Sound Wave Velocity
Shear Modulus
Cohesive Energy
```

This field specifies the path to the directory the output properties for the materials that will be included in the scatterplot are relative to where `ale visualize` is run.

```
scatter_dir: <string>
```

This field can be used to specify a subset of the files in the `scatter_dir` that should be used in the scatter plot. If this field is left empty `ale visualize` will look at all files.

```
scatter_files: <array of strings>
```

1.4.3.2 Multi-config:

Ale supports using several processes to start multiple simulations in parallel which can be run locally (on a machine with a multi core processor) or on a supercomputer.

The multi program takes the fields in the <multi_config> and generates several simulations with the <base_config> as the base and substitutes the fields specified in the <multi_config>.

For example:

base_config.yaml

```yaml
#-----------Atoms Setup------------#
guess_latticeconstant: 5
cell: # Given by m_config
scaled_positions : # Given by multi_config
symbol : # Given by multi_config
pbc : True
size : 22

#-----------Simulation Setup------------#
make_traj: True
run_MD: True
ensemble: "NVE"
temperature_K : # Given by multi_config
checkForEquilibrium : True
potential: # Given by multi_config.yaml
dt: 5 # simulation time step [fs]
iterations: 5000
interval: 50

#-----------Analyse------------#
output:
  - Temperature
  - Volume
  - Debye Temperature
  - Self Diffusion Coefficient
  - Density
  - Pressure
  - MSD
  - Self Diffusion Coefficient Array
  - Specific Heat Capacity
  - Lindemann criterion

#-----------Visualize------------#
visualize:
  - Temperature
  - Scatter
scatter_type_d1: "Density"
scatter_type_d2: "Specific Heat Capacity"
scatter_files: []
run_MSD_plot: False
```

multi_config.yaml

```
elements:
  - ["AlCu", "CuZr"]

potentials:
  AlCu: "openKIM:EAM_Dynamo_CaiYe_1996_AlCu__MO_942551040047_005"
  CuZr: "openKIM:EAM_Dynamo_BorovikovMendelevKing_2016_CuZr__MO_097471813275_000"
  default: "LJ"

temperatures:
  AlCu: 17
  default: 600

cells:
  CuZr: "BCC"
  default: "FCC"

scaled_positions:
  AlCu: [[0, 0, 0], [0.17, 0.17, 0.17]]
  default: [[0, 0, 0], [0.5, 0.5, 0.5]]
```

With these input files `ale multi` will read the `multi_config.yaml` and create as many simulations as there are entries in the `elements` list and substitute the fields in the `base_config.yaml` with the fields specified in the `multi_config.yaml`. This allows the user to specify certain configurations for certain simulations and have a default setting in other cases to ease the configuration of a large number of simulations. The user can also define default values by specifying a value in the corresponding field in the `base_config.yaml`.

The fields map as follows:

multi_config	base_config
elements	element
temperatures	temperature_K
cells	cell
scaled_positions	scaled_positions

These two files will therefore create two simulations when run with `ale multi`. One with an aluminium and copper alloy at 17 K set in an FCC bravais lattice with the aluminium atoms placed in the origin of the unit cell and repeated from there and copper atoms shifted inwards in the cell and and repeated in an FCC bravais lattice from there. The other simulation will be copper and zirconium placed in two BCC bravais lattices with the copper lattice beginning at the origin and the zirconium lattice being shifted a half unit cell in all directions. All of this will be simulated at 600 K.

1.4.4 Limitations

1.4.4.1 Supported unit cells

Ale only supports cubic lattices with one parameter. There are two preprogrammed bravais lattices, face centered cubic (FCC) and body centered cubic (BCC), but more importantly Ale can handle base matrices as long as it only scales with one parameter. However the simulations which are setup by a basis matrix has not been as thoroughly tested as the simulations which uses the built in FCC and BCC support.

Ale also supports multi atom simulations although these have proven to be unstable under certain starting conditions and especially when using the simple built in Lennard Jones potential which can cause super lattices to explode and crash the simulation.

1.4.4.2 Interatomic potentials

The development team behind Ale highly encourages the user to look for adequate potentials on the openKIM web site since these are designed to function for certain elements and combinations of elements, which is crucial to obtain good prediction of material properties. Ale only supports potentials labeled to function with "any" simulator or with "asap" (which is used to accelerate calculations with ale).

A word of warning is that some potentials which are labeled in the openKIM library to function with any simulator are known to crash or not function anyway for an unknown reason. These non-functioning interatomic potentials have not been cataloged by the Ale team.

Built in Lennard Jones potential

The built in Lennard Jones is a kind of back-up potential and for it to function properly it needs to have the model parameters `epsilon` and `sigma` supplied. The simulation will start without the model parameters but the fallback parameters are designed for solid argon and it's not recommended to use for anything else if the user's goal is to make predictions on material properties.

The fallback parameters for the Lennard Jones potential are:

