physical$validation$ $Documentation$

Release 1.0.5

Pascal T. Merz, Michael R. Shirts

Jul 09, 2024

USER GUIDE:

`physical_validation` is a package aimed at testing results obtained by molecular dynamics simulations for their physical validity.

Note: The physical validation methodology has been described in Merz PT, Shirts MR (2018), Testing for physical validity in molecular simulations. PLoS ONE 13(9): e0202764. https://doi.org/10.1371/journal.pone.0202764

Note: We are always looking to enlarge our set of tests. If you are a MD user or developer and have suggestions for physical validity tests missing in this package, we would love to hear from you! Please consider getting in touch with us via our github repository.

CHAPTER

ONE

INTRODUCTION

Advances in recent years have made molecular dynamics (MD) simulations a powerful tool in molecular-level research, allowing the prediction of experimental observables in the study of systems such as proteins, drug targets or membranes. The quality of any prediction based on MD results will, however, strongly depend on the validity of underlying physical assumptions.

This package is intended to help detect (sometimes hard-to-spot) unphysical behavior of simulations, which may have statistically important influence on their results. It is part of a two-fold approach to increase the robustness of molecular simulations.

First, it empowers users of MD programs to test the physical validity on their respective systems and setups. The tests range from simple post-processing analysis to more involved tests requiring additional simulations. These tests can significantly increase the reliability of MD simulations by catching a number of common simulation errors violating physical assumptions, such as non-conservative integrators, deviations from the specified Boltzmann ensemble, or lack of ergodicity between degrees of freedom. To make usage as easy as possible, parsers for the outputs of several popular MD programs are provided

Second, it can be integrated in MD code testing environments. While unphysical behavior can be due to poor or incompatible choices of parameters by the user, it can also originate in coding errors within the program. Physical validation tests can be integrated in the code-checking mechanism of MD software packages to facilitate the detection of such bugs. The `physical_validation` package is currently used in the automated code-testing facility of the GROMACS software package, ensuring that every major releases passes a number of physical sanity checks performed on selected representative systems before shipping.

Note: The physical validation tests have been described in [Merz2018].

Note: We are always looking to enlarge our set of tests. If you are a MD user or developer and have suggestions for physical validity tests missing in this package, we would love to hear from you! Please consider getting in touch with us via our github repository.

CHAPTER

TWO

INSTALLATION

2.1 pip

The most recent release of *physical_validation* can be installed from PyPI via `pip`

```
pip install physical_validation
```

2.2 conda

The most recent release of *physical_validation* can also be installed using `conda`

```
conda install -c conda-forge physical_validation
```

2.3 Development version

The latest version is available on our github repository. You can install it via `pip`

```
pip install git+https://github.com/shirtsgroup/physical_validation.git
```

CHAPTER

THREE

SIMULATION DATA

The data of simulations to be validated are represented by objects of the *SimulationData* type.

The *SimulationData* objects contain information about the simulation and the system. This information is collected in objects of different classes, namely:

- *SimulationData.units* of type *UnitData*: Information on the units used by the simulation program.
- *SimulationData.ensemble* of type *EnsembleData*: Information on the sampled ensemble. This includes the temperature, pressure, and chemical potential, with specific requirements depending on the ensemble specified.
- *SimulationData.system* of type *SystemData*: Information on the system (number of atoms, molecules, constraints, etc.).
- *SimulationData.observables* of type *ObservableData*: Trajectories of observables along the simulation, such as energy or volume.
- *SimulationData.trajectory* of type *TrajectoryData*: Position / velocity / force trajectories along the simulation.
- *SimulationData.dt* of type `float`: The time step at which the simulation was performed.

The different physical validation tests do not all require all data to be able to run. Each `physical_validation` function checks whether the required information was provided, and raises an error if the information is insufficient. *Data contained in SimulationData objects* lists by which tests the single members of *SimulationData* are required.

The *SimulationData* objects can either be constructed directly from arrays and numbers, or (partially) automatically via parsers. The preferred way to populate *SimulationData* objects is by assigning its sub-objects explicitly with data obtained from the simulation package. Many simulation packages have a well-defined Python interface which allows to read observable, position and velocity trajectories into Python data structures. The remaining information, such as details on the simulated ensemble or the molecular system, is usually rather easy to fill in by hand. The examples in this documentation follow this model.

Please see *Creation of SimulationData objects* for more details on the *SimulationData* type and on how to create objects from results obtained from different simulation packages.

CHAPTER

FOUR

KINETIC ENERGY VALIDATION

Kinetic energy validation includes testing the likelihood of a trajectory to originate from the theoretically expected gamma distribution and validating the temperature equipartition between groups of degrees of freedom. For details on the employed algorithms, please check the respective function documentations.

For both the full distribution test and the equipartition test, a strict and a non-strict version are available. They are triggered using the `strict=[True|False]` keyword. The strict version does a full distribution similarity analysis using the Kolmogorov-Smirnov (K-S) test. The K-S test returns a p-value indicating the likelihood that the sample originates from the expected distribution. Its sensitivity increases with increasing sample size, and can flag even the smallest deviations from the expected distribution at large sample sizes. When developing or implementing new temperature control algorithms in a controlled testing environment which keeps errors from other sources negligible, such a high sensibility is desirable. In other applications, however, a deviation insignificant in comparison with other sources of inaccuracies might be enough to flag long simulation trajectories of large systems as not having a gamma distribution. For example, deviations from the desired kinetic energy distribution that are smaller in magnitude than other well-controlled approximations, such as the interaction cutoff or the treatment of bond constraints, might be enough to flag large samples as not being properly distributed.

As an alternative to the strict test, the `physical_validation` suite offers the non-strict version. In this case, the mean and the standard deviation of the sample are calculated and compared to the expected values. To make the test easily interpretable, two distinct temperatures T_μ and T_σ are estimated from the kinetic energy distribution. They represent the temperature at which the sample mean and standard would be physically expected. An error estimate computed via bootstrapping of the provided kinetic energy samples is given for each of the temperatures, giving information on the statistical significance of the results.

For more details about the difference between the strict test and non-strict test, please see *physical_validation.kinetic_energy.distribution()*.

4.1 Full system distribution validation

4.1.1 Function reference

physical_validation.kinetic_energy.distribution()

4.1.2 Example

Kinetic energy distribution example

4.2 Equipartition validation

4.2.1 Function reference

physical_validation.kinetic_energy.equipartition()

4.2.2 Example

Kinetic energy equipartition example

CHAPTER

FIVE

ENSEMBLE VALIDATION

As the distribution of configurational quantities like the potential energy U, the volume V or (for the grand and semi-grand canonical ensembles) the number of each species N_i are in general not known analytically, testing the likelihood of a trajectory sampling a given ensemble is less straightforward than for the kinetic energy. However, generally, the *ratio* of the probability distribution between samplings of the same system generated at different state points (e.g. simulations run at at different temperatures or different pressures) is exactly known for each ensemble [Merz2018], [Shirts2013]. Providing two simulations at different state points therefore allows a validation of the sampled ensemble.

Note that the ensemble validation function is automatically inferring the correct test based on the simulation input data (such as temperature and pressure) that are given as input.

5.1 Choice of the state points

As the above ensemble tests require two simulations at distinct state points, the choice of interval between the two points is an important question. Choosing two state points too far apart will result in poor or zero overlap between the distributions, leading to very noisy results (due to sample errors in the tails) or a breakdown of the method, respectively. Choosing two state points very close to each others, on the other hand, makes it difficult to distinguish the slope from statistical error in the samples.

A rule of thumb states [Shirts2013] that the maximal efficiency of the method is reached when the distance between the peaks of the distributions are roughly equal to the sum of their standard deviations. For most systems with the exception of extremely small or very cold systems, it is reasonable to assume that the difference in standard deviations between the state points will be negligable. This leads to two ways of calculating the intervals:

Using calculated standard deviations: Given a simulation at one state point, the standard deviation of the distributions can be calculated numerically. The suggested intervals are then given by

- $\Delta T = 2k_B T^2/\sigma_E$, where σ_E is the standard deviation of the energy distribution used in the test (potential energy, enthalpy, or total energy).
- $\Delta P = 2k_B T/\sigma_V$, where σ_V is the standard deviation of the volume distribution.

Using physical observables: The standard deviations can also be estimated using physical observables such as the heat capacity and the compressibility. The suggested intervals are then given by:

- $\Delta T = T(2k_B/C_V)^{1/2}$ (NVT), or $\Delta T = T(2k_B/C_P)^{1/2}$ (NPT), where C_V and C_P denote the isochoric and the isobaric heat capacities, respectively.
- $\Delta P = (2k_B T/V\kappa_T)$, where κ_T denotes the isothermal compressibility.

When setting `verbosity >= 1` in *physical_validation.ensemble.check()*, the routine is printing an estimate for the optimal spacing based on the distributions provided. Additionally, *physical_validation.ensemble.estimate_interval()* calculates the estimate given a single simulation result. This can be used to determine at which state point a simulation should be repeated in order to efficiently check its sampled ensemble.

5.1.1 Function reference

physical_validation.ensemble.check()

physical_validation.ensemble.estimate_interval()

5.1.2 Example

Ensemble validation example

CHAPTER

SIX

INTEGRATOR VALIDATION

A symplectic integrator can be shown to conserve a constant of motion (such as the energy in a microcanonical simulation) up to a fluctuation that is quadratic in time step chosen. Comparing two or more constant-of-motion trajectories realized using different time steps (but otherwise unchanged simulation parameters) allows a check of the symplecticity of the integration. Note that lack of symplecticity does not necessarily imply an error in the integration algorithm, it can also hint at physical violations in other parts of the model, such as non-continuous potential functions, imprecise handling of constraints, etc.

6.1 Functions

physical_validation.integrator.convergence()

6.2 Example

Integrator convergence example

CHAPTER

SEVEN

KINETIC ENERGY DISTRIBUTION

Note: This notebook can be run locally by cloning the Github repository. *The notebook is located in* `doc/examples/kinetic_energy_distribution.ipynb`. *Be aware that probabilistic quantities such as error estimates based on bootstrapping will differ when repeating the analysis.*

[1]:
```
# enable plotting in notebook
%matplotlib notebook
```

The results imported here are the time series of kinetic and potential energy from example simulations, which are stored in another Python file. In real-world usage, the results would either come from the Python interface of the simulation package, from flat files containing the results, or from package-specific parsers. See *SimulationData* for more details.

[2]:
```
from simulation_results import example_simulations

import physical_validation
```

In this example, we will check the distributions of two simulations of 900 water molecules simulated under NVT conditions at 298.15K. One simulation used velocity-rescale temperature coupling, the other used Berendsen temperature coupling.

[3]:
```
simulation_vrescale = example_simulations.get(
    "900 water molecules, NVT at 298K with v-rescale thermostat"
)
simulation_berendsen = example_simulations.get(
    "900 water molecules, NVT at 298K with Berendsen thermostat"
)
```

First, we will create the object containing system information. The example system used in both simulations consists of 900 water molecules which are fully constrained. For the kinetic energy distribution check, we only need information about the complete system (i.e. no per-molecule or per-atom information) so we will only fill these.

[4]:
```
num_molecules = 900
# Each water molecule has three atoms
num_atoms_per_molecule = 3
# Each fully constrained water molecule has three constraints
num_constraints_per_molecule = 3
# In this simulation, translational center of mass motion was removed
num_constrained_translational_dof = 3
# Rotational center of mass motion was not removed
num_constrained_rotational_dof = 0

system_data = physical_validation.data.SystemData(
```

(continues on next page)

(continued from previous page)

```
    natoms=num_molecules * num_atoms_per_molecule,
    nconstraints=num_molecules * num_constraints_per_molecule,
    ndof_reduction_tra=num_constrained_translational_dof,
    ndof_reduction_rot=num_constrained_rotational_dof,
)
```

Next, we will create the ensemble information:

[5]:
```
ensemble_data = physical_validation.data.EnsembleData(
    ensemble="NVT",
    natoms=num_molecules * 3,
    volume=3.01125 ** 3,
    temperature=298.15,
)
```

Now we will create the simulation data objects which we will feed to the physical validation tests.

[6]:
```
simulation_data_vrescale = physical_validation.data.SimulationData(
    # Example simulations were performed using GROMACS
    units=physical_validation.data.UnitData.units("GROMACS"),
    system=system_data,
    ensemble=ensemble_data,
    observables=physical_validation.data.ObservableData(
        # This test requires only the kinetic energy
        kinetic_energy=simulation_vrescale["kinetic energy"]
    ),
)
simulation_data_berendsen = physical_validation.data.SimulationData(
    # Example simulations were performed using GROMACS
    units=physical_validation.data.UnitData.units("GROMACS"),
    system=system_data,
    ensemble=ensemble_data,
    observables=physical_validation.data.ObservableData(
        # This test requires only the kinetic energy
        kinetic_energy=simulation_berendsen["kinetic energy"]
    ),
)
```

We can now check the velocity-rescale temperature coupling, using first the strict, then the non-strict kinetic energy distribution check. We are using `screen=True` to display a result plot on screen (see argument `filename` to print that same plot to file).

