Statdyn Analysis Documentation

Release 0.11.6

Malcolm Ramsay

Jul 30, 2020

CONTENTS

CHAPTER

ONE

DYNAMICS MODULE

Module for reading and processing input files.

class sdanalysis.dynamics.**Dynamics**(*timestep*, *box*, *position*, *orientation=None*, *molecule=None*, *wave_number=None*, *angular_resolution=360*)

Bases: object

Compute dynamic properties of a simulation.

Parameters

- **timestep** (int) – The timestep on which the configuration was taken.
- **box** (ndarray) – The lengths of each side of the simulation cell including any tilt factors.
- **position** (ndarray) – The positions of the molecules with shape (nmols, 3). Even if the simulation is only 2D, all 3 dimensions of the position need to be passed.
- **orientation** (Optional[ndarray]) – The orientations of all the molecules as a quaternion in the form (w, x, y, z). If no orientation is supplied then no rotational quantities are calculated.
- **molecule** (Optional[*Molecule*]) – The molecule for which to compute the dynamics quantities. This is used to compute the structural relaxation for all particles.
- **wave_number** (Optional[float]) – The wave number of the maximum peak in the Fourier transform of the radial distribution function. If None this is calculated from the initial configuration.

add(*position*, *orientation=None*)

Update the state of the dynamics calculations by adding a Frame.

This updates the motion of the particles, comparing the positions and orientations of the current frame with the previous frame, adding the difference to the total displacement. This approach allows for tracking particles over periodic boundaries, or through larger rotations assuming that there are sufficient frames to capture the information. Each single displacement obeys the minimum image convention, so for large time intervals it is still possible to have missing information.

Parameters

- **position** (ndarray) – The updated position of each particle
- **orientation** (Optional[ndarray]) – The updated orientation of each particle, represented as a quaternion

add_frame(*frame*)

Update the state of the dynamics calculations by adding a Frame.

This updates the motion of the particles, comparing the positions and orientations of the current frame with the previous frame, adding the difference to the total displacement. This approach allows for tracking particles over periodic boundaries, or through larger rotations assuming that there are sufficient frames to capture the information. Each single displacement obeys the minimum image convention, so for large time intervals it is still possible to have missing information.

Parameters **frame** (*Frame*) – The configuration containing the current particle information.

compute_all(*timestep*, *position*, *orientation=None*, *scattering_function=False*)

Compute all possible dynamics quantities.

Parameters

- **timestep** (int) – The current timestep of the dynamic quantity
- **position** (ndarray) – The position of all particles at the new point in time
- **orientation** (Optional[ndarray]) – The orientation (as a quaternion) of all particles

Return type Dict[str, Union[int, float]]

Returns Mapping of the names of each dynamic quantity to their values for each particle.

Where a quantity can't be calculated, an array of nan values will be supplied instead, allowing for continued compatibility.

compute_alpha()

Compute the non-Gaussian parameter alpha for translational motion in 2D.

$$\alpha = \frac{\langle \Delta r^4 \rangle}{2\langle \Delta r^2 \rangle^2} - 1$$

Return type float

compute_alpha_rot()

Compute the non-Gaussian parameter alpha for rotational motion in 2D.

Rotational motion in 2D, is a single dimension of rotational motion, hence the use of a different divisor than translational motion.

$$\alpha = \frac{\langle \Delta \theta^4 \rangle}{3\langle \Delta \theta^2 \rangle^2} - 1$$

Return type float

compute_displacement()

Compute the translational displacement for each particle.

Return type ndarray

compute_displacement2()

Compute the squared displacement for each particle.

Return type ndarray

compute_gamma()

Calculate the second order coupling of translations and rotations.

$$\gamma = \frac{\langle (\Delta r \Delta \theta)^2 \rangle}{\langle \Delta r^2 \rangle \langle \Delta \theta^2 \rangle} - 1$$

Returns The squared coupling of translations and rotations γ

Return type float

compute_isf()
Compute the intermediate scattering function.

Return type float

compute_mean_rotation()
Compute the rotation from the initial frame.

Return type float

compute_mfd()
Compute the fourth power of displacement.

Return type float

compute_msd()
Compute the mean squared displacement.

Return type float

compute_msr()
Compute the mean squared rotation from the initial frame.