```
atomic_number = 1 # unit charge
epsilon = 0.010323 # eV
sigma = 3.40 # Å
cutoff = 6.625 # Å
```

1.4.4.3 Lattice constant calculation

The lattice constant calculation is heavily dependent on the initial guess made in the config file (`guess_latticeconstant`). If no lattice constant is provided at all there is a fallback guess of 4 Å but ale only checks an interval of guess +- 15%*guess. This means that if the real or expected lattice constant lies above 4.6 Å or below 3.4 Å a guess should be provided or undefined behavior might occur in the simulation.

The quality of the calculated lattice constant also improves as the distance between the `guess_latticeconstant` and the real/expected lattice constant decreases. This includes the case where the real/expected lattice constant is included in the interval created from the guessed lattice constant.

1.4.4.4 Equilibrium check

The built in equilibrium check has only proven effective for sufficiently large systems (around 1000 atoms or a size of 10 for an FCC lattice). A smaller number of atoms tends to generate more oscillative behavior in the temperature and energy, which is used to calculate if the system has reached equilibrium.

If the system doesn't reach equilibrium within a certain time, depending on the size of the system and the write-to-trajectory-file-interval (`interval:` in the config file), the simulation will continue anyway and start producing the output file in case the system reaches equilibrium later or if the system has reached equilibrium but it hasn't been detected by Ale.

The output `<element(s)>.json` file contains information on the equilibrium check, if the system reached equilibrium and how long it took to reach equilibrium or for the equilibrium check to timeout.

1.4.5 Output

1.4.5.1 Output from `ale`

The simulation output consists of three files (which can be placed in a directory if specified by the user), one called `raw<element(s)>.traj` (if the equilibrium check is enabled) which stores the entire simulation from start to finnish, on file named `<element(s)>.traj` which stores the simulation from equilibrium and onwards and this file is what `ale analyze` uses to calculate the output properties (specified in the config file) stored in the last output file named `<element(s)>.json`. The `<element(s)>.json` file also includes a note on the equilibrium check (if enabled) which is a flag indicating if the system reached equilibrium and how long it took to do so or how long it took for the check to reach timeout.

The properties .json-file is a json line formatted file (i.e. one JSON object per line) which enables large data sets to be handled since the computer only reads or writes a single line to the file at a time rather than the entire file. The unit on each calculated property is specified in the entry name of each property.

1.4.5.2 Output from `ale multi`

The naming of the output files (of each simulation) is the same for `ale multi` as for the normal mode in `ale` but `ale multi` requires the user to specify an output directory to enable organizing the output files and later specify the entire directory to ale visualize, when creating scatter plots.

1.4.6 ale package

1.4.6.1 Subpackages

ale.plotting package

Submodules

ale.plotting.generic_plotter module

ale.plotting.generic_plotter.**make_generic_time_plotter**(*retrieve_data*, *label*, *dt*, *time_unit=None*, *title=None*, *unit=None*)

Factory function for creating plotters that can plot data over time. The function returns a function which can be called whenever the plot should be drawn. This function takes no arguments and will create a new figure and plot the given data when called. This function doesn't call plt.show() so this must be done by the calling code.

Parameters

- **retrive_data** – function that returns data to plot over time when called with no arguments.
- **label** (*str*) – Label representing the data.
- **dt** (*number*) – delta time between time steps in data.
- **time_unit** (*str*) – unit of time, e.g. 'fs'.
- **title** (*str*) – title of plot.
- **unit** (*str*) – unit of data, e.g. 'K'.

Module contents

1.4.6.2 ale.MSD module

ale.MSD.**MSD**(*t*, *atom_list*)
The MSD(t,atom:list) function calculates and returns the mean square displacement for one time t. The function takes two arguments, the time t and atom_list which is a list of atom objects from .traj file. The time t is at what timestep of the simulation that the MSD is wanted to be calculated

ale.MSD.**lindemann_criterion**(*atom_list*)