[7]:
```
physical_validation.kinetic_energy.distribution(
    data=simulation_data_vrescale, strict=True, screen=True
)
```

```
After equilibration, decorrelation and tail pruning, 96.02% (4802 frames) of original␣
↪Kinetic energy remain.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Kinetic energy distribution check (strict)
Kolmogorov-Smirnov test result: p = 0.898369
```

(continues on next page)

(continued from previous page)

```
Null hypothesis: Kinetic energy is Maxwell-Boltzmann distributed
```

[7]:
```
0.89836894520092
```

The `strict` test prints and returns a p-value which indicates that the null hypothesis stands with high confidence. The printed figure confirms that the sampled and the analytical distribution are very similar.

The non-strict test confirms this result:

[8]:
```
# We turn plotting off here (`screen=False`), because the plot is
# identical to the one in the strict test
physical_validation.kinetic_energy.distribution(
    data=simulation_data_vrescale, strict=False, screen=False
)
```

```
After equilibration, decorrelation and tail pruning, 96.02% (4802 frames) of original␣
↪Kinetic energy remain.
Kinetic energy distribution check (non-strict)
Analytical distribution (T=298.15 K):
 * mu: 6689.47 kJ/mol
 * sigma: 128.77 kJ/mol
Trajectory:
 * mu: 6688.12 +- 1.88 kJ/mol
   T(mu) = 298.09 +- 0.08 K
 * sigma: 128.70 +- 1.35 kJ/mol
   T(sigma) = 297.98 +- 3.14 K
```

[8]:
```
(0.714917087637526, 0.053451278986070015)
```

The non-strict test calculates the temperature of the mean and the width of the distribution. Analytically, we are expecting both to be close to 298.15K. The result confirms that the simulation behaves as expected, with both the calculated mean and variance being within one standard deviation of the analytical value. The returned tuple is the number of standard deviations the calculated mean and variance differ from the expected value - less than a standard deviations for the mean, and only about 0.05 standard deviations for the variance. Note that any deviation within about 2-3 standard deviations is usually considered as statistically insignificant.

Also note that the error estimates were computed using bootstrapping and are hence a probabilistic quantity which will slightly differ when repeating the analysis.

We can now compare this to the Berendsen temperature coupling, using first the strict, then the non-strict kinetic energy distribution check.

[9]:
```
physical_validation.kinetic_energy.distribution(
    data=simulation_data_berendsen, strict=True, screen=True
)
```

```
After equilibration, decorrelation and tail pruning, 95.96% (4799 frames) of original␣
↪Kinetic energy remain.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Kinetic energy distribution check (strict)
Kolmogorov-Smirnov test result: p = 1.85798e-17
Null hypothesis: Kinetic energy is Maxwell-Boltzmann distributed
```

[9]:
```
1.8579783151867675e-17
```

The calculated p-value means that the trajectory is *very* unklikely to come from a Maxwell-Boltzmann distribution at 298.15K. Commonly, one would consider a p-value of less than 0.05 (5% confidence interval) a reasonable cut-off, which means that the test results make it easy to reject the null hypothesis that the kinetic energy samples the correct distribution. This does not come as a suprise, the Berendsen thermostat is known to keep the correct mean temperature, but not to sample the right distribution.

[10]:
```
# We turn plotting off here (`screen=False`), because the plot is
# identical to the one in the strict test
physical_validation.kinetic_energy.distribution(
    data=simulation_data_berendsen, strict=False, screen=False
)
```

```
After equilibration, decorrelation and tail pruning, 95.96% (4799 frames) of original␣
↪Kinetic energy remain.
Kinetic energy distribution check (non-strict)
Analytical distribution (T=298.15 K):
 * mu: 6689.47 kJ/mol
 * sigma: 128.77 kJ/mol
Trajectory:
 * mu: 6690.21 +- 1.29 kJ/mol
   T(mu) = 298.18 +- 0.06 K
 * sigma: 98.81 +- 1.01 kJ/mol
   T(sigma) = 228.78 +- 2.33 K
```

[10]:
```
(0.5763182556173292, 29.80276265170178)
```

The non-strict test confirms the conclusion from the strict test, but adds another explanation of the sampled distribution. The mean is found to be within about half a standard deviation of the expected value. The width of the sampled distribution, however, is what we would expect for a simulation at about 229K. This means that especially for simulations which rely on correct sampling of the tail regions, the Berendsen temperature coupling is likely to introduce artefacts. Again, the tuple returned here could be slightly different in different trials due to its probabilistic nature.

CHAPTER

EIGHT

ENSEMBLE CHECK

Note: This notebook can be run locally by cloning the Github repository. *The notebook is located in* `doc/examples/ensemble_check.ipynb`. *Be aware that probabilistic quantities such as error estimates based on bootstrapping will differ when repeating the analysis.*

[1]:
```
# enable plotting in notebook
%matplotlib notebook
```

The results imported here are the time series of kinetic and potential energy from example simulations, which are stored in another Python file. In real-world usage, the results would either come from the Python interface of the simulation package, from flat files containing the results, or from package-specific parsers. See *SimulationData* for more details.

[2]:
```
from simulation_results import example_simulations

import physical_validation
```

8.1 Check NVT simulations

To check the configurational quantities in NVT, two (otherwise identical) simulations run at different temperatures are required.

We start by loading the first NVT simulation of 900 water molecules, which was performed at 298.15K using velocity-rescale temperature coupling.

[3]:
```
simulation_nvt_vrescale_low = example_simulations.get(
    "900 water molecules, NVT at 298K with v-rescale thermostat"
)
num_molecules = 900
simulation_data_nvt_low = physical_validation.data.SimulationData(
    # Example simulations were performed using GROMACS
    units=physical_validation.data.UnitData.units("GROMACS"),
    ensemble=physical_validation.data.EnsembleData(
        ensemble="NVT",
        natoms=num_molecules * 3,
        volume=3.01125 ** 3,
        temperature=298.15,
    ),
    observables=physical_validation.data.ObservableData(
        # This test requires only the potential energy
        potential_energy=simulation_nvt_vrescale_low["potential energy"]
```

(continues on next page)

(continued from previous page)

```
    ),
)
```

It is not trivial to decide at which temperature to perform a second simulation. The best results are achieved when the two simulations are close enough to have good overlap between the distributions, while keeping them far enough apart to be able to distinguish the physical difference between the distributions from the statistical error present in simulations.

`physical_validation` offers functionality to compute a rule-of-thumb estimate of the optimal interval in state point between two functions. We will now use our first simulation result to get an estimate of where a second simulation would optimally be located:

[4]:
```
physical_validation.ensemble.estimate_interval(
    data=simulation_data_nvt_low,
)
```

```
A rule of thumb states that good error recognition can be expected when
spacing the tip of the distributions by about two standard deviations.
Based on this rule, and the assumption that the standard deviation of the
distributions is largely independent of the state point, here's an estimate
for the interval given the current simulation:
Current trajectory: NVT, T = 298.15 K
Suggested interval: dT = 8.4 K
```

[4]:
```
{'dT': 8.44234664462332}
```

The second simulation available in our example set was performed at 308.15K, which is reasonably close to the estimate calculated above. Let's load these results:

[5]:
```
simulation_nvt_vrescale_high = example_simulations.get(
    "900 water molecules, NVT at 308K with v-rescale thermostat"
)
simulation_data_nvt_high = physical_validation.data.SimulationData(
    # Example simulations were performed using GROMACS
    units=physical_validation.data.UnitData.units("GROMACS"),
    ensemble=physical_validation.data.EnsembleData(
        ensemble="NVT",
        natoms=num_molecules * 3,
        volume=3.01125 ** 3,
        temperature=308.15,
    ),
    observables=physical_validation.data.ObservableData(
        # This test requires only the potential energy
        potential_energy=simulation_nvt_vrescale_high["potential energy"]
    ),
)
```

Using both simulation data objects, we can now check the ensemble sampled by our simulations. We are using `screen=True` to display a result plot on screen. See argument `filename` to print that same plot to file.

[6]:
```
physical_validation.ensemble.check(
    data_sim_one=simulation_data_nvt_low,
    data_sim_two=simulation_data_nvt_high,
    screen=True,
```

(continues on next page)

(continued from previous page)

```
)
```

```
After equilibration, decorrelation and tail pruning, 89.40% (4471 frames) of original␣
↪Trajectory 1 remain.
After equilibration, decorrelation and tail pruning, 90.72% (4537 frames) of original␣
↪Trajectory 2 remain.
Overlap is 82.4% of trajectory 1 and 73.2% of trajectory 2.
Rule of thumb estimates that dT = 8.6 would be optimal (currently, dT = 10.0)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
==================================================
Maximum Likelihood Analysis (analytical error)
==================================================
Free energy
    477.11477 +/- 10.85952
Estimated slope                  |  True slope
    0.013409  +/- 0.000305       |  0.013091
    (1.04 quantiles from true slope)
Estimated dT                     |  True dT
    10.2   +/- 0.2               |  10.0
==================================================
```

[6]:
```
[1.0426587118546036]
```

By default, the ensemble check is estimating the distance in temperature between the two sampled ensembles using a maximum likelihood approach. This distance estimate is expected to be close to the true value. As a rule of thumb, if the true interval is not within about 2-3 standard deviations of the estimated interval, the trajectory is unlikely to have been sampled from the expected ensemble. The quantiles (number of standard deviations) of difference between the true value and the estimate is returned from the test as a machine-readable test result.

Note that in order to print the plot, the line is also linearly fitted to the simulations. This leads to a slightly different estimate, which explains the difference between the quantiles printed in the plot and in the terminal. As the maximum likelihood estimate is considered to be more exact, it's value is reported on the terminal and used as a return value.

We will now repeat this analysis using the same system, but a simulation which was performed using Berendsen pressure coupling. This temperature coupling method was found not to sample the expected ensemble. We will use this example to illustrate that the `physical_validation` checks are able to pick up this discrepancy in sampling.

Since the simulated system was identical to the one first analyzed, we will simply replace the observable trajectory in our simulation data objects:

[7]:
```
simulation_nvt_berendsen_low = example_simulations.get(
    "900 water molecules, NVT at 298K with Berendsen thermostat"
)
simulation_data_nvt_low.observables = physical_validation.data.ObservableData(
    potential_energy=simulation_nvt_berendsen_low["potential energy"]
)
simulation_nvt_berendsen_high = example_simulations.get(
    "900 water molecules, NVT at 308K with Berendsen thermostat"
)
simulation_data_nvt_high.observables = physical_validation.data.ObservableData(
    potential_energy=simulation_nvt_berendsen_high["potential energy"]
)
```

[8]:
```
physical_validation.ensemble.check(
    data_sim_one=simulation_data_nvt_low,
    data_sim_two=simulation_data_nvt_high,
    screen=True,
)
```

```
After equilibration, decorrelation and tail pruning, 89.28% (4465 frames) of original␣
↪Trajectory 1 remain.
After equilibration, decorrelation and tail pruning, 99.80% (4991 frames) of original␣
↪Trajectory 2 remain.
Overlap is 44.3% of trajectory 1 and 45.6% of trajectory 2.
Rule of thumb estimates that dT = 11.4 would be optimal (currently, dT = 10.0)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
================================================
Maximum Likelihood Analysis (analytical error)
================================================
Free energy
    808.62976 +/- 22.44259
Estimated slope                  |  True slope
    0.022723  +/- 0.000631       |  0.013091
    (15.27 quantiles from true slope)
Estimated dT                     |  True dT
    17.4   +/- 0.5               |  10.0
================================================
```

[8]:
```
[15.271945829603142]
```

The check is confirming that the sampled ensemble using the Berendsen thermostat is not behaving as expected when changing the temperature. The reported estimated temperature interval is around 15 standard deviations from the true value, which makes it easy to reject the hypothesis that the potential energy was sampled from the correct ensemble.

8.2 Check NPT simulations

To check the sampled ensemble of the configurational quantities in NPT, we again need two otherwise identical simulations which were performed at slightly different state points (target temperature and / or pressure). The checks can be performed using identical pressure and different temperatures, which will test whether the sampled ensembles exhibit the expected temperature dependence. The checks can also be performed using identical temperature and different pressures, which will in turn test the pressure dependence of the sampled ensemble. Finally, we can also use two simulations which differs in both the target temperature and pressure, combining the two tests into one. Here, we will showcase the last option for a system of 900 water molecules, sampled using velocity-rescale temperature coupling and Parrinello-Rahman pressure coupling. These coupling algorithms were analytically shown to sample the correct distribution, so we will check whether the simulated results fulfill this expectation.

We will start by loading a first simulation, performed at 298.15 K and 1 bar.

[9]:
```
simulation_npt_low = example_simulations.get(
    "900 water molecules, NPT at 298K and 1bar, using v-rescale and Parrinello-Rahman"
)
num_molecules = 900
simulation_data_npt_low = physical_validation.data.SimulationData(
```

(continues on next page)

(continued from previous page)

```
    # Example simulations were performed using GROMACS
    units=physical_validation.data.UnitData.units("GROMACS"),
    ensemble=physical_validation.data.EnsembleData(
        ensemble="NPT",
        natoms=num_molecules * 3,
        pressure=1.0,
        temperature=298.15,
    ),
    observables=physical_validation.data.ObservableData(
        # This test requires the potential energy and the volume
        potential_energy=simulation_npt_low["potential energy"],
        volume=simulation_npt_low["volume"],
    ),
)
```

As in the NVT case, we can use this simulation to have `physical_validation` suggesting a state point to perform the second simulation in.