Return type float

compute_rotation()
Compute the rotational motion for each particle.

Return type ndarray

compute_rotation2()
Compute the rotation from the initial frame.

Return type ndarray

compute_rotational_relax1()
Compute the first-order rotational relaxation function.

$$C_1(t) = \langle \hat{\mathbf{e}}(0) \cdot \hat{\mathbf{e}}(t) \rangle$$

Returns The rotational relaxation

Return type float

compute_rotational_relax2()
Compute the second rotational relaxation function.

$$C_1(t) = \langle 2(\hat{\mathbf{e}}(0) \cdot \hat{\mathbf{e}}(t))^2 - 1 \rangle$$

Returns The rotational relaxation

Return type float

compute_struct_relax()

Return type float

compute_time_delta(*timestep*)
Time difference between initial frame and timestep.

Return type int

property delta_rotation

property delta_translation

property distance

Return type Optional[float]

classmethod from_frame(*frame*, *molecule=None*, *wave_number=None*)
Initialise the Dynamics class from a Frame object.

There is significant overlap between the frame class and the dynamics class, so this is a convenience method to make the initialisation simpler.

Return type Dynamics

get_molid()
Molecule ids of each of the values.

class sdanalysis.dynamics.**TrackedMotion**(*box*, *position*, *orientation*)
Bases: object

Keep track of the motion of a particle allowing for multiple periods.

This keeps track of the position of a particle as each frame is added, which allows for tracking the motion of a particle through multiple periods, as long as each motion takes the shortest distance.

add(*position*, *orientation*)
Update the state of the dynamics calculations by adding the next values.

This updates the motion of the particles, comparing the positions and orientations of the current frame with the previous frame, adding the difference to the total displacement. This approach allows for tracking particles over periodic boundaries, or through larger rotations assuming that there are sufficient frames to capture the information. Each single displacement obeys the minimum image convention, so for large time intervals it is still possible to have missing information.

Parameters

- **position** (ndarray) – The current positions of the particles
- **orientation** (Optional[ndarray]) – The current orientations of the particles represented as a quaternion

class sdanalysis.dynamics.**LastMolecularRelaxation**(*num_elements*, *threshold*, *irreversibility=1.0*, *relaxation_type=None*)
Bases: sdanalysis.dynamics.relaxations.MolecularRelaxation

add(*timediff*, *distance*)

Return type None

get_status()

class sdanalysis.dynamics.**MolecularRelaxation**(*num_elements*, *threshold*, *relaxation_type=None*)
Bases: object

Compute the relaxation of each molecule.

add(*timediff*, *distance*)

Return type None

get_status()

relaxation_type = 2

class sdanalysis.dynamics.**Relaxations**(*timestep*, *box*, *position*, *orientation*, *molecule=None*, *is2D=None*, *wave_number=None*)
Bases: object

add(*timestep*, *position*, *orientation*)
Update the state of the relaxation calculations by adding a Frame.

This updates the motion of the particles, comparing the positions and orientations of the current frame with the previous frame, adding the difference to the total displacement. This approach allows for tracking particles over periodic boundaries, or through larger rotations assuming that there are sufficient frames to capture the information. Each single displacement obeys the minimum image convention, so for large time intervals it is still possible to have missing information.

Parameters

- **timestep** (int) – The timestep of the frame being added
- **position** (ndarray) – The new position of each particle in the simulation
- **orientation** (ndarray) – The updated orientation of each particle, represented as a quaternion.

Return type None

add_frame(*frame*)
Update the state of the relaxation calculations by adding a Frame.

This updates the motion of the particles, comparing the positions and orientations of the current frame with the previous frame, adding the difference to the total displacement. This approach allows for tracking particles over periodic boundaries, or through larger rotations assuming that there are sufficient frames to capture the information. Each single displacement obeys the minimum image convention, so for large time intervals it is still possible to have missing information.

Parameters **frame** (*Frame*) – The configuration containing the current particle information.

property distance

classmethod from_frame(*frame*, *molecule=None*, *wave_number=None*)
Initialise a Relaxations class from a Frame class.

This uses the properties of the Frame class to fill the values of the Relaxations class, for which there is significant overlap.