Checks if melting has occurred. The functions takes a list of atoms at different time steps. The lindemann criterion states that melting happens when the the root mean vibration exceeds 10% of the nearest neighbor (NN) distance. The function checks this condition by calling MSD() and returns True if the condition is met

ale.MSD.**make_MSD_plotter**(*data*, *dt*)
The make_MSD_plotter function takes in data dictionary from .json file, takes the MSD data out from it and returns a plotter that can plot MSD over time

ale.MSD.**self_diffusion_coefficient**(*options*, *atom_list*)
The self_diffusion_coefficient(atom_list) function calculates and returns the self diffusion coefficient. The function takes an atom_list at different time steps which it sends to the MSD(t,atom_list) function to retrieve the MSD. It also takes options to convert time_step to seconds (time_step length may vary). The lindemann_critertion() first checks if the element is a solid or liquid. For solids we approximate the self_diffusion_coefficient as 0 and for liquids the self diffusion coefficient is taken as the slope of the mean-square-displacement. Self diffusion coefficient is returned in units m^2/s.

1.4.6.3 ale.analyze module

ale.analyze.**MSD_data_calc**(*traj_read*)
Calculates all MSD for all timesteps registered in the .traj file

ale.analyze.**output_properties_to_file**(*options*, *traj*)
Outputs the chosen properties from a traj file to json-file.

ale.analyze.**run_analysis**(*options*)
The function run_analysis takes options as arguments where options are the options for analyzing the simulated material. It is specified in config file exactly what the user wants to calculate

ale.analyze.**self_diffusion_coefficient_calc**(*options*, *traj_read*)
Calculates diffusion coefficients for all timesteps registered in the .traj file

1.4.6.4 ale.atomic_masses module

ale.atomic_masses.**atomic_masses**(*atoms_object*)
Returns the sum of the atomic masses for the molecule/elements specified by symbol in the config-file. It however does this by looking at the atoms_object and retrieving one molecule from one point in the lattice, i.e. it does not look in the config_file. This requires the set_tag function of Atoms() to save the size of the super cell when the atoms object is created by create_atoms()

1.4.6.5 ale.bulk_modulus module

ale.bulk_modulus.**calc_lattice_constant**(*options*)
This function takes a traj-file containing atom-objects with varying lattice constants, calculates the Bulk modulus B, optimal volume v0 and optimal lattice constant a0 through equation of state.

ale.bulk_modulus.**create_lattice_traj**(*options*)
This function creates atoms objects with varying lattice constants from a guessed value, a, of the chosen element. Using the linspace function to choose a range and number of variations created, and saves them all in a traj file.

ale.bulk_modulus.**read_cell**(*options*)

ale.bulk_modulus.**read_lattice_constant_or_calculate**(*options*)

1.4.6.6 ale.cohesive_energy module

ale.cohesive_energy.**cohesive_energy**(*options*, *atoms*, *iterations*, *file_output_path*)
The cohesive_energy function takes argument atoms, that is the simulated material, and makes another simulation for the material at temperature 0 K. It then extracts the potential energy which is the cohesive energy for the material

ale.cohesive_energy.**retrieve_cohesive_energy**(*traj_file*)
This function reads a traj file containing a cohesive energy simulation created by the function cohesive_energy. It reads the result of the simulation from this file and returns the cohesive energy. If the traj file can't be read it returns None.

1.4.6.7 ale.command_line_arg_parser module

class ale.command_line_arg_parser.**CreateParser**(*default*, *multi*, *simulate*, *analyze*, *visualize*)
Bases: `object`

Class thar parses the command line arguments given to ale. It handles sub commands that corresponds to different sub domains of the ale software. This is realized with subparsers for each sub command.

Parameters

- **default** (*lambda or function.*) – function to call when no subcommand is specified.
- **multi** (*lambda or function.*) – function to call when multi subcommand is specified.
- **simulate** (*lambda or function.*) – function to call when simulate subcommand is specified.
- **analyze** (*lambda or function.*) – function to call when analyze subcommand is specified.
- **visualize** (*lambda or function.*) – function to call when visualize subcommand is specified.

parse_args(*args=None*)
This function runs the parser. If no arguments to parse are specified it reads the arguments from the command line.

Parameters **args** (*list*) – args to parse

1.4.6.8 ale.create_atoms module

ale.create_atoms.**create_atoms**(*options*)