[10]:
```
physical_validation.ensemble.estimate_interval(data=simulation_data_npt_low)
```

```
A rule of thumb states that good error recognition can be expected when
spacing the tip of the distributions by about two standard deviations.
Based on this rule, and the assumption that the standard deviation of the
distributions is largely independent of the state point, here's an estimate
for the interval given the current simulation:
Current trajectory: NPT, T = 298.15 K, P = 1.00 bar
Suggested interval:
  Temperature-only: dT = 7.7 K
  Pressure-only: dP = 315.1 bar
  Combined: dT = 7.8 K, dP = 314.8 bar
```

[10]:
```
{'dT': 7.749821872873016,
 'dP': 315.11202690293436,
 'dTdP': [7.763310613476873, 314.82913138004045]}
```

The rule of thumb suggests that a second state point with a temperature difference of about 7.8 K and a pressure difference of about 315 bar would be optimal. The second simulation which is available in our example set was performed at 308.15 K and 101 bar, so at a distance of 10 K and 100 bar. According to the `physical_validation` estimate, the pressure distance should be a bit further to have optimal error recognition. The check will, however, not be invalid with this choice of state points.

[11]:
```
simulation_npt_high = example_simulations.get(
    "900 water molecules, NPT at 308K and 101bar, using v-rescale and Parrinello-Rahman"
)
num_molecules = 900
simulation_data_npt_high = physical_validation.data.SimulationData(
    # Example simulations were performed using GROMACS
    units=physical_validation.data.UnitData.units("GROMACS"),
    ensemble=physical_validation.data.EnsembleData(
        ensemble="NPT",
        natoms=num_molecules * 3,
        pressure=101.0,
        temperature=308.15,
```

(continues on next page)

(continued from previous page)

```
    ),
    observables=physical_validation.data.ObservableData(
        # This test requires the potential energy and the volume
        potential_energy=simulation_npt_high["potential energy"],
        volume=simulation_npt_high["volume"],
    ),
)
```

Using both simulation data objects, we can now check the ensemble sampled by our simulations. Note that plotting is not available for NPT simulations which differ in both temperature and pressure, since the 2-dimensional plot would be very hard to interpret.

[12]:
```
physical_validation.ensemble.check(
    data_sim_one=simulation_data_npt_low,
    data_sim_two=simulation_data_npt_high,
)
```

```
After equilibration, decorrelation and tail pruning, 92.46% (4624 frames) of original␣
↪Trajectory 1 remain.
After equilibration, decorrelation and tail pruning, 86.86% (4344 frames) of original␣
↪Trajectory 2 remain.
Overlap is 72.8% of trajectory 1 and 76.7% of trajectory 2.
Rule of thumb estimates that (dT,dP) = (8.0,320.4) would be optimal (currently, (dT,dP)␣
↪= (10.0,100.0))
=====================================================
Maximum Likelihood Analysis (analytical error)
=====================================================
Free energy
    521.66706 +/- 13.00986
Estimated slope                  |  True slope
    0.012963  +/- 0.000301       |  0.013091
    (0.42 quantiles from true slope)
    -2.232868 +/- 0.157176       |  -2.349681
    (0.74 quantiles from true slope)
Estimated dT                     |  True dT
    9.9    +/- 0.2               |  10.0
Estimated dP                     |  True estimated dP
    93.5   +/- 6.6               |  98.3
=====================================================
```

[12]:
```
array([0.42492725, 0.74319977])
```

The ensemble check now prints both the estimated temperature and pressure intervals. We note that in both cases, the true value is within less than a standard deviation, which means that the null hypothesis of sampling the expected ensemble stands.

It's worth noting that the true pressure difference is given as 98.3 bar rather than 100 bar. When checking simulations which differ in both their pressure and temperature, the pressure interval can only be approximated since the temperature and pressure are not perfectly separable in the NPT partition function. Please refer to Merz & Shirts 2018, eq (18), for details.

8.3 Check μVT simulations

To check the sampled ensemble of the configurational quantities in μVT, we again need two otherwise identical simulations which were performed at slightly different state points (target temperature and / or chemical potential). The checks can be performed using identical chemical potential and different temperatures, which will test whether the sampled ensembles exhibit the expected temperature dependence. The checks can also be performed using identical temperature and different chemical potentials, which will in turn test the dependence of the sampled ensemble on the chemical potential. Finally, we can also use two simulations which differ in both the target temperature and chemical potentials, combining the two tests into one. Here, we will showcase checking the dependence on the chemical potential for a system of difluoromethane vapor, sampled using GCMC.

We will start by loading a first simulation, performed at a temperature of 300K and chemical potential of -37 kJ/mol.

[13]:
```
simulation_muvt_low = example_simulations.get(
    "GCMC Difluoromethane vapor, muVT at 300K and -37.5kJ/mol"
)
simulation_data_muvt_low = physical_validation.data.SimulationData(
    # Example simulations were performed in units compatible with GROMACS units
    units=physical_validation.data.UnitData.units("GROMACS"),
    ensemble=physical_validation.data.EnsembleData(
        ensemble="muVT",
        mu=-37.5,
        volume=512,
        temperature=300,
    ),
    observables=physical_validation.data.ObservableData(
        # This test requires the potential energy and the volume
        potential_energy=simulation_muvt_low["potential energy"],
        number_of_species=simulation_muvt_low["number of species"],
    ),
)
```

As for the other ensembles, we can use this simulation to have `physical_validation` suggesting a state point to perform the second simulation in.

[14]:
```
physical_validation.ensemble.estimate_interval(data=simulation_data_muvt_low)
```

```
################################################################################
# WARNING: Support for muVT ensemble is an experimental feature under current #
#          development. You can help us to improve it by reporting errors     #
#          at https://github.com/shirtsgroup/physical_validation              #
#          Thank you!                                                         #
################################################################################
A rule of thumb states that good error recognition can be expected when
spacing the tip of the distributions by about two standard deviations.
Based on this rule, and the assumption that the standard deviation of the
distributions is largely independent of the state point, here's an estimate
for the interval given the current simulation:
Current trajectory: muVT, T = 300.00 K, mu = -37.50 kJ/mol
Suggested interval:
  Temperature-only: dT = 4.3 K
  Chemical potential-only: dmu = 0.6 kJ/mol
  Combined: dT = 27.6 K, dmu = 0.6 kJ/mol
```

[14]:
```
{'dT': 4.315794686477163,
 'dmu': 0.6161396912499071,
 'dTdmu': [27.615821479158512, 0.6188326725201275]}
```

The rule of thumb suggests that a second state point with chemical potential difference of about 0.6 kJ/mol would be optimal. For the temperature, the suggestion of a second state point varies depending on whether a 1D fit (simulations varying only in their target temperature) or a 2D fit (simulations varying in both their target temperature and the chemical potential) is to be performed.

The second simulation available in the example set was performed at 300K and -37.0 kJ/mol. The chemical potential difference (0.5 kJ/mol) is close to the value suggested by `physical_validation`, so we should get close to optimal error recognition when testing for the dependence on the chemical potential.

We proceed by loading the second simulation:

[15]:
```
simulation_muvt_high = example_simulations.get(
    "GCMC Difluoromethane vapor, muVT at 300K and -37.0kJ/mol"
)
simulation_data_muvt_high = physical_validation.data.SimulationData(
    # Example simulations were performed in units compatible with GROMACS units
    units=physical_validation.data.UnitData.units("GROMACS"),
    ensemble=physical_validation.data.EnsembleData(
        ensemble="muVT",
        mu=-37.0,
        volume=512,
        temperature=300,
    ),
    observables=physical_validation.data.ObservableData(
        # This test requires the potential energy and the volume
        potential_energy=simulation_muvt_high["potential energy"],
        number_of_species=simulation_muvt_high["number of species"],
    ),
)
```

Using both simulation data objects, we can now check the ensemble sampled by our simulations. We are using `screen=True` to display a result plot on screen. See argument `filename` to print that same plot to file.

[16]:
```
physical_validation.ensemble.check(
    data_sim_one=simulation_data_muvt_low,
    data_sim_two=simulation_data_muvt_high,
    screen=True
)
```

```
################################################################################
# WARNING: Support for muVT ensemble is an experimental feature under current #
#          development. You can help us to improve it by reporting errors     #
#          at https://github.com/shirtsgroup/physical_validation              #
#          Thank you!                                                         #
################################################################################
After equilibration, decorrelation and tail pruning, 55.45% (2496 frames) of original␣
↪Trajectory 1 remain.
After equilibration, decorrelation and tail pruning, 34.26% (1542 frames) of original␣
↪Trajectory 2 remain.
Overlap is 89.4% of trajectory 1 and 88.0% of trajectory 2.
Rule of thumb estimates that mu = 0.6 would be optimal (currently, dmu = 0.5)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
================================================
Maximum Likelihood Analysis (analytical error)
================================================
Free energy
    -14.82375 +/- 0.51045
Estimated slope                  |  True slope
    0.201836  +/- 0.007093       |  0.200454
    (0.19 quantiles from true slope)
Estimated dmu                    |  True estimated dmu
    -0.5   +/- 0.0               |  -0.5
================================================
```

[16]:
```
[0.1948414103184058]
```

The ensemble check finds a slope which is 0.2 quantiles from the real slope according to the Maximum-Likelihood estimator. This is a strong indicator that the sampled ensembles exhibit the correct dependence on the target chemical potential.

CHAPTER

NINE

KINETIC ENERGY EQUIPARTITION

Note: This notebook can be run locally by cloning the Github repository. *The notebook is located in* `doc/examples/kinetic_energy_equipartition.ipynb`. *Be aware that probabilistic quantities such as error estimates based on bootstrapping will differ when repeating the analysis.*

[1]:
```
# enable plotting in notebook
%matplotlib notebook
```

The results imported here are the time series of kinetic and potential energy, positions and velocities from example simulations, which are stored in another Python file. In real-world usage, the results would either come from the Python interface of the simulation package, from flat files containing the results, or from package-specific parsers. See *SimulationData* for more details.

[2]:
```
from simulation_results import example_simulations

import physical_validation
import numpy as np
```

9.1 Check solute from solvated simulation

`physical_validation` does not only allow to test the kinetic energy of the entire system, but also the kinetic energy of parts of the system, or how the kinetic energy is distributed within the molecules (translational, rotational and internal kinetic energy).

To demonstrate this check, we will first look at a simulation of a Trp-cage mini-protein. This simulation was performed in water, using a single thermostat connected to the entire system (solute and solvent). Since there is significantly more solvent than solute in typical solvated simulations, unphysical behavior of the solute might be hidden due to the large amount of solvent when looking at full-system properties. We will therefore look at the trajectory which was stripped from the water in post-processing to check the kinetic energy of the solute alone.

We will first load the results from the imported example simulations:

[3]:
```
simulation = example_simulations.get(
    "Trp-cage, NPT at 300K and 1bar, protein trajectory only"
)
```

Next, we will create the object containing system information. The single Trp-cage molecule consists of 304 atoms, and has 310 constrained bonds. The translational center of mass motion was removed during the simulation, while the rotation of the center of mass was not constrained.

The equipartition check does not only require information for the entire system, but also details about the atoms and the different molecules. We need to provide the mass of each atom in the system to allow the test to calculate the center

of mass of each molecule at each step. We also need to provide the index of the first atom of each molecule, and the number of constraints of each molecule. In our case, the latter two quantities are an array with a single entry, because we operate on a single molecule. If we had, for example, two fully constrained water molecules in our system instead of the Trp-cage protein, the molecule index would read `np.array([0, 3])` (the first atom of the first water molecule has index 0, the first atom of the second water molecule has index 3), while the number of constraints per molecule would read `np.array([3, 3])` (3 constraints in each water molecule).

Here, we are loading these values from the example simulation data structure. In typical use, this information needs to be read from the simulation package: Through its Python interface if one is available, by the user otherwise.

[4]:
```
system_data = physical_validation.data.SystemData(
    # Read system information from the example simulation
    natoms=simulation["natoms"],
    nconstraints=simulation["nconstraints"],
    ndof_reduction_tra=simulation["ndof_reduction_tra"],
    ndof_reduction_rot=simulation["ndof_reduction_rot"],
    mass=simulation["mass"],
    # We only have one molecule
    molecule_idx=np.array([0]),
    # The single molecule has all constraints of the system
    nconstraints_per_molecule=np.array([simulation["nconstraints"]]),
)
```

Next, we will create the ensemble information:

[5]:
```
ensemble_data = physical_validation.data.EnsembleData(
    ensemble="NPT",
    natoms=simulation["natoms"],
    pressure=1.0,
    temperature=300.0,
)
```

Now we will create the simulation data objects which we will feed to the physical validation tests.

[6]:
```
simulation_data = physical_validation.data.SimulationData(
    # Example simulations were performed using GROMACS
    units=physical_validation.data.UnitData.units("GROMACS"),
    system=system_data,
    ensemble=ensemble_data,
    # The equipartition test needs position and velocity trajectories
    trajectory=physical_validation.data.TrajectoryData(
        position=simulation["position"],
        velocity=simulation["velocity"],
    )
)
```

The simulation was performed using a single thermostat connected to the entire system (solute and solvent). Since there is significantly more solvent than solute in typical solvated simulations, unphysical behavior of the solute might be hidden We are now analyzing the solute alone.

We can now check the kinetic energy distribution of the solute. By default, the equipartition check will verify the distribution of the total, the translational, the rotational, and the internal kinetic energy. We are using `screen=True` to display a result plot on screen (see argument `filename` to print that same plot to file).