Return type Relaxations

get_timediff(*timestep*)

set_mol_relax(*definition*)

Return type None

summary()

Return type DataFrame

CHAPTER

TWO

RELAXATION MODULE

These are a series of summary values of the dynamics quantities.

This provides methods of easily comparing values across variables.

class sdanalysis.relaxation.**Result**
Bases: tuple

Hold the result of a relaxation calculation.

This uses the NamedTuple class to make the access of the returned values more transparent and easier to understand.

property error
Alias for field number 1

property mean
Alias for field number 0

sdanalysis.relaxation.**compute_molecular_relaxations**(*df*)

Return type DataFrame

sdanalysis.relaxation.**compute_relaxation_value**(*timesteps*, *values*, *relax_type*)
Compute a single representative value for each dynamic quantity.

Parameters

- **timesteps** (ndarray) – The timestep for each value of the relaxation.
- **values** (ndarray) – The values of the relaxation quantity for each time interval.
- **relax_type** (str) – A string describing the relaxation.

Return type *Result*

Returns The representative relaxation time for a quantity.

There are some special values of the relaxation which are treated in a special way. The main one of these is the "msd", for which the relaxation is fitted to a straight line. The "struct_msd" relaxation, is a threshold_relaxation, with the time required to pass the threshold of 0.16. The other relaxations which are treated separately are the "alpha" and "gamma" relaxations, where the relaxation time is the maximum of these functions.

All other relaxations are assumed to have the behaviour of exponential decay, with the representative time being how long it takes to decay to the value 1/e.

sdanalysis.relaxation.**compute_relaxations**(*infile*)
Summary time value for the dynamic quantities.

This computes the characteristic timescale of the dynamic quantities which have been calculated and are present in INFILE. The INFILE is a path to the pre-computed dynamic quantities and needs to be in the HDF5 format with either the '.hdf5' or '.h5' extension.

The output is written to the table 'relaxations' in INFILE.

Return type None

sdanalysis.relaxation.**diffusion_constant**(*time*, *msd*, *dimensions=2*)
Compute the diffusion_constant from the mean squared displacement.

Parameters

- **time** (ndarray) – The timesteps corresponding to each msd value.
- **msd** (ndarray) – Values of the mean squared displacement

Returns The diffusion constant error (float): The error in the fit of the diffusion constant

Return type diffusion_constant (float)

sdanalysis.relaxation.**exponential_relaxation**(*time*, *value*, *sigma=None*, *value_width=0.3*)
Fit a region of the exponential relaxation with an exponential.

This fits an exponential to the small region around the value 1/e. A small region is chosen as the interest here is the time for the decay to reach a value, rather than a fit to the overall curve, so this provides a method of getting an accurate time, while including a collection of points.

Parameters

- **time** (ndarray) – The timesteps corresponding to each value
- **value** (ndarray) – The value at each point in time
- **sigma** (Optional[ndarray]) – The uncertainty associated with each point
- **value_width** (float) – The width of values over which the fit takes place

Returns The relaxation time for the given quantity error (float): Estimated error of the relaxation time.

Return type relaxation_time (float)

sdanalysis.relaxation.**max_time_relaxation**(*time*, *value*)
Time at which the maximum value is recorded.

Parameters

- **time** (ndarray) – The time index
- **value** (ndarray) – The value at each of the time indices

Returns The time at which the maximum value occurs. float: Estimate of the error of the time

Return type float

sdanalysis.relaxation.**max_value_relaxation**(*time*, *value*)
Maximum value recorded.

Parameters

- **time** (ndarray) – The time index
- **value** (ndarray) – The value at each of the time indices

Returns The value at which the maximum value occurs. float: Estimate of the error in the maximum value.

Return type float

sdanalysis.relaxation.**series_relaxation_value**(*series*)

Calculate the relaxation of a pandas Series.

When a *pandas.Series* object, which has an index being the timesteps, and the name of the series being the dynamic quantity, this function provides a simple method of calculating the relaxation aggregation. In particular this function is useful to use with the aggregate function.

Parameters **series** (Series) – The series containing the relaxation quantities

Return type float

Returns The calculated value of the relaxation.

sdanalysis.relaxation.**threshold_relaxation**(*time*, *value*, *threshold=0.36787944117144233*, *decay=True*)

Compute the relaxation through the reaching of a specific value.