create_atoms(options) takes the argument options which is the key to read from the configuration files. Parameters are loaded from the config.yaml file and the function then returns an Atoms/lattice object with the chosen parameters that can be used for simulations.

1.4.6.9 ale.create_potential module

ale.create_potential.**built_in_LennardJones**(*options*)
Returns the built in Lennard-Jones potential. It will try to use ASAP if user didn't specify not to.

To be able to create a good potential some values need to present in the config. If not present fallback values will be used instead.

ale.create_potential.**create_potential**(*options*)
Creates potential from options passed by user in configuration. A built in Lennard-Jones potential or openKIM potential is supported.

Will raise *ale.errors.ConfigError* if it fails to create a potential from the configuration.

1.4.6.10 ale.debye_temperature module

ale.debye_temperature.**debye_temperature**(*atoms_object*, *options*, *bulk_modulus*)
debyeTemperature(atoms_object) takes two argument, an atoms objects and options (a config-file). It calculates and returns the Debye temperature by calling effectiveVelocity atomic_masses and density

1.4.6.11 ale.density module

ale.density.**density**(*atoms_object*)
The function 'density()' takes a time t, a list of atoms from .traj file and config options as argument and calculates the density of the chosen time step t. Prints and returns density in g/cm^3

ale.density.**density_plot**(*time*, *atom_list*, *options*)
The function 'density_plot()' takes a amount of timesteps, a list of atoms from a .traj file amd options from config file and plots the density over time. This can be used if the Volume is not constant over time.

1.4.6.12 ale.effective_velocity module

ale.effective_velocity.**effective_velocity**(*atoms_object*, *options*, *bulk_modulus*)
effective-Velocity calculates and returns the effective velocity by calling transversa_sound_wave_velocity and longitudinal_sound_wave_velocity.

ale.effective_velocity.**longitudinal_sound_wave_velocity**(*atoms_object*, *options*, *bulk_modulus*)
longitudinal_sound_wave_velocity takes two arguments, an atoms objects and options (config-file). It calculates and returns the longitudinal velocity of a sound wave by calling bulk_modulus shear_modulus and density

ale.effective_velocity.**transversal_sound_wave_velocity**(*atoms_object*, *options*)
transversa_sound_wave_velocity takes two arguments, an atoms objects and options (config-file). It calculates and returns the transversal velocity of a sound wave by calling shear_modulus and density

1.4.6.13 ale.equilibrium_condition module

ale.equilibrium_condition.**equilibrium_check**(*atomsTraj*, *numberOfAtoms*, *ensamble*, *checkInterval*)
This function determines if the system has reached an equilibrium which then determines if the simulation has reached equilibrium. This checks the temperature if the NVE ensamble is used and the energy if the NVT ensamble is used.

The condition is to have a low variance in the temperature/energy in the last batch of iterations which will depend on the size of the system.

1.4.6.14 ale.errors module

exception ale.errors.**ConfigError**(*config_properties*, *message*)
Bases: Exception

Used to raise errors when issues with the configuration file is encountered

Parameters

- **config_properties** (*list of str*) – properties involved in error
- **message** (*str*) – message describing the error

1.4.6.15 ale.main module

ale.main.**analyze**(*options*, *args=None*)
This function runs analysis on simulation data. What analysis and what to output is specified in the options object.

ale.main.**default**(*options*, *args=None*)
This function is run when no subcommand is passed to ale, e.g. 'ale -c my_config.yaml' It will run simulation followed by analysis.

ale.main.**multi**(*options*, *args*)
This function runs multiple simulations and analyzations in parallel on multiple cores. For this it needs a multi-config-file and a directory where to store the resulting files.

ale.main.**run**(*arguments=None*)
This function is the main entrypoint for ale. It takes an optional argument which if passed makes the parser read those arguments instead of argv from the command line.

This function parses the contents of the given config file and passes these options to the functions which actually do calculations or visualizations.

The argument parser takes a flag which enables or disables the use of asap on the current run with the flags '–asap' for enabling it and '–no-asap' to disable it.