[7]:
```
physical_validation.kinetic_energy.equipartition(
    data=simulation_data, strict=True, screen=True
)
```

```
Equipartition: Testing group-wise kinetic energies (strict)
* total:
After equilibration, decorrelation and tail pruning, 90.60% (906 frames) of original␣
↪Kinetic energy remain.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
p = 0.19142
* translational:
After equilibration, decorrelation and tail pruning, 98.70% (987 frames) of original␣
↪Kinetic energy remain.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
p = 0.222366
* rotational and internal:
After equilibration, decorrelation and tail pruning, 89.30% (893 frames) of original␣
↪Kinetic energy remain.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
p = 0.214754
* rotational:
After equilibration, decorrelation and tail pruning, 100.00% (1000 frames) of original␣
↪Kinetic energy remain.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
p = 0.387559
* internal:
After equilibration, decorrelation and tail pruning, 91.30% (913 frames) of original␣
↪Kinetic energy remain.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
p = 0.137762
```

[7]:
```
[0.19141994837327247,
 0.22236562034118057,
 0.21475378052598215,
 0.3875590890276527,
 0.13776199439893277]
```

The `strict` test prints and returns p-values which indicate that the null hypothesis stands for all different kinetic energies. Commonly, one would consider a p-value of less than 0.05 (5% confidence interval) a reasonable cut-off. The printed figures confirm that the sampled and the analytical distributions are very similar in all cases.

We can therefore be confident that the full system temperature coupling does indeed lead to the solute sampling the correct kinetic energy distribution.

9.2 Check gas phase simulations for correctness

As another example, we will now look at two simulations of octanol in gas phase at 298.15 K. Each simulation consisted of 512 non-interacting octanol molecules. One of the simulations was performed using molecular dynamics with velocity-rescale temperature coupling, while the other was performed using stochastic dynamics.

[8]:
```
simulation_md = example_simulations.get(
    "512 octanol molecules in gas phase at 298K, MD with v-rescale"
)
simulation_sd = example_simulations.get(
    "512 octanol molecules in gas phase at 298K, SD"
)
```

The system consists of 512 united-atom 1-octanol molecules. Each (linear) molecule is made up of 10 atoms, with 9 constrained bonds between them. In both simulations, the translational center of mass motion was removed, while the rotational was unconstrained. The mass vector is a 512-fold repetition of the masses of a single octanol molecule. The molecule index starts at atom 0 and increases in increments of 10 until the last molecule which starts at atom 511x10 = 5110. The number of constraints per molecule is 9 for all molecules

[9]:
```
num_molecules = 512
num_atoms_per_molecule = 10
num_constraints_per_molecule = num_atoms_per_molecule - 1

system_data = physical_validation.data.SystemData(
    # Read system information from the example simulation
    natoms=num_molecules * num_atoms_per_molecule,
    nconstraints=num_molecules * num_constraints_per_molecule,
    ndof_reduction_tra=3,
    ndof_reduction_rot=0,
    # np.tile repeats the single-molecule mass vector `num_molecules` times
    mass=np.tile(
        [
            1.008,
            15.9994,
            14.027,
            14.027,
            14.027,
            14.027,
            14.027,
            14.027,
            14.027,
            15.035,
        ],
        num_molecules,
    ),
    # The first atom of each molecule: [0, 10, 20, ..., 5110]
    molecule_idx=np.arange(
        0, num_molecules * num_atoms_per_molecule, num_atoms_per_molecule
    ),
    # Each molecule has nine constraints: [9, 9, ..., 9]
    nconstraints_per_molecule=num_constraints_per_molecule * np.ones(num_molecules),
)
```

The simulations were performed under NVT:

[10]:
```
ensemble_data = physical_validation.data.EnsembleData(
    ensemble="NVT",
    natoms=num_molecules * num_atoms_per_molecule,
    volume=20,
    temperature=298.15,
)
```

We can now create the two simulation data objects:

[11]:
```
simulation_data_md = physical_validation.data.SimulationData(
    # Example simulations were performed using GROMACS
    units=physical_validation.data.UnitData.units("GROMACS"),
    system=system_data,
    ensemble=ensemble_data,
    observables=physical_validation.data.ObservableData(
        kinetic_energy=simulation_md["kinetic energy"],
        potential_energy=simulation_md["potential energy"],
    ),
    trajectory=physical_validation.data.TrajectoryData(
        position=simulation_md["position"],
        velocity=simulation_md["velocity"],
    ),
)
simulation_data_sd = physical_validation.data.SimulationData(
    # Example simulations were performed using GROMACS
    units=physical_validation.data.UnitData.units("GROMACS"),
    system=system_data,
    ensemble=ensemble_data,
    observables=physical_validation.data.ObservableData(
        kinetic_energy=simulation_sd["kinetic energy"],
        potential_energy=simulation_sd["potential energy"],
    ),
    trajectory=physical_validation.data.TrajectoryData(
        position=simulation_sd["position"],
        velocity=simulation_sd["velocity"],
    ),
)
```

What is remarkable about these two simulations, is that they both seem to sample the correct full-system kinetic energy distribution:

[12]:
```
physical_validation.kinetic_energy.distribution(data=simulation_data_md, strict=True)
```

```
After equilibration, decorrelation and tail pruning, 100.00% (76 frames) of original␣
↪Kinetic energy remain.
Kinetic energy distribution check (strict)
Kolmogorov-Smirnov test result: p = 0.286917
Null hypothesis: Kinetic energy is Maxwell-Boltzmann distributed
```

[12]:
```
0.28691678127130094
```

[13]:
```
physical_validation.kinetic_energy.distribution(data=simulation_data_sd, strict=True)
```

```
After equilibration, decorrelation and tail pruning, 69.33% (52 frames) of original␣
↪Kinetic energy remain.
```

(continues on next page)

(continued from previous page)

```
Kinetic energy distribution check (strict)
Kolmogorov-Smirnov test result: p = 0.206281
Null hypothesis: Kinetic energy is Maxwell-Boltzmann distributed
```

[13]:
```
0.20628105920809725
```

Yet their potential energy average is significantly different, at about 3 standard deviations of difference between the two estimates:

[14]:
```
(simulation_data_md.observables.potential_energy.mean(), simulation_data_md.observables.
↪potential_energy.std())
```

[14]:
```
(8038.964400421053, 145.53838254074824)
```

[15]:
```
(simulation_data_sd.observables.potential_energy.mean(), simulation_data_sd.observables.
↪potential_energy.std())
```

[15]:
```
(7579.539869773332, 169.56653568511697)
```

Running the equipartition check for both simulations sheds some light into what is happening:

[16]:
```
physical_validation.kinetic_energy.equipartition(data=simulation_data_md, strict=False)
```

```
Equipartition: Testing group-wise kinetic energies (non-strict)
* total:
After equilibration, decorrelation and tail pruning, 100.00% (76 frames) of original␣
↪Kinetic energy remain.
T(mu) = 298.51 +- 0.41 K
T(sigma) = 275.45 +- 21.05 K
* translational:
After equilibration, decorrelation and tail pruning, 39.47% (30 frames) of original␣
↪Kinetic energy remain.
T(mu) = 251.01 +- 0.86 K
T(sigma) = 125.65 +- 17.20 K
* rotational and internal:
After equilibration, decorrelation and tail pruning, 100.00% (76 frames) of original␣
↪Kinetic energy remain.
T(mu) = 305.70 +- 0.46 K
T(sigma) = 263.82 +- 23.94 K
* rotational:
After equilibration, decorrelation and tail pruning, 64.47% (49 frames) of original␣
↪Kinetic energy remain.
T(mu) = 275.60 +- 0.80 K
T(sigma) = 153.42 +- 16.98 K
* internal:
After equilibration, decorrelation and tail pruning, 93.42% (71 frames) of original␣
↪Kinetic energy remain.
T(mu) = 312.35 +- 0.41 K
T(sigma) = 219.04 +- 17.84 K
```

[16]:
```
[(0.8877078177579152, 1.0785152946004155),
 (54.75807873647404, 10.026739798585906),
 (16.32772106740449, 1.43385330148231),
 (28.132986949338967, 8.525395451107745),
 (34.95911199352702, 4.434764508076646)]
```

The MD simulation does sample the correct full system kinetic energy, with both the mean and the variance within about a standard deviation of the true value. The internal temperature is, however, significantly too high (over 30 standard deviations above the expected value), while the temperature of the translational and rotational degrees of freedom are too low. Note that any deviation within about 2-3 standard deviations is usually considered as statistically insignificant.

[17]:
```
physical_validation.kinetic_energy.equipartition(data=simulation_data_sd, strict=False)
```

```
Equipartition: Testing group-wise kinetic energies (non-strict)
* total:
After equilibration, decorrelation and tail pruning, 70.67% (53 frames) of original␣
↪Kinetic energy remain.
T(mu) = 297.72 +- 0.47 K
T(sigma) = 278.28 +- 26.70 K
* translational:
After equilibration, decorrelation and tail pruning, 53.33% (40 frames) of original␣
↪Kinetic energy remain.
T(mu) = 298.32 +- 1.63 K
T(sigma) = 282.62 +- 34.53 K
* rotational and internal:
After equilibration, decorrelation and tail pruning, 74.67% (56 frames) of original␣
↪Kinetic energy remain.
T(mu) = 297.67 +- 0.48 K
T(sigma) = 244.46 +- 25.16 K
* rotational:
After equilibration, decorrelation and tail pruning, 97.33% (73 frames) of original␣
↪Kinetic energy remain.
T(mu) = 296.16 +- 1.14 K
T(sigma) = 245.29 +- 23.93 K
* internal:
After equilibration, decorrelation and tail pruning, 84.00% (63 frames) of original␣
↪Kinetic energy remain.
T(mu) = 298.05 +- 0.56 K
T(sigma) = 269.77 +- 21.29 K
```

[17]:
```
[(0.9008763345265538, 0.7439987842383414),
 (0.10410349921537503, 0.4498521476007015),
 (0.9975145461349327, 2.1335398129869403),
 (1.7457884062564413, 2.2089826013916993),
 (0.17589254634896292, 1.3331699127717522)]
```

The SD simulation does, unlike the MD simulation, sample the right kinetic energy distribution for all degrees of freedom. Most crucially, it does sample the correct distribution for the internal degrees of freedom.

Since the molecules are in gas phase, i.e. non-interacting, all potential energy will come from intra-molecular interactions. Sampling the correct kinetic energy in the molecule-internal degree of freedom is hence key for a physically valid estimate of the potential energy. The MD simulation's elevated internal temperature explains why its potential energy estimate was significantly larger than the SD estimate.

CHAPTER

TEN

INTEGRATOR VALIDATION

Note: This notebook can be run locally by cloning the Github repository. *The notebook is located in* `doc/examples/integrator_validation.ipynb`.

[1]:
```
# enable plotting in notebook
%matplotlib notebook
```

The results imported here are the time series of kinetic and potential energy from example simulations, which are stored in another Python file. In real-world usage, the results would either come from the Python interface of the simulation package, from flat files containing the results, or from package-specific parsers. See *SimulationData* for more details.

[2]:
```
from simulation_results import example_simulations

import physical_validation
```

The example system consists of 1000 Lennard-Jones (LJ) particles simulated under NVE conditions. The system uses Argon parameters = 0.3345 nm and = 1.045128 kJ/mol (White 1999). To illustrate how the integrator convergence validation can pick up small errors in the simulation, the simulation was repeated with different interaction cutoff schemes. In all cases, the LJ interactions were discarded if the particles were more than 1 nm (3) apart. Uncorrected, this leads to a small discontinuity in the potential and the force at the cutoff distance. A first approach to correct this discontinuity is by shifting the potential by a constant value such that it reaches 0 at the cutoff distance. This fixes the discontinuity in the potential, but does not alter the forces. A second approach is to smoothly switch off both the forces and the potential such that both reach zero at the cut-off distance.

The integrator validation analyzes the convergence of the fluctuations around the integrator constant of motion. For a symplectic integrator, the fluctuation is expected to be directly proportional to the square of the time step. The `physical_validation` check calculates the fluctuation convergence from multiple simulations with different time steps and validates the expectation.

We will start by analyzing simulations without any cut-off corrections. The example simulations contain five otherwise identical simulations performed at timesteps of 0.004, 0.002, 0.001, 0.0005, and 0.00025 ps, respectively.

We will prepare for the analysis by creating a `SimulationData` object for each of these simulations.