Parameters

- **time** (ndarray) – The timesteps corresponding to each msd value.
- **value** (ndarray) – Values of the relaxation parameter

Returns The relaxation time for the given quantity. error (float): The error in the fit of the relaxation

Return type relaxation time (float)

sdanalysis.relaxation.**translate_relaxation**(*quantity*)

Convert names of dynamic quantities to their relaxations.

Parameters **quantity** (str) – The name of the quantity to convert the name of.

Return type str

Returns The translated name.

CHAPTER

THREE

PARAMS MODULE

Parameters for passing between functions.

class sdanalysis.params.**SimulationParams**(*temperature=0.4*, *pressure=13.5*, *molecule=Trimer(radius=0.637556*, *distance=1.0*, *angle=120*, *moment_inertia_scale=1.0)*, *moment_inertia_scale=None*, *harmonic_force=None*, *wave_number=None*, *space_group=None*, *num_steps=None*, *linear_steps=100*, *max_gen=500*, *gen_steps=20000*, *output_interval=10000*, *infile=None*, *outfile=None*, *output=None*)

Bases: object

Store the parameters of the simulation.

filename(*prefix=None*)

Use the simulation parameters to construct a filename.

Return type Path

property infile

Return type Optional[Path]

property outfile

Return type Optional[Path]

property output

Return type Path

temp_context(***kwargs*)

CHAPTER

FOUR

FIGURES MODULE

Plot configuration.

sdanalysis.figures.configuration.**colour_orientation**(*orientations*, *light_colours=False*)

Return type ndarray

sdanalysis.figures.configuration.**frame2data**(*frame*, *order_function=None*, *order_list=None*, *molecule=Trimer(radius=0.637556, distance=1.0, angle=120, moment_inertia_scale=1.0)*, *categorical_colour=False*)

Convert a Frame to data for plotting in Bokeh.

This takes a frame and performs all the necessary calculations for plotting, in particular the colouring of the orientation and crystal classification.

Parameters

- **frame** (*Frame*) – The configuration which is to be plotted.
- **order_function** (Optional[Callable[[*Frame*], ndarray]]) – A function which takes a frame as it's input which can be used to classify the crystal.
- **order_list** (Optional[ndarray]) – A pre-classified collection of values. This is an alternate approach to using the order_function
- **molecule** (*Molecule*) – The molecule which is being plotted.
- **categorical_colour** (bool) – Whether to classify as categories, or liquid/crystalline.

Return type Dict[str, Any]

Returns

Dictionary containing x, y, colour, orientation and radius values for each molecule.

sdanalysis.figures.configuration.**plot_circles**(*mol_plot*, *source*, *categorical_colour=False*, *factors=None*, *colormap=('#1f77b4', '#ff7f0e', '#2ca02c', '#d62728', '#9467bd', '#8c564b', '#e377c2', '#7f7f7f', '#bcbd22', '#17becf')*)

Add the points to a bokeh figure to render the trimer molecule.

This enables the trimer molecules to be drawn on the figure using only the position and the orientations of the central molecule.

Return type figure

sdanalysis.figures.configuration.**plot_frame**(*frame*, *order_function=None*, *order_list=None*, *source=None*, *molecule=Trimer(radius=0.637556, distance=1.0, angle=120, moment_inertia_scale=1.0)*, *categorical_colour=False*, *factors=None*, *colormap=None*)

Plot a snapshot using bokeh.

Parameters

- **frame** (*Frame*) – The frame determining the positions to plot
- **order_function** (Optional[Callable[[*Frame*], ndarray]]) – A function which takes a frame and determines ordering
- **order_list** (Optional[ndarray]) – A pre-computed list of ordering.
- **source** (Optional[ColumnDataSource]) – An existing bokeh ColumnDataSource to use for plotting.
- **molecule** (*Molecule*) – The molecule which is being plotted, used to calculate additional positions.
- **categorical_colour** (bool) – Toggle which colours liquid/crystal, or each crystal
- **factors** (Optional[List[Any]]) – The factors used for plotting. This is for continuity across a range of figures.
- **colourmap** – The collection of colours to use when plotting.