Passing this flag is to avoid getting the error 'illegal instruction (core dumped)' in the terminal since some machines cannot run the current version of ASAP which is used in this project.

ale.main.**simulate**(*options*, *args=None*)
This function runs a molecular dynamics simulation using the options specified in an options object.

ale.main.**visualize**(*options*, *args=None*)
This function creates visualizations of the simulated and analyzed data. What visualizations to create is specified in the options object.

1.4.6.16 ale.md_config_reader module

ale.md_config_reader.**parse_config**(*config_file*)
Parses a config file written in the YAML-format.

1.4.6.17 ale.multi module

ale.multi.**generate_options_list**(*multi_config*, *options*)
This function generates a list of options objects derived from the multi config and the options object. It uses the options object as a template and replaces certain properties in the options object.

This way many new options objects are generated which can vary certain properties.

ale.multi.**get_combinations_of_elements**(*elements*)
Creates combination of the given elements. E.g. [[Na,Ca],[Cl]] gives ((Na,Cl), (Ca,Cl))

ale.multi.**multi**(*multi_config*, *options*)
This function runs many simulations and analyses in parallel on different processes. It takes a multi_config object which specifies how to generate configurations for each simulation to run.

Parameters

- **multi_config** – object specifying how to generate many simulations.
- **options** – template options object to start from when generating new options.

ale.multi.**options_from_element_combination**(*element_combination*, *multi_config*, *template_options*)
This function returns a new options object for a certain element combination. It uses the element combination and the multi config to derive what other properties to update.

Parameters

- **element_combination** – tuple representing the material to generate options object for.
- **multi_config** – object containing configurations for how to vary this options object.
- **template_options** – options object to start from when creating the new options object for this element combination.

ale.multi.**run_in_parallell**(*options_list*)
This function pickles the options list to file and starts multiple processes using openmpi to run simulations for each options object in the list.

The pickle file is hardcoded to be named 'options_pickle' and the script it calls for the multiprocessing is hard-coded to have the path 'ale/parallel_mpi_script.py'.

The function waits until all processes are finished before it returns.

ale.multi.**serialize_element_combination**(*element_combination*)
This function is used to create a string representation from an element combination which is a tuple of strings. This new representation could be used as keys in dictionaries.

This is also later passed as the symbol parameter to ase when creating an atoms object.

Parameters **element_combination** – arbitrary length tuple of strings which represents a material.

1.4.6.18 ale.parallel_mpi_script module

ale.parallel_mpi_script.**analyze**(*options*)

ale.parallel_mpi_script.**do_work**(*options*)
The work that one process should do, i.e. simulate and analyze with its assigned options. If the process has many simulations to perform it will call this function many times in sequence.

If an issue arises it will catch those exceptions and continue with the next computation.

ale.parallel_mpi_script.**get_symbol**(*options*)

ale.parallel_mpi_script.**simulate**(*options*)

1.4.6.19 ale.pressure module

ale.pressure.**avg_pressure**(*traj*)
Calculates average pressure of atoms over time.

ale.pressure.**pressure**(*atoms_object*)
Calculates instant pressure of atoms.

ale.pressure.**printpressure**(*atoms_object*)
Function to calculate and print the instant pressure for every timestep.

1.4.6.20 ale.scatter module

ale.scatter.**find_json_files**(*options*)

ale.scatter.**make_scatter_plotter**(*options*, *data_type1*, *data_type2*, *filelist=[]*)
Creates and returns a plotter function that creates a scatter plot

Parameters

- **options** – parsed options from config file.
- **data_type1** – data type to write on the x-axis.
- **data_type2** – data type to write on the y-axis.
- **filelist** – list of .json files, if specification of which files to do plotter of is needed, default=[].

1.4.6.21 ale.shear_modulus module

ale.shear_modulus.**shear_modulus**(*options*)
Shear_modulus takes one argument, options (a config-file), and returns the shear modulus for the element/molecule defined by the config-file.

1.4.6.22 ale.simulate module

ale.simulate.**MD**(*options*)