[3]:
```
simulation_list = []
for timestep in [0.004, 0.002, 0.001, 0.0005, 0.00025]:
    # Load appropriate simulation from examples
    simulation = example_simulations.get(
        f"1000 Lennard-Jones particles, no cut-off correction, timestep {timestep} ps"
    )
    # Append simulation data object to the list. Since the integrator check only needs␣
↪knowledge
    # about the time step and the constant of motion, we leave the remaining fields␣
↪empty.
```

(continues on next page)

(continued from previous page)

```
    simulation_list.append(
        physical_validation.data.SimulationData(
            dt=timestep,
            observables=physical_validation.data.ObservableData(
                constant_of_motion=simulation["constant of motion"]
            ),
        )
    )
```

We can now pass the list of simulations into the integrator test to check whether the convergence behaves as expected:

[4]:
```
physical_validation.integrator.convergence(simulations=simulation_list, screen=True)
```

```
-------------------------------------------------------------------
        dt        avg       rmsd      slope         ratio
                                                  dt^2       rmsd
-------------------------------------------------------------------
     0.004   -4749.12   3.66e-01   2.86e-04         --         --
     0.002   -4749.27   3.72e-01   2.77e-04       4.00       0.99
     0.001   -4749.26   3.34e-01   3.47e-04       4.00       1.11
    0.0005   -4749.23   3.37e-01   3.33e-04       4.00       0.99
   0.00025   -4749.23   3.45e-01   2.54e-04       4.00       0.98
-------------------------------------------------------------------
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[4]:
```
0.7556745974285974
```

The output of the function consists of the time step, the average value of the constant of motion, and its RMSD during the simulation. The fourth column gives the measured slope of the constant of motion - a large value here would indicate a strong drift and hence a problem in the integrator. Even without strong drift, as in the current situation, a large deviation in the ratio between the RMSD values compared to the ratio between the time step will indicate some error in the integrator. The reason for a failure of this test might not always be intuitively clear, as many components play into the integrator convergence - the integrator algorithm itself, but also the interaction function (e.g. non-continuous cut-off) or the numerical precision of the floating point operations.

In this example, we have introduced the numerical error consciously by not correcting for the LJ cut-off discontinuity. The output makes it clear that there is hardly any time-step dependency of the fluctuation, indicating that the error due to the incorrect cut-off treatment is dominating the fluctuations.

The return value represents the largest deviation from the expected behavior, i.e. the maximum value of `abs(1 - rmsd_ratio / time_step_ratio_squared)`. This allows to evaluate the success of this function programmatically.

We can now compare this to the simulations in which the discontinuity in the potential was corrected by shifting it so that its value at the cut-off reaches zero. Since this is simply an additive constant to the potential, the forces are unchanged. We will again load a set of simulations, and then perform the integrator convergence check.

[5]:
```
simulation_list = []
for timestep in [0.004, 0.002, 0.001, 0.0005, 0.00025]:
    # Load appropriate simulation from examples
    simulation = example_simulations.get(
        f"1000 Lennard-Jones particles, potential shift cut-off correction, timestep
↪{timestep} ps"
    )
```

(continues on next page)

(continued from previous page)

```
    # Append simulation data object to the list. Since the integrator check only needs␣
↪knowledge
    # about the time step and the constant of motion, we leave the remaining fields␣
↪empty.
    simulation_list.append(
        physical_validation.data.SimulationData(
            dt=timestep,
            observables=physical_validation.data.ObservableData(
                constant_of_motion=simulation["constant of motion"]
            ),
        )
    )
```

[6]:
```
physical_validation.integrator.convergence(simulations=simulation_list, screen=True)
```

```
-------------------------------------------------------------------
        dt        avg       rmsd      slope         ratio
                                                  dt^2       rmsd
-------------------------------------------------------------------
     0.004   -4491.08   1.63e-02  -1.76e-07         --         --
     0.002   -4491.24   4.51e-03  -1.98e-06       4.00       3.62
     0.001   -4491.24   1.36e-03  -2.55e-06       4.00       3.31
    0.0005   -4491.21   2.83e-04  -2.46e-07       4.00       4.81
   0.00025   -4491.19   1.20e-04   2.96e-07       4.00       2.35
-------------------------------------------------------------------
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[6]:
```
0.41248328620311137
```

The shifted potential correction does slightly improve the integrator convergence, moving it closer to the expected ratio. When reducing the time step, the error introduced by the non-continuous forces does, however, start to be more dominant, and the decrease of the fluctuations with the time step does not follow expectations anymore.

Finally, we can repeat the analysis using a cut-off correction which smoothly switches both the potential and the forces off.

[7]:
```
simulation_list = []
for timestep in [0.004, 0.002, 0.001, 0.0005, 0.00025]:
    # Load appropriate simulation from examples
    simulation = example_simulations.get(
        f"1000 Lennard-Jones particles, potential and force switch cut-off correction,␣
↪timestep {timestep} ps"
    )
    # Append simulation data object to the list. Since the integrator check only needs␣
↪knowledge
    # about the time step and the constant of motion, we leave the remaining fields␣
↪empty.
    simulation_list.append(
        physical_validation.data.SimulationData(
            dt=timestep,
            observables=physical_validation.data.ObservableData(
```

(continues on next page)

(continued from previous page)

```
                constant_of_motion=simulation["constant of motion"]
            ),
        )
    )
```

[8]:
```
physical_validation.integrator.convergence(simulations=simulation_list, screen=True)
```

```
-----------------------------------------------------------------------
        dt        avg       rmsd      slope         ratio
                                                   dt^2       rmsd
-----------------------------------------------------------------------
     0.004   -4335.09   1.69e-02   5.54e-07          --         --
     0.002   -4335.25   4.37e-03  -4.87e-07        4.00       3.87
     0.001   -4335.24   1.09e-03  -3.81e-08        4.00       4.02
    0.0005   -4335.22   2.77e-04  -2.66e-08        4.00       3.93
   0.00025   -4335.20   6.90e-05  -9.38e-09        4.00       4.02
-----------------------------------------------------------------------
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[8]:
```
0.03146252406582639
```

Having now eliminated both sources of numerical artefacts in the integration, the fluctuations do scale with the time step almost exactly as expected. The very low deviations give us confidence that the integrator is operating correctly.

CHAPTER

ELEVEN

CHECK ENSEMBLE OF OPENMM TEMPERATURE REPLICA EXCHANGE SIMULATIONS

Note: This notebook can be run locally by cloning the Github repository. *The notebook is located in* `doc/examples/openmm_replica_exchange.ipynb`. *The input and output files of the simulation are located in* `doc/examples/simulation_results/openMMTemperatureReplicaExchange/`. *Be aware that probabilistic quantities such as error estimates based on bootstrapping will differ when repeating the analysis.*

[1]:
```
# enable plotting in notebook
%matplotlib notebook
```

Ensemble validation is particularly useful for validating enhanced sampling methods such as temperature replica exchange (parallel tempering) molecular dynamics simulations. In replica exchange MD, configurational swaps are attempted periodically between simulations running concurrently in separate simulation cells at different temperatures. Since the acceptance criteria for exchange moves depends on the energies of each replica involved, it is critical that the energy distributions are correct at all thermodynamic states used the parallel tempering scheme.

In this example, parallel tempering simulations on a simple coarse-grained oligomer system were run using the openmmtools framework. 6 temperature states spaced logarithmically over the range of 300K to 500K, over which the oligomer undergoes a transition from a helix to unfolded state.

[2]:
```
import numpy as np
import physical_validation

import openmmtools.multistate
import simtk.unit
```

```
Warning: importing 'simtk.openmm' is deprecated.  Import 'openmm' instead.
```

For convenience, we will use native `physical_validation` units:

[3]:
```
energy_unit = simtk.unit.kilojoule_per_mole
length_unit = simtk.unit.nanometer
volume_unit = length_unit ** 3
temperature_unit = simtk.unit.kelvin
pressure_unit = simtk.unit.bar
time_unit = simtk.unit.picosecond
kb = simtk.unit.MOLAR_GAS_CONSTANT_R.value_in_unit(energy_unit / temperature_unit)
```

Create the `UnitData` object which we will use to inform `physical_validation` of our choices. Note that the conversion factors will all be `1.0`, since we're using native units. The notation here is more general though, and would allow to change any unit set in the previous cell.

[4]:
```
unit_data = physical_validation.data.UnitData(
    kb=kb,
    energy_conversion=energy_unit.conversion_factor_to(simtk.unit.kilojoule_per_mole),
    length_conversion=length_unit.conversion_factor_to(simtk.unit.nanometer),
    volume_conversion=volume_unit.conversion_factor_to(simtk.unit.nanometer ** 3),
    temperature_conversion=temperature_unit.conversion_factor_to(simtk.unit.kelvin),
    pressure_conversion=pressure_unit.conversion_factor_to(simtk.unit.bar),
    time_conversion=time_unit.conversion_factor_to(simtk.unit.picosecond),
    energy_str=energy_unit.get_symbol(),
    length_str=length_unit.get_symbol(),
    volume_str=volume_unit.get_symbol(),
    temperature_str=temperature_unit.get_symbol(),
    pressure_str=pressure_unit.get_symbol(),
    time_str=time_unit.get_symbol(),
)
```

We first read the replica exchange output file and extract the thermodynamics states used in the simulation. We will use these states to create the `EnsembleData` objects which inform `physical_validation` of the sampled ensembles. Note that the example simulations were performed at constant temperature in a non-periodic box. We will therefore set the (undefined) volume of the NVT ensemble to `-1`.

[5]:
```
output_data = "simulation_results/openMMTemperatureReplicaExchange/output/output.nc"
reporter = openmmtools.multistate.MultiStateReporter(output_data, open_mode="r")

states = reporter.read_thermodynamic_states()[0]
num_states = len(states)
ensemble_data = [
    physical_validation.data.EnsembleData(
        ensemble="NVT",
        natoms=state.n_particles,
        volume=-1,
        temperature=state.temperature.value_in_unit(temperature_unit),
    )
    for state in states
]
```

```
Warning: The openmmtools.multistate API is experimental and may change in future releases
```

We will then read the replica energies and the state indices. Note that

- `replica_state_indices[replica, step]` denotes the thermodynamics state index sampled by replica `replica` during step `step`
- `replica_energies[replica, state, step]` is the reduced potential of replica `replica` at state `state` during step `step`

[6]:
```
analyzer = openmmtools.multistate.ReplicaExchangeAnalyzer(reporter)
replica_energies, _, _, replica_state_indices = analyzer.read_energies()
```

```
Warning: The openmmtools.multistate API is experimental and may change in future releases
```

For our analysis, we are only interested in the energies at the states the replicas were sampling at, rather than at all states. It will also make our remaining analysis easier if these energies are organized as time series per state rather than time series per replica.

Also, the replica energies are stored in reduced potential. For our analysis, we are interested in the non-reduced form,

which we can obtain by multiplying the result by kT of the respective thermodynamic state.

[7]:
```
# Prepare array of kT values
kT = np.array(
    [kb * state.temperature.value_in_unit(temperature_unit) for state in states]
)

total_steps = replica_energies.shape[2]
potential_energies = []

for state in range(num_states):
    state_energy = np.zeros(total_steps)
    for step in range(total_steps):
        # Find the replica which sampled at state `state` during step `step`
        state_energy[step] = replica_energies[
            np.nonzero(replica_state_indices[:, step] == state), state, step
        ]
    # Append non-reduced potential energy time series
    potential_energies.append(state_energy * kT[state])
```

We now have all the required information to create a `SimulationData` object for each separate state:

[8]:
```
simulation_data = []
for ensemble, potential_energy in zip(ensemble_data, potential_energies):
    simulation_data.append(
        physical_validation.data.SimulationData(
            units=unit_data,
            ensemble=ensemble,
            observables=physical_validation.data.ObservableData(
                potential_energy=potential_energy
            ),
        )
    )
```

We can now run the ensemble validation on all adjacent temperature pairs:

[9]:
```
for simulation_lower, simulation_upper in zip(
    simulation_data[:-1], simulation_data[1:]
):
    physical_validation.ensemble.check(
        data_sim_one=simulation_lower, data_sim_two=simulation_upper, screen=True
    )
```

```
After equilibration, decorrelation and tail pruning, 4.20% (126 frames) of original␣
↪Trajectory 1 remain.
After equilibration, decorrelation and tail pruning, 16.33% (490 frames) of original␣
↪Trajectory 2 remain.
Overlap is 95.2% of trajectory 1 and 91.8% of trajectory 2.
Rule of thumb estimates that dT = 58.0 would be optimal (currently, dT = 32.3)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
================================================
Maximum Likelihood Analysis (analytical error)
```

(continues on next page)

(continued from previous page)

```
================================================
Free energy
    5.40765 +/- 0.60277
Estimated slope                  |  True slope
    0.038921  +/- 0.005296       |  0.038936
    (0.00 quantiles from true slope)
Estimated dT                     |  True dT
    32.3   +/- 4.4               |  32.3
================================================
After equilibration, decorrelation and tail pruning, 16.33% (490 frames) of original␣
↪Trajectory 1 remain.
After equilibration, decorrelation and tail pruning, 15.93% (478 frames) of original␣
↪Trajectory 2 remain.
Overlap is 95.7% of trajectory 1 and 92.5% of trajectory 2.
Rule of thumb estimates that dT = 63.9 would be optimal (currently, dT = 35.7)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
================================================
Maximum Likelihood Analysis (analytical error)
================================================
Free energy
    2.36236 +/- 0.21589
Estimated slope                  |  True slope
    0.035230  +/- 0.002913       |  0.035155
    (0.03 quantiles from true slope)
Estimated dT                     |  True dT
    35.8   +/- 3.0               |  35.7
================================================
After equilibration, decorrelation and tail pruning, 15.93% (478 frames) of original␣
↪Trajectory 1 remain.
After equilibration, decorrelation and tail pruning, 9.20% (276 frames) of original␣
↪Trajectory 2 remain.
Overlap is 96.2% of trajectory 1 and 93.1% of trajectory 2.
Rule of thumb estimates that dT = 71.6 would be optimal (currently, dT = 39.6)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
================================================
Maximum Likelihood Analysis (analytical error)
================================================
Free energy
    0.34585 +/- 0.14426
Estimated slope                  |  True slope
    0.030943  +/- 0.003609       |  0.031740
    (0.22 quantiles from true slope)
Estimated dT                     |  True dT
    38.6   +/- 4.5               |  39.6
================================================
After equilibration, decorrelation and tail pruning, 9.20% (276 frames) of original␣
↪Trajectory 1 remain.
```

(continues on next page)

(continued from previous page)