Returns Bokeh plot

Bokeh dashboard for interactive visualisation of thermodynamic properties.

sdanalysis.figures.thermodynamics.**read_file**(*source*)
Read a file into a pandas dataframe.

sdanalysis.figures.thermodynamics.**update_datacolumns**(*attr*, *old*, *new*)
Update data as a callback.

sdanalysis.figures.thermodynamics.**update_factors**(*attr*, *old*, *new*)
Update factors as a callback.

sdanalysis.figures.thermodynamics.**update_file**(*attr*, *old*, *new*)
Update current file as a callback.

sdanalysis.figures.thermodynamics.**update_file_list**(*attr*, *old*, *new*)
Update list of all files as a callback.

CHAPTER

FIVE

FRAME MODULE

Classes which hold frames.

class sdanalysis.frame.**Frame**

Bases: abc.ABC

abstract property box

Return type ndarray

abstract property dimensions

Return type int

freud_box()

Return type Box

abstract property image

Return type Optional[ndarray]

abstract property orientation

Return type ndarray

abstract property position

Return type ndarray

abstract property timestep

Return type int

abstract property x_position

Return type ndarray

abstract property y_position

Return type ndarray

abstract property z_position

Return type ndarray

class sdanalysis.frame.**HoomdFrame**(*frame*)

Bases: *sdanalysis.frame.Frame*

property box

Return type ndarray

property dimensions

Return type int

property image

Return type Optional[ndarray]

property num_mols

property orientation

Return type ndarray

property position

Return type ndarray

property timestep

Return type int

property x_position

Return type ndarray

property y_position

Return type ndarray

property z_position

Return type ndarray

class sdanalysis.frame.**LammpsFrame**(*frame*)

Bases: *sdanalysis.frame.Frame*

property box

Return type ndarray

property dimensions

Return type int

property image

Return type Optional[ndarray]

property orientation

Return type ndarray

property position

Return type ndarray

property timestep

Return type int

property x_position

Return type ndarray

property y_position

Return type ndarray

property z_position

Return type ndarray

CHAPTER

SIX

UTIL MODULE

A collection of utility functions.

class sdanalysis.util.**Variables**(*temperature*, *pressure*, *crystal*, *iteration_id*)

Bases: tuple

property crystal

Alias for field number 2

classmethod from_filename(*fname*)

Create a Variables instance taking information from a path.

This extracts the information about the value of variables used within a simulation trajectory from the filename. This is expecting the information in a specific format, where values are separated by the dash character -.

Parameters **fname** (Union[str, Path]) – The full path of the filename from which to extract the information.

Warning: This is expecting the full name of the file, including the extension. Should there not be an extension on the filename, values could be stripped giving undefined behaviour.

Return type *Variables*

property iteration_id

Alias for field number 3

property pressure

Alias for field number 1

property temperature

Alias for field number 0

sdanalysis.util.**create_freud_box**(*box*, *is_2D=True*)

Convert an array of box values to a box for use with freud functions

The freud package has a special type for the description of the simulation cell, the Box class. This is a function to take an array of lengths and tilts to simplify the creation of the Box class for use with freud.

Return type Box

sdanalysis.util.**get_filename_vars**(*fname*)

Extract variables information from a filename.

This extracts the information about the value of variables used within a simulation trajectory from the filename. This is expecting the information in a specific format, where values are separated by the dash character -.

Parameters **fname** (Union[str, Path]) – The full path of the filename from which to extract the information.

Warning: This is expecting the full name of the file, including the extension. Should there not be an extension on the filename, values could be stripped giving undefined behaviour.

Return type *Variables*

sdanalysis.util.**orientation2positions**(*mol*, *position*, *orientation*)

Return type ndarray

sdanalysis.util.**parse_directory**(*directory*, *glob*)

Return type Dict[str, Dict]

sdanalysis.util.**quaternion2z**(*quaternion*)
Convert a rotation about the z axis to a quaternion.

This is a helper for 2D simulations, taking the rotation of a particle about the z axis and converting it to a quaternion. The input angle *theta* is assumed to be in radians.