The function 'MD()' runs defines the ASE and ASAP environment to run the molecular dynamics simulation with. The elements and configuration to run the MD simulation is defined in the 'config.yaml' file which needs to be present in the same directory as the MD program (the 'main.py' file).

ale.simulate.**run_simulation**(*options*)
The 'run_simulation()' function runs the 'MD()' function which runs the simulation. 'run_simulation()' also prints out the density or other properties of the material at hand (which is to be implemented in future versions of this program, as of only density exists). What to print out during the run is defined in the 'config.yaml' file.

ale.simulate.**try_to_run_to_equilibrium**(*options*, *raw_trajectory_file_path*, *dyn*, *atoms*, *interval*, *number_of_atoms*)
This function tries to run the simulation until equilibrium is obtained. It writes this data to a raw traj-file.

1.4.6.23 ale.simulation_data_IO module

ale.simulation_data_IO.**input_simulation_data**(*out_file_path*)
This function returns data found in file at out_file_path. If this file can't be read it will return None.

Parameters **out_file_path** (*str*) – file path from which to read data.

ale.simulation_data_IO.**output_generic_from_traj**(*traj*, *out_file*, *name*, *f*)
Creates and returns outputter function that dumps some data about atoms to json.

The resulting file will be in the JSON Lines format, i.e. one JSON-document on each line.

Parameters

- **traj** – traj object containing atoms objects.
- **outfile** – open file handle that the data should be appended to.
- **name** (*str*) – name of data, this will be the name of the json-field.
- **f** (*function*) – lambda used to extract some data from an atoms object.

ale.simulation_data_IO.**output_generic_result_lazily**(*out_file*, *name*, *retrieve_result*)
This function is used to output data to file. It doesn't do this straight away but instead returns a function which can be called when the data actually should be written.

When the returned function is called a complete JSON-document containing data from retrieve_result will be written as one line to the JSON-file.

Parameters

- **out_file** – file handle to write data to.
- **name** – name to store data under in file.
- **retrieve_result** – function that returns data to write. This should take no arguments and is only called when calling the returned function.

ale.simulation_data_IO.**output_single_property**(*out_file*, *name*, *value*)
Returns a function which writes a single value to a JSON-file.

When the returned function is called a complete JSON-document containing the value will be written as one line to the JSON-file.

The resulting file will be in the JSON Lines format, i.e. one JSON-document on each line.

Parameters

- **out_file** – file handle to write data to.
- **name** (*str*) – key that value should be stored under in file.
- **value** – value to write to file.

1.4.6.24 ale.specific_heat_capacity module

ale.specific_heat_capacity.**specific_heat_capacity**(*ensemble*, *traj*)
This function calculates the specific heat capacity from the trajectory output file and the ensemble that has been simulated.

1.4.6.25 ale.utils module

class ale.utils.**EoSResults**(*options*)
Bases: object

EoSResults stores results from EoS equations lazily. The class can be instantiated without running the costly computation but when someone tries to access a value, the computation will be run if it hasn't been run yet.

get_bulk_modulus()

get_bulk_optimal_lattice_volume()

get_optimal_lattice_constant()

ale.utils.**call_only_once**(*f*)
Wrapper that makes sure a function is called only once. This is useful for example if it's known that a very expensive function is run many times but with the same parameters and return value. It's not memoization since this wrapper doesn't take the paramters into consideration. The parameters used on the first call dictates what value is stored and returned upon all future calls.

This is thus not the same as memoization where the cache takes parameter values into consideration. The reason this was used instead of memoization was that the parameters to a target function weren't hashable and able to be used in memoization.

1.4.6.26 ale.visualize_main module

ale.visualize_main.**make_lattice_viewer**(*options*)
Creates viewer to visualize atoms with ase gui visualizer.

It visualizes the initial positions of the atoms in the lattice.

ale.visualize_main.**visualize**(*options*, *data_file_path*)
This function visualizes properties specified by the user in the config-file.

1.4.6.27 Module contents

PYTHON MODULE INDEX

a

INDEX

E

F

G

I

L

M

O

P

R

S

T

V