```
After equilibration, decorrelation and tail pruning, 6.30% (189 frames) of original␣
↪Trajectory 2 remain.
Overlap is 81.2% of trajectory 1 and 97.4% of trajectory 2.
Rule of thumb estimates that dT = 81.0 would be optimal (currently, dT = 43.8)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
==================================================
Maximum Likelihood Analysis (analytical error)
==================================================
Free energy
    -0.79533 +/- 0.14068
Estimated slope                  |  True slope
    0.030647  +/- 0.003797       |  0.028658
    (0.52 quantiles from true slope)
Estimated dT                     |  True dT
    46.9   +/- 5.8               |  43.8
==================================================
After equilibration, decorrelation and tail pruning, 6.30% (189 frames) of original␣
↪Trajectory 1 remain.
After equilibration, decorrelation and tail pruning, 7.00% (210 frames) of original␣
↪Trajectory 2 remain.
Overlap is 92.6% of trajectory 1 and 100.0% of trajectory 2.
Rule of thumb estimates that dT = 96.4 would be optimal (currently, dT = 48.6)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
==================================================
Maximum Likelihood Analysis (analytical error)
==================================================
Free energy
    -0.96215 +/- 0.23485
Estimated slope                  |  True slope
    0.019539  +/- 0.003492       |  0.025875
    (1.81 quantiles from true slope)
Estimated dT                     |  True dT
    36.7   +/- 6.6               |  48.6
==================================================
```

In this analysis, we observe satisfactory dependence of the ensemble distribution on the temperature. As a rule of thumb, if the true interval is not within about 2-3 standard deviations of the estimated interval, the trajectory is unlikely to have been sampled from the expected ensemble. All intervals are below 2 quantiles from the true slope in the maximum likelihood analysis, the first four are around or below 0.5 quantiles from the true slope.

CHAPTER

TWELVE

CREATION OF `SIMULATIONDATA` OBJECTS

The data of simulations to be validated need to be represented by objects of the *SimulationData* type. The *SimulationData* objects are consisting of information about the simulation and the system. This information is collected in objects of different classes, namely

- *SimulationData.units* of type *UnitData*: Information on the units used by the simulation program.
- *SimulationData.ensemble* of type *EnsembleData*: Information describing the sampled ensemble.
- *SimulationData.system* of type *SystemData*: Information on the system (numbers of atoms, molecules, constraints, etc.).
- *SimulationData.observables* of type *ObservableData*: Trajectories of observables along the simulation.
- *SimulationData.trajectory* of type *TrajectoryData*: Position / velocity / force trajectories along the simulation.
- *SimulationData.dt* of type `float`: The time step at which the simulation was performed.

The *SimulationData* objects can either be constructed directly from arrays and numbers, or (partially) automatically via parsers.

12.1 Create SimulationData objects from python data

Example usage, system of 900 water molecules in GROMACS units simulated in NVT:

```
import numpy as np
import physical_validation

simulation_data = physical_validation.data.SimulationData()

num_molecules = 900
simulation_data.system = physical_validation.data.SystemData(
    # Each water molecule has three atoms
    natoms=num_molecules * 3,
    # Each molecule has three constraints
    nconstraints=num_molecules * 3,
    # In this simulation, translational center of mass motion was removed
    ndof_reduction_tra=3,
    # Rotational center of mass motion was not removed
    ndof_reduction_rot=0,
    # Repeat weight of one oxygen and two hydrogen atoms 900 times
    mass=np.tile([15.9994, 1.008, 1.008], num_molecules),
```

(continues on next page)

(continued from previous page)

```
    # Denotes the first atom of each molecules: [0, 3, 6, ...]
    molecule_idx=np.linspace(0, num_molecules * 3, num_molecules, endpoint=False,␣
↪dtype=int),
    # Each molecule has three constraints
    nconstraints_per_molecule=3 * np.ones(num_molecules),
)

# Set GROMACS units
simulation_data.units = physical_validation.data.UnitData.units("GROMACS")

# Simulation was performed under NVT conditions
simulation_data.ensemble = physical_validation.data.EnsembleData(
    ensemble='NVT',
    natoms=num_molecules * 3,
    volume=3.01125 ** 3,
    temperature=298.15,
)

# This snippet is assuming that `kin_ene`, `pot_ene` and `tot_ene` are lists
# or numpy arrays filled with the time series of kinetic, potential and total energy
# of a simulation run. These might be obtained, e.g., from the python
# API of a simulation code, or from other python-based analysis tools.
simulation_data.observables = physical_validation.data.ObservableData(
    kinetic_energy=kin_ene,
    potential_energy=pot_ene,
    total_energy=tot_ene,
)

# We are further assuming that `positions` and `velocities` are arrays
# of shape (number of frames) x (number of atoms) x 3, where the last
# number stands for the 3 spatial dimensions. Again, these arrays would
# most likely have been obtained from a python interface of the simulation
# package or from other python-based analysis tools
simulation_data.trajectory = physical_validation.data.TrajectoryData(
    position=positions,
    velocity=velocities,
)
```

12.2 Package-specific instructions

12.2.1 GROMACS

GROMACS does not offer a well-established Python interface to read out energies or trajectories. `physical_validation` therefore offers a parser, which will return a fully populated *SimulationData* object by reading in GROMACS input and output files.

The *GromacsParser* takes the GROMACS input files `mdp` (run options) and `top` (topology file) to read the details about the system, the ensemble and the time step. The observable trajectory is extracted from an `edr` (binary energy trajectory), while the position and velocity trajectory can be read either from a `trr` (binary trajectory) or a `gro` (ASCII trajectory) file. The constructor optionally takes the path to a gromacs binary as well as the path to the topology library as inputs. The first is necessary to extract information from binary files (using `gmx energy` and `gmx dump`), while the

second becomes necessary if the `top` file contains `#include` statements which usually rely on GROMACS environment variables. The parser is able to find GROMACS installations which are in the path (e.g. after sourcing the `GMXRC` file) and the corresponding topology library automatically.

Example usage:

```
import physical_validation

parser = physical_validation.data.GromacsParser()

res = parser.get_simulation_data(
     mdp='mdout.mdp',
     top='system.top',
     gro='system.gro',
     edr='system.edr'
)
```

Note: Always double-check the results received from the automatic parser. Since this is not an official GROMACS tool, it is very likely that some special cases or changes in recent versions might not be interpreted correctly.

12.2.2 LAMMPS

To analyze simulations performed with LAMMPS, we strongly suggest using its Python interface Pizza.py to create a SimulationData object as explained in *Create SimulationData objects from python data*. Note that `UnitData` offers access to a UnitData object representing the LAMMPS `real` units by using `.data.UnitData.units("LAMMPS real")`.

As an alternative, `physical_validation` ships with a LAMMPS parser, which tries to read part of the system information, the observable and position / velocity trajectories from LAMMPS output files.

Example usage:

```
import physical_validation

parser = physical_validation.data.LammpsParser()

res = parser.get_simulation_data(
    # The LAMMPS parser cannot infer the ensemble from the LAMMPS files,
    # so we pass an EnsembleData object with the information matching the simulation
    ensemble=physical_validation.data.EnsembleData(
        ensemble="NVT",
        natoms=900,
        volume=20**3,
        temperature=300
    ),
    in_file=dir_1 + '/water.in',
    log_file=dir_1 + '/log.lammps',
    data_file=dir_1 + '/water.lmp',
    dump_file=dir_1 + '/dump.atom'
)
```

Warning: The LAMMPS parser is in an early development stage. It is part of the `physical_validation` package in the hope that it is helpful to someone, but it is very likely to go wrong in a number of cases. Please check any object data create by the LAMMPS parser carefully.

12.3 Flatfile parser

For MD packages not supported by the package-specific parsers, it is possible to create the *SimulationData* objects via the *FlatfileParser*. This parser fills the *SimulationData.trajectory* object via 3-dimensional ASCII files containing the position and velocity trajectories, and the *SimulationData.observables* via 1-dimensional ASCII files containing the trajectories for the observables of interest. As the details on the units, the simulated system and the sampled ensemble can not easily be read from such files, this information has to be provided by the user by passing objects of the respective data structures. See *FlatfileParser.get_simulation_data* for more details on the *SimulationData* creation via the flat file parser, and *Data contained in SimulationData objects* for details on which test requires which information.

Example usage, system of 900 water molecules in GROMACS units simulated in NVT (note that this example leaves some fields in *SystemData* empty, as well as the trajectory of some observables and the position and velocities):

```
import physical_validation as pv

parser = pv.data.FlatfileParser()

system = pv.data.SystemData(
    natoms=900*3,
    nconstraints=900*3,
    ndof_reduction_tra=3,
    ndof_reduction_rot=0
)

# We need to specify the units in which the simulation was performed,
# specifically the value of k_B in the used energy units, the conversion
# factor of the simulation units to the physical validation units
# (*_conversion keywords), and a string representation of the simulation
# units (*_str keywords, used for output only).
# See documentation below about UnitData object for more details.
units = pv.data.UnitData(
    kb=8.314462435405199e-3,
    energy_str='kJ/mol',
    energy_conversion=1.0,
    length_str='nm',
    length_conversion=1.0,
    volume_str='nm^3',
    volume_conversion=1.0,
    temperature_str='K',
    temperature_conversion=1.0,
    pressure_str='bar',
    pressure_conversion=1.0,
    time_str='ps',
    time_conversion=1.0
)
```

(continues on next page)

(continued from previous page)

```
ensemble = pv.data.EnsembleData(
    ensemble='NVT',
    natoms=900*3,
    volume=3.01125**3,
    temperature=298.15
)

res = parser.get_simulation_data(
    units=units, ensemble=ensemble, system=system,
    kinetic_ene_file='kinetic.dat',
    potential_ene_file='potential.dat',
    total_ene_file='total.dat'
)
```

12.4 Additional examples

12.4.1 Use `MDAnalysis` to create mass vector

Using `MDAnalysis`, creating a mass vector which can be fed to *`SystemData.mass`* is straightforward. See the following snippet for an example using a GROMACS topology:

```
import MDAnalysis as mda
import numpy as np

u = mda.Universe('system.gro')
mass=np.array([u.atoms[i].mass for i in range(len(u.atoms))])
```

12.4.2 Use `MDAnalysis` to define molecule groups for equipartition testing

`physical_validation.kinetic_energy.equipartition()` allows to specify molecule groups which can be tested for equipartition. The segments used in `MDAnalysis` can easily be used to define molecule groups as input to the equipartition check:

```
import MDAnalysis as mda
import numpy as np

u = mda.Universe('system.tpr', 'system.gro')
molec_groups = []
for i in range(len(u.segments)):
    seg = u.segments[i]
    molec_groups.append(np.array([seg.atoms[j].index for j in range(len(seg.atoms))]))
```

12.4.3 Use `MDAnalysis` to read position and velocity trajectory

`MDAnalysis` also makes it easy to create *`TrajectoryData`* objects which require position and velocity trajectories as inputs. Given a `Universe` object which contains a trajectory, we can simply use a list comprehension to create a full trajectory in memory:

```
import MDAnalysis as mda
import numpy as np
import physical_validation

u = mda.Universe('system.tpr', 'system.trr')
trajectory = physical_validation.data.TrajectoryData(
    position=[frame.positions for frame in u.trajectory],
    velocity=[frame.velocities for frame in u.trajectory])
```

We can also use the atom selector to only feed part of the trajectory to the `physical_validation` tests. This is useful if we want to analyze the equipartition of parts of the system only (e.g. the solute) which can massively speed up the validation check. Note that we have to adapt the *`SystemData`* object accordingly to inform `physical_validation` that we are only analyzing part of the system.