Return type ndarray

sdanalysis.util.**quaternion_angle**(*quaternion*)

Return type ndarray

sdanalysis.util.**quaternion_rotation**(*initial*, *final*)

sdanalysis.util.**rotate_vectors**(*quaternion*, *vector*)

sdanalysis.util.**set_filename_vars**(*fname*, *sim_params*)
Set the variables of the simulations params according to the filename.

Return type None

sdanalysis.util.**z2quaternion**(*theta*)
Convert a rotation about the z axis to a quaternion.

This is a helper for 2D simulations, taking the rotation of a particle about the z axis and converting it to a quaternion. The input angle *theta* is assumed to be in radians.

Return type ndarray

sdanalysis.util.**zero_quaternion**(*num*)

CHAPTER

SEVEN

READ MODULE

Module for reading and processing input files.

sdanalysis.read.**read_gsd_trajectory**(*infile*, *steps_max=None*, *linear_steps=100*, *keyframe_interval=1000000*, *keyframe_max=500*)

Perform analysis of a GSD file.

Return type Iterator[Tuple[List[int], *Frame*]]

sdanalysis.read.**read_lammps_trajectory**(*infile*, *steps_max=None*)

Return type Iterator[Tuple[List[int], *LammpsFrame*]]

sdanalysis.read.**open_trajectory**(*filename*, *progressbar=None*, *frame_interval=1*)

Open a simulation trajectory for processing.

This reads each configuration in turn from the trajectory, handling most of the common errors with reading a file.

This handles trajectories in both the gsd file format and simple lammpstrj files.

Parameters

- **filename** (Path) – The path to the file which is to be opened.
- **progressbar** – Whether to display a progress bar when reading the file.

Returns *Frame* objects.

Return type Generator which returns class

sdanalysis.read.**process_file**(*infile*, *wave_number*, *steps_max=None*, *linear_steps=None*, *keyframe_interval=1000000*, *keyframe_max=500*, *mol_relaxations=None*, *outfile=None*, *scattering_function=False*)

Read a file and compute the dynamics quantities.

This computes the dynamic quantities from a file returning the result as a pandas DataFrame. This is only suitable for cases where all the data will fit in memory, as there is no writing to a file.

Args:

Returns DataFrame with the dynamics quantities.

Return type (py:class:*pandas.DataFrame*)

CHAPTER

EIGHT

MOLECULE MODULE

Module to define a molecule to use for simulation.

class sdanalysis.molecules.**Dimer**(*radius=0.637556*, *distance=1.0*, *moment_inertia_scale=1*)
Bases: *sdanalysis.molecules.Molecule*

Defines a Dimer molecule for initialisation within a hoomd context.

This defines a molecule of three particles, shaped somewhat like Mickey Mouse. The central particle is of type *'A'* while the outer two particles are of type *'B'*. The type *'B'* particles, have a variable radius and are positioned at a specified distance from the central type *'A'* particle. The angle between the two type *'B'* particles, subtended by the type *'A'* particle is the other degree of freedom.

class sdanalysis.molecules.**Disc**
Bases: *sdanalysis.molecules.Molecule*

Defines a 2D particle.

class sdanalysis.molecules.**Molecule**(*dimensions=3*, *particles=NOTHING*, *positions=NOTHING*, *radii=NOTHING*, *rigid=False*, *moment_inertia_scale=1*)
Bases: object

A template class for the generation of molecules for analysis.

The positions of all molecules will be adjusted to ensure the center of mass is at the position (0, 0, 0).

get_radii()
Radii of the particles.

Return type ndarray

get_types()
Get the types of particles present in a molecule.

Return type List[str]

property num_particles
Count of particles in the molecule

Return type int

class sdanalysis.molecules.**Sphere**
Bases: *sdanalysis.molecules.Molecule*

Define a 3D sphere.

class sdanalysis.molecules.**Trimer**(*radius=0.637556*, *distance=1.0*, *angle=120*, *moment_inertia_scale=1.0*)
Bases: *sdanalysis.molecules.Molecule*

Defines a Trimer molecule for initialisation within a hoomd context.

This defines a molecule of three particles, shaped somewhat like Mickey Mouse. The central particle is of type *'A'* while the outer two particles are of type *'B'*. The type *'B'* particles, have a variable radius and are positioned at a specified distance from the central type *'A'* particle. The angle between the two type *'B'* particles, subtended by the type *'A'* particle is the other degree of freedom.

property rad_angle

Return type float

CHAPTER

NINE

ORDER MODULE

Module for the computation of ordering.