```
import MDAnalysis as mda
import numpy as np
import physical_validation

u = mda.Universe('system.tpr', 'system.trr')
protein = u.select_atoms('protein')
trajectory = physical_validation.data.TrajectoryData(
    position=[protein.positions for _ in u.trajectory],
    velocity=[protein.velocities for _ in u.trajectory])
```

Note: `MDAnalysis` uses Å (ångström) as a length unit. Don't forget to choose the *`UnitData`* accordingly!

CHAPTER

THIRTEEN

DATA CONTAINED IN `SIMULATIONDATA` OBJECTS

13.1 Units: `SimulationData.units` of type `UnitData`

Attributes:

- *UnitData.kb*, `float`
- *UnitData.energy_conversion*, `float`
- *UnitData.length_conversion*, `float`
- *UnitData.volume_conversion*, `float`
- *UnitData.temperature_conversion*, `float`
- *UnitData.pressure_conversion*, `float`
- *UnitData.time_conversion*, `float`
- *UnitData.energy_str*, `str`
- *UnitData.length_str*, `str`
- *UnitData.volume_str*, `str`
- *UnitData.temperature_str*, `str`
- *UnitData.pressure_str*, `str`
- *UnitData.time_str*, `str`

The information about units consists of different parts:

- The value of k_B in the used energy units,
- the conversion factor to `physical_validation` units (kJ/mol, nm, nm^3, K, bar, ps, the same as the GROMACS default units), and
- the name of the units (`energy_str`, `length_str`, `volume_str`, `temperature_str`, `pressure_str`, `time_str`).

The names are only used for output (console printing and plotting), and are optional. The conversion factors and kB are, on the other hand, used in computations and need to be given. To avoid silent errors, these keywords to not have defaults and must be specified.

Needed by

- *physical_validation.ensemble.check()*
- *physical_validation.ensemble.estimate_interval()*
- *physical_validation.kinetic_energy.distribution()*, only

- *UnitData.kb*

13.2 Ensemble: SimulationData.ensemble of type EnsembleData

Attributes:

- *EnsembleData.ensemble*, str
- *EnsembleData.natoms*, int
- *EnsembleData.mu*, float
- *EnsembleData.volume*, float
- *EnsembleData.pressure*, float
- *EnsembleData.energy*, float
- *EnsembleData.temperature*, float

The ensemble is a string indicating the thermodynamical ensemble a simulation was performed in, and is any of :code:'NVE', :code:'NVT', :code:'NPT', :code:'muVT'.

Depending on the ensemble, *EnsembleData* then holds additional information defining the ensemble, such as the number of particles N, the chemical potential mu, the volume V, the pressure P, the constant energy E or the temperature T. While any of these additional information are technically optional, most of them are needed by certain tests, such that not fully defining the ensemble results in warnings. The notable exception to this rule is the constant energy E for NVE, which is not needed by any test and can hence be omitted without raising a warning.

Needed by

- *physical_validation.kinetic_energy.distribution()*
- *physical_validation.ensemble.check()*

13.3 System: SimulationData.system of type SystemData

Attributes:

- *SystemData.natoms*, the total number of atoms in the system; e.g. for a system containing 100 water molecules: `system_data.natoms = 300`
- *SystemData.nconstraints*, the total number of constraints in the system, not including the global translational and rotational constraints (see next two attributes); e.g. for a system containing 100 *rigid* water molecules: `system_data.nconstraints = 300`
- *SystemData.ndof_reduction_tra*, global reduction of translational degrees of freedom (e.g. due to constraining the center of mass of the system)
- *SystemData.ndof_reduction_rot*, global reduction of rotational degrees of freedom (e.g. due to constraining the center of mass of the system)
- *SystemData.mass*, a list of the mass of every atom in the system; e.g. for a single water molecule: `system_data.mass = [15.9994, 1.008, 1.008]`
- *SystemData.molecule_idx*, a list of the index of the first atom of every molecule (this assumes that the atoms are sorted by molecule); e.g. for a system containing 3 water molecules: `system_data.molecule_idx = [0, 3, 6]`

- *SystemData.nconstraints_per_molecule*, a list of the number of constraints in every molecule; e.g. for a system containing 3 *rigid* water molecules: `system_data.nconstraints_per_molecule = [3, 3, 3]`
- *SystemData.bonds*, a list containing all bonds in the system; e.g. for a system containing 3 water molecules: `system_data.bonds = [[0, 1], [0, 2], [3, 4], [3, 5], [6, 7], [6, 8]]`
- *SystemData.constrained_bonds*, a list containing only the constrained bonds in the system, must be a subset of *SystemData.bonds* (and equal, if all bonds are constrained).

Todo: Currently, there is some redundancy in the attributes listed above. The *SystemData.bonds* and *SystemData.constrained_bonds* are reserved for future use - included already in the information about the system, but not yet used by any tests included in the currently published package. In a future version, the *SystemData* should be streamlined to make the object initialization easier.

Needed by

- *physical_validation.kinetic_energy.distribution()*, partially:
  - *SystemData.natoms*,
  - *SystemData.nconstraints*,
  - *SystemData.ndof_reduction_tra*,
  - *SystemData.ndof_reduction_rot*
- *physical_validation.kinetic_energy.equipartition()*, all attributes except *SystemData.bonds* and *SystemData.constrained_bonds*.

13.4 Observables: `SimulationData.observables` of type `ObservableData`

Attributes:

- *ObservableData.kinetic_energy*, the kinetic energy trajectory (nframes x 1), also accessible via `observable_data['kinetic_energy']`
- *ObservableData.potential_energy*, the potential energy trajectory (nframes x 1), also accessible via `observable_data['potential_energy']`
- *ObservableData.total_energy*, the total energy trajectory (nframes x 1), also accessible via `observable_data['total_energy']`
- *ObservableData.volume*, the volume trajectory (nframes x 1), also accessible via `observable_data['volume']`
- *ObservableData.pressure* the pressure trajectory (nframes x 1), also accessible via `observable_data['pressure']`
- *ObservableData.temperature* the temperature trajectory (nframes x 1), also accessible via `observable_data['temperature']`
- *ObservableData.constant_of_motion* the constant of motion trajectory (nframes x 1), also accessible via `observable_data['constant_of_motion']`
- *ObservableData.number_of_species* the trajectory of the number of molecules of a species, used for muVT, (nframes x num_species), also accessible via `observable_data['number_of_species']`

Needed by

- *physical_validation.kinetic_energy.distribution()*
  - *ObservableData.kinetic_energy*
- *physical_validation.ensemble.check()*
  - *ObservableData.total_energy*, or
  - *ObservableData.potential_energy*,
  - *ObservableData.volume* (for NPT),
  - *ObservableData.number_of_species* (for muVT)
- *physical_validation.integrator.convergence()*
  - *ObservableData.constant_of_motion*

13.5 Atom trajectories: `SimulationData.trajectory` of type `TrajectoryData`

Attributes:

- *TrajectoryData.position*, the position trajectory (nframes x natoms x 3), also accessible via `trajectory_data['position']`
- *TrajectoryData.velocity*, the velocity trajectory (nframes x natoms x 3), also accessible via `trajectory_data['velocity']`

Needed by

- *physical_validation.kinetic_energy.equipartition()*

13.6 Time step: `SimulationData.dt` of type `float`

The timestep used during the simulation run, a single `float` value.

Needed by

- *physical_validation.integrator.convergence()*

CHAPTER

FOURTEEN

PHYSICAL_VALIDATION PACKAGE

Physical validation suite for MD simulations

14.1 physical_validation.kinetic_energy module

The *kinetic_energy* module is part of the physical_validation package, and consists of checks of the kinetic energy distribution and its equipartition.

physical_validation.kinetic_energy.**distribution**(*data:* SimulationData, *strict: bool = False*, *verbosity: int = 2*, *screen: bool = False*, *filename: Optional[str] = None*, *bs_repetitions: int = 200*, *bootstrap_seed: Optional[int] = None*, *data_is_uncorrelated: bool = False*) → Union[float, Tuple[float, float]]

Checks the distribution of a kinetic energy trajectory.

Parameters

- **data** – Simulation data object
- **strict** – If True, check full kinetic energy distribution via K-S test. Otherwise, check mean and width of kinetic energy distribution. Default: False
- **verbosity** – Verbosity level, where 0 is quiet and 3 shows full details. Default: 2.
- **screen** – Plot distributions on screen. Default: False.
- **filename** – Plot distributions to *filename*. Default: None, no plotting to file.
- **bs_repetitions** – Number of bootstrap samples used for error estimate (if strict=False). Default: 200.
- **bootstrap_seed** – Sets the random number seed for bootstrapping (if strict=False). If set, bootstrapping will be reproducible. Default: *None*, bootstrapping is non-reproducible.
- **data_is_uncorrelated** – Whether the provided data is uncorrelated. If this option is set, the equilibration, decorrelation and tail pruning of the trajectory is skipped. This can speed up the analysis, but note that if the provided data is correlated, the results of the physical validation checks might be invalid.

Returns

If *strict=True*: The p value of the test. If *strict=False*: Distance of the estimated T(mu) and T(sigma) from the expected temperature, measured in standard deviations of the respective estimate.

Return type

result

Notes

Non-strict test

If *strict = False* (the default), this function will estimate the mean and the standard deviation of the data. Analytically, a gamma distribution with shape $k = N/2$ (with N the number of degrees of freedom) and scale $\theta = k_B T$ (with T the target temperature) is expected. The mean and the standard deviation of a gamma distribution are given by $\mu = k\theta$ and $\sigma = \sqrt{k}\theta$.

The standard error of the mean and standard deviation are estimated via bootstrap resampling. The function prints the analytically expected mean and variance as well as the fitted values and their error estimates. It also prints T(mu) and T(sigma), which are defined as the temperatures to which the estimated mean and standard deviation correspond, given the number of degrees of freedom N in the system:

$$T(\mu') = \frac{2\mu'}{N k_B}$$

$$T(\sigma') = \frac{\sqrt{2}\sigma'}{\sqrt{N} k_B}$$

The return value is a tuple containing the distance of the estimated T(mu) and T(sigma) from the expected temperature, measured in standard deviations of the respective estimates.

Strict test

If *strict = True*, this function tests the hypothesis that a sample of kinetic energies is Maxwell-Boltzmann distributed given a specific target temperature and the number of degrees of freedom in the system,

$$P(K) \sim K^{N/2-1} e^{-\beta K} .$$

The test is performed using the Kolmogorov-Smirnov test provided by scipy.stats.kstest. It returns the p-value, measuring the likelihood that a sample at least as extreme as the one given is originating from the expected distribution.

Note: The Kolmogorov-Smirnov test is known to have two weaknesses.

1. The test is more sensitive towards deviations around the center of the distribution than at its tails. We deem this to be acceptable for most MD applications, but be wary if yours is sensible to the kinetic distribution tails.
2. The test is not valid if its parameters are guessed from the data set. Using the target temperature of the MD simulation as an input is therefore perfectly valid, but using the average temperature over the trajectory as an input to the test can potentially invalidate it.

---

physical_validation.kinetic_energy.**equipartition**(*data: SimulationData*, *strict: bool = False*, *molec_groups: Optional[List[ndarray]] = None*, *random_divisions: int = 0*, *random_groups: int = 0*, *random_division_seed: Optional[int] = None*, *verbosity: int = 2*, *screen: bool = False*, *filename: Optional[str] = None*, *bootstrap_seed: Optional[int] = None*, *data_is_uncorrelated: bool = False*) → Union[List[float], List[Tuple[float, float]]]

Checks the equipartition of a simulation trajectory.

Parameters

- **data** – Simulation data object

- **strict** – If True, check full kinetic energy distribution via K-S test. Otherwise, check mean and width of kinetic energy distribution. Default: False
- **molec_groups** – List of 1d arrays containing molecule indices defining groups. Useful to pre-define groups of molecules (e.g. solute / solvent, liquid mixture species, ...). If None, no pre-defined molecule groups will be tested. Default: None.

  Note: If an empty 1d array is found as last element in the list, the remaining molecules are collected in this array. This allows, for example, to only specify the solute, and indicate the solvent by giving an empty array.
- **random_divisions** – Number of random division tests attempted. Default: 0 (random division tests off).
- **random_groups** – Number of groups the system is randomly divided in. Default: 2.
- **random_division_seed** – Seed making the random divisions reproducible. Default: None, random divisions not reproducible
- **verbosity** – Verbosity level, where 0 is quiet and 3 very chatty. Default: 2.
- **screen** – Plot distributions on screen. Default: False.
- **filename** – Plot distributions to *filename*. Default: None, no plotting to file
- **bootstrap_seed** – Sets the random number seed for bootstrapping (if strict=False). If set, bootstrapping will be reproducible. Default: None, bootstrapping is non-reproducible.
- **data_is_uncorrelated** – Whether the provided data is uncorrelated. If this option is set, the equilibration, decorrelation and tail pruning of the trajectory is skipped. This can speed up the analysis, but note that if the provided data is correlated, the results of the physical validation checks might be invalid.

Returns

If *strict=True*: The p value for every tests. If *strict=False*: Distance of the estimated T(mu) and T(sigma) from the expected temperature, measured in standard deviations of the respective estimate, for every test.

Return type

result

Notes

This function compares the kinetic energy between groups of degrees of freedom. Theoretically, the kinetic energy is expected (via the equipartition theorem) to be equally distributed over all degrees of freedom. In practice, deviations of temperature between groups of degrees of freedom up to several degrees K are routinely observed. Larger deviations can, however, hint to misbehaving simulations, such as, e.g., frozen degrees of freedom, lack of energy exchange between degrees of freedom, and transfer of heat from faster to slower degrees of freedom.

Splitting of degrees of freedom is done both on a sub-molecular and on a molecular level. On a sub-molecular level, the degrees of freedom of a molecule can be partitioned into rigid-body contributions (translation of the center-of-mass, rotation around the center-of-mass) and intra-molecular contributions. On a molecular level, the single molecules of the system can be divided in groups, either by function (solute / solvent, different species of liquid mixtures, ...) or randomly.

check_equipartition() partitions the kinetic energy of the entire system and, optionally, of predefined or randomly separated groups. It then computes either the mean and the standard deviation of each partition and compares them to the theoretically expected value (*strict=True*, the default), or it performs a Kolmogorov-Smirnov test of the distribution. See physical_validation.kinetic_energy.distribution for more detail on the checks.

14.2 physical_validation.ensemble module

This software allows users to perform statistical test to determine if a given molecular simulation is consistent with the thermodynamic ensemble it is performed in.

Users should cite the JCTC paper: Shirts, M. R. "Simple Quantitative Tests to Validate Sampling from Thermodynamic Ensembles", J. Chem. Theory Comput., 2013, 9 (2), pp 909-926, https://dx.doi.org/10.1021/ct300688p

physical_validation.ensemble.**check**(*data_sim_one: SimulationData*, *data_sim_two: SimulationData*, *total_energy: bool = False*, *bootstrap_error: bool = False*, *bootstrap_repetitions: int = 200*, *bootstrap_seed: Optional[int] = None*, *screen: bool = False*, *filename: Optional[str] = None*, *verbosity: int = 1*, *data_is_uncorrelated: bool = False*) → List[float]

Check the ensemble. The correct check is inferred from the simulation data given.

Parameters

- **data_sim_one** – Simulation data object of first simulation
- **data_sim_two** – Simulation data object of second simulation differing in its state point from the first
- **total_energy** – Whether to use the total energy for the calculation Default: False, use potential energy only
- **bootstrap_error** – Calculate the standard error via bootstrap resampling Default: False
- **bootstrap_repetitions** – Number of bootstrap repetitions drawn Default: 200
- **bootstrap_seed** – Sets the random number seed for bootstrapping. If set, bootstrapping will be reproducible. Default: None, bootstrapping is non-reproducible.
- **screen** – Plot distributions on screen. Default: False.
- **filename** – Plot distributions to *filename*. Default: *None*, no plotting to file.
- **verbosity** – Level of verbosity, from 0 (quiet) to 3 (very verbose). Default: 1
- **data_is_uncorrelated** – Whether the provided data is uncorrelated. If this option is set, the equilibration, decorrelation and tail pruning of the trajectory is skipped. This can speed up the analysis, but note that if the provided data is correlated, the results of the physical validation checks might be invalid.

Returns
The number of quantiles the computed result is off the analytical one.

Return type
quantiles

physical_validation.ensemble.**estimate_interval**(*data: SimulationData*, *verbosity: int = 1*, *total_energy: bool = False*, *data_is_uncorrelated: bool = False*) → Dict[str, float]

In order to perform an ensemble check, two simulations at distinct state point are needed. Choosing two state points too far apart will result in poor or zero overlap between the distributions, leading to very noisy results (due to sample errors in the tails) or a breakdown of the method, respectively. Choosing two state points very close to each others, on the other hand, makes it difficult to distinguish the slope from statistical error in the samples.

This function implements a rule of thumb based on the standard deviations of distributions. It takes a single simulation and suggests appropriate intervals for a second simulation to be used for ensemble checking.

Parameters

- **data** – The performed simulation.
- **verbosity** – If 0, no output is printed on screen. If 1, estimated intervals are printed. If larger, additional information during calculation are printed. Default: 1
- **total_energy** – Use total energy instead of potential energy only. Default: False
- **data_is_uncorrelated** – Whether the provided data is uncorrelated. If this option is set, the equilibration, decorrelation and tail pruning of the trajectory is skipped. This can speed up the analysis, but note that if the provided data is correlated, the results of the physical validation checks might be invalid. Default: False

Returns

If *data* was performed under NVT conditions, *intervals* contains only one entry:

- *'dT'*, containing the suggested temperature interval.

If *data* was performed under NPT conditions, *intervals* contains three entries:

- *'dT'*: Suggested temperature interval at constant pressure
- *'dP'*: Suggested pressure interval at constant temperature
- *'dTdP'*: Suggested combined temperature and pressure interval

Return type
intervals

14.3 physical_validation.integrator module

The *integrator.convergence* module is part of the physical_validation package, and consists of checks of the convergence of the MD integrator.

physical_validation.integrator.**convergence**(*simulations: List[SimulationData]*, *convergence_test: str = 'max_deviation'*, *verbose: bool = True*, *screen: bool = False*, *filename: Optional[str] = None*) → float

Compares the convergence of the fluctuations of conserved quantities with decreasing simulation time step to theoretical expectations.

Parameters

- **simulations** – The (otherwise identical) simulations performed using different time steps
- **convergence_test** – A function defining the convergence test. Currently, only one test is implemented: *max_deviation*, which is chosen by default
- **verbose** – If True, print more detailed output. Default: False.
- **screen** – Plot convergence on screen. Default: False.
- **filename** – Plot convergence to *filename*. Default: None, no plotting to file.

Return type
The largest deviation from the expected ratio of fluctuations.

Notes

For a simplectic integration algorithm, the fluctuations δE of a constant of motion E (such as, for example, the total energy in a NVE simulations) are theoretically expected to scale like the squared timestep of the integration. When comparing two otherwise identical simulations performed at different time step Δt, the following equality is hence expected to hold:

$$\frac{\Delta t_1^2}{\Delta t_2^2} = \frac{\delta E_1}{\delta E_2}$$

This function calculates the ratio of the fluctuation for simulations performed at different timesteps and compares it to the analytically expected value. If the deviation is larger than *tol*, the test is considered failed.

CHAPTER

FIFTEEN

PHYSICAL_VALIDATION.DATA SUBPACKAGE

15.1 physical_validation.data.simulation_data module

Data structures carrying simulation data.

class physical_validation.data.simulation_data.**SimulationData**(*units=None*, *dt=None*, *system=None*, *ensemble=None*, *observables=None*, *trajectory=None*)

SimulationData: System information and simulation results

The SimulationData class holds both the information on the system and the results of a simulation run of that system. SimulationData contains all information on a simulation run needed by the physical validation tests. SimulationData objects can either be created directly by calling the class constructor, or by using a parser returning a SimulationData object.

static compatible(*data_1*, *data_2*) → bool

Checks whether two simulations are compatible for common validation.

Parameters

- **data_1** (SimulationData) –
- **data_2** (SimulationData) –

Returns
result

Return type
bool

property ensemble: *EnsembleData*

Information on the sampled ensemble

Returns
ensemble

Return type
EnsembleData

Type
EnsembleData

property units: *UnitData*

Information on the sampled units

Returns
units

Return type
UnitData

Type
UnitsData

property observables: *ObservableData*

Observables collected during the simulation

Returns
observables

Return type
ObservableData

Type
ObservableData

property trajectory: *TrajectoryData*

Trajectories collected during the simulation

Returns
trajectory

Return type
TrajectoryData

Type
TrajectoryData

property system: *SystemData*

Information on the system's system

Returns
system

Return type
SystemData

Type
SystemData

property dt: float

The timestep of the simulation run.

Returns
timestep

Return type
float

set_ensemble(*ensemble: str*, *natoms: Optional[float] = None*, *mu: Optional[float] = None*, *volume: Optional[float] = None*, *pressure: Optional[float] = None*, *energy: Optional[float] = None*, *temperature: Optional[float] = None*) → None

raise_if_units_are_none(*test_name: str*, *argument_name: str*) → None

Raise if the unit data was not set

Parameters

- **test_name** – String naming the test used for error output
- **argument_name** – String naming the SimulationData argument used for error output

raise_if_ensemble_is_invalid(*test_name: str*, *argument_name: str*, *check_pressure: bool*, *check_mu: bool*) → None

Raise InputError if the ensemble data does not hold the required information needed for the tests.

Parameters

- **test_name** – String naming the test used for error output
- **argument_name** – String naming the SimulationData argument used for error output
- **check_pressure** – Whether to check if the pressure is defined (NPT only).
- **check_mu** – Whether to check if the chemical potential is defined (muVT only).

raise_if_system_data_is_invalid(*test_name: str*, *argument_name: str*, *check_full_system_data_only: bool*) → None

Raise InputError if the ensemble data does not hold the required information needed for the tests.

Parameters

- **test_name** – String naming the test used for error output
- **argument_name** – String naming the SimulationData argument used for error output
- **check_full_system_data_only** – Whether to check the full system data only (number of atoms in the system, number of constraints in the system, number of translational and rotational degree of freedom reduction of the system). If false, this also checks the masses per atom and the molecule index and constraints.

raise_if_observable_data_is_invalid(*required_observables: List[str]*, *test_name: str*, *argument_name: str*) → None

Raise InputError if there are missing observables entries, or if the required observables don't have the same number of frames

Parameters

- **required_observables** – List of strings denoting the observables required
- **test_name** – String naming the test used for error output
- **argument_name** – String naming the SimulationData argument used for error output

raise_if_trajectory_data_is_invalid(*test_name: str*, *argument_name: str*) → None

Raise InputError if there are missing observables entries, or if the required observables don't have the same number of frames

Parameters

- **test_name** – String naming the test used for error output
- **argument_name** – String naming the SimulationData argument used for error output

15.2 physical_validation.data.unit_data module

Data structures carrying simulation data.

class physical_validation.data.unit_data.**UnitData**(*kb: float*, *energy_conversion: float*, *length_conversion: float*, *volume_conversion: float*, *temperature_conversion: float*, *pressure_conversion: float*, *time_conversion: float*, *energy_str: str = 'ENE'*, *length_str: str = 'LEN'*, *volume_str: str = 'VOL'*, *temperature_str: str = 'TEMP'*, *pressure_str: str = 'PRESS'*, *time_str: str = 'TIME'*)

UnitData: Information about the units used

The information about units consists of different parts:

- **The name of the units (energy_str, length_str, volume_str,** temperature_str, pressure_str, time_str),
- the value of kB in the used energy units, and
- the conversion factor to GROMACS units (kJ/mol, nm, nm^3, K, bar, ps).

The names are only used for output (console printing and plotting), and are optional. The conversion factors and kB are, on the other hand, used in computations and need to be given.

classmethod units(*name: Optional[str] = None*)

property kb: float

The value of the Boltzmann constant

Type
float

property energy_str: str

Energy unit

Type
str

property length_str: str

Length unit

Type
str

property volume_str: str

Volume unit

Type
str

property temperature_str: str

Temperature unit

Type
str

property pressure_str: str

Pressure unit

Type
str

property time_str: str

Time unit

Type
str

property energy_conversion: float

Energy conversion factor, 1 energy_unit = energy_conversion * kJ/mol

Type
float

property length_conversion: float

Length conversion factor, 1 length_unit = length_conversion * nm

Type
float

property volume_conversion: float

Volume conversion factor, 1 volume_unit = volume_conversion * nm^3

Type
float

property temperature_conversion: float

Temperature conversion factor, 1 temperature_unit = temperature_conversion * K

Type
float

property pressure_conversion: float

Pressure conversion factor, 1 pressure_unit = pressure_conversion * bar

Type
float

property time_conversion: float

Time conversion factor, 1 time_unit = time_conversion * ps

Type
float

15.3 physical_validation.data.ensemble_data module

Data structures carrying simulation data.

class physical_validation.data.ensemble_data.**EnsembleData**(*ensemble*, *natoms=None*, *mu=None*, *volume=None*, *pressure=None*, *energy=None*, *temperature=None*)

EnsembleData: Holds data defining the ensemble

The ensemble is a string indicating the thermodynamical ensemble a simulation was performed in, and is any of 'NVE', 'NVT', 'NPT', 'muVT'. Depending on the ensemble, EnsembleData then holds additional information

defining the ensemble, such as the number of particles N, the chemical potential mu, the volume V, the pressure P, the constant energy E or the temperature T. While any of these additional information are optional, most of them are needed by certain tests, such that not fully defining the ensemble results in warnings. The notable exception to this rule is the constant energy E for the NVE, which is not needed by any test and can hence be omitted without raising a warning.

static ensembles() → Tuple[str, str, str, str]

property ensemble: str

Get ensemble

property natoms: int

Get natoms

property mu: ndarray

Get mu

property volume: float

Get volume

property pressure: float

Get pressure

property energy: float

Get energy

property temperature: float

Get temperature

15.4 physical_validation.data.trajectory_data module

Data structures carrying simulation data.

class physical_validation.data.trajectory_data.**RectangularBox**(*box: ndarray*)

property box

gather(*positions: ndarray*, *bonds: List[List[int]]*, *molec_idx: List[int]*)

class physical_validation.data.trajectory_data.**TrajectoryData**(*position: Optional[Any] = None*, *velocity: Optional[Any] = None*)

TrajectoryData: The position and velocity trajectory along the simulation

The full trajectory is needed to calculate the equipartition of the kinetic energy. As they are used in connection, the position and velocity trajectories are expected to have the same shape and number of frames.

The position and velocity trajectories can be accessed either using the getters of an object, as in

- trajectory.position
- trajectory.velocity

or using the key notation, as in

- trajectory['position']
- trajectory['velocity']

static trajectories() → Tuple[str, str]

property position: Optional[ndarray]

Get position

property velocity: Optional[ndarray]

Get velocity

15.5 physical_validation.data.observable_data module

Data structures carrying simulation data.

class physical_validation.data.observable_data.ObservableData(*kinetic_energy: Optional[Any] = None*, *potential_energy: Optional[Any] = None*, *total_energy: Optional[Any] = None*, *volume: Optional[Any] = None*, *pressure: Optional[Any] = None*, *temperature: Optional[Any] = None*, *constant_of_motion: Optional[Any] = None*, *number_of_species: Optional[Any] = None*)

ObservableData: The trajectory of (macroscopic) observables during the simulation

Stores a number of different observables:

- kinetic_energy: the kinetic energy of the system,
- potential_energy: the potential energy of the system,
- total_energy: the total energy of the system,
- volume: the volume of the system box,
- pressure: the pressure of the system,
- temperature: the temperature of the system,
- constant_of_motion: a constant of motion of the trajectory.

The observable trajectories can be accessed either using the getters of an object, as in

observables.kinetic_energy

or using the key notation, as in

observables['kinetic_energy']

static observables() → List[str]

property kinetic_energy: Optional[ndarray]

Get kinetic_energy

property potential_energy: Optional[ndarray]

Get potential_energy

property total_energy: Optional[ndarray]

Get total_energy

property volume: Optional[ndarray]

Get volume

property pressure: Optional[ndarray]

Get pressure

property temperature: Optional[ndarray]

Get temperature

property constant_of_motion: Optional[ndarray]

Get constant_of_motion

property number_of_species: Optional