These are tools and utilities for calculating the ordering of local structures.

sdanalysis.order.**compute_neighbours**(*box*, *position*, *max_radius=3.5*, *max_neighbours=8*)
Compute the neighbours of each molecule.

Parameters

- **box** (ndarray) – The parameters of the simulation cell
- **position** (ndarray) – The positions of each molecule
- **max_radius** (float) – The maximum radius to search for neighbours
- **max_neighbours** (int) – The maximum number of neighbours to find.

Return type ndarray

Returns An array containing the index of the neighbours of each molecule. Each molecule will have *max_neighbours* listed, with the value *2 ** 32 - 1* indicating a missing value.

sdanalysis.order.**compute_voronoi_neighs**(*box*, *position*)

Return type ndarray

sdanalysis.order.**create_ml_ordering**(*model*)
Create a machine learning initialised from a pickled model.

This reads a machine learning model from a file, creating a function to classify the ordering of a configuration.

Parameters **model** (Path) – The path to a file containing a pickled model to be loaded using joblib

Return type Callable[[*Frame*], ndarray]

Returns A function to classify the ordering within a configuration.

sdanalysis.order.**create_neigh_ordering**(*neighbours*)

Return type Callable[[*Frame*], ndarray]

sdanalysis.order.**create_orient_ordering**(*threshold*)

Return type Callable[[*Frame*], ndarray]

sdanalysis.order.**num_neighbours**(*box*, *position*, *max_radius=3.5*)
Calculate the number of neighbours of each molecule.

This function is optimised to quickly calculate the number of nearest neighbours each particle has.

Parameters

- **box** (ndarray) – The lengths of the simulation cell in each direction

- **position** (ndarray) – The position of each particle
- **max_radius** (float) – The maximum radius at which a particle is considered a neighbour.

Return type ndarray

sdanalysis.order.**orientational_order**(*box*, *position*, *orientation*, *max_radius=3.5*, *max_neighbours=8*)

Compute the orientational order parameter for a given input.

The orientational order parameter compares the orientation of a particle with that of all it's neighbours, using the relation

..math:

Theta = sum_{i=1}^N cos^2((theta_i - theta))

taking the orientation of each of the neighbouring particles compared to the current particle. The square ensures that the angles which are both parallel and antiparallel contribute to the ordering.

Parameters

- **box** (ndarray) – The lengths of the simulation cell in each direction
- **position** (ndarray) – The position of each particle
- **orientation** (ndarray) – The orientation of each particle, given as quaternions.
- **max_radius** (float) – The maximum radius to search for neighbours
- **max_neighbours** (int) – The maximum number of neighbours to search for

Return type ndarray

sdanalysis.order.**relative_distances**(*box*, *position*, *max_radius=3.5*, *max_neighbours=8*)

Compute the distance to each neighbour.

Parameters

- **box** (ndarray) – The lengths of the simulation cell in each direction
- **position** (ndarray) – The position of each particle
- **max_radius** (float) – The maximum radius at which a particle is considered a neighbour.
- **max_neighbours** (int) – The maximum number of neighbours to search for

Return type ndarray

Returns The distance to each neighbour in a numpy array. Values which correspond to missing neighbours are represented by the value -1.

sdanalysis.order.**relative_orientations**(*box*, *position*, *orientation*, *max_radius=3.5*, *max_neighbours=8*)

Find the relative orientations of each neighbouring particle.

This finds each of the nearest neighbours for each particle and computes the orientation of those neighbours relative to the orientation of the central particle.

Parameters

- **box** (ndarray) – The lengths of the simulation cell in each direction
- **position** (ndarray) – The position of each particle
- **orientation** (ndarray) – The orientation of each particle represented as a quaternion

- **max_radius** (float) – The maximum distance to look to nearest neighbours
- **max_neighbours** (int) – The maximum number of neighbours considered nearest.

Return type ndarray

sdanalysis.order.**setup_neighbours**(*box*, *position*, *max_radius=3.5*, *max_neighbours=8*, *is_2D=True*)

Return type NearestNeighbors

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

E

F

G

H

I

L

M

N

O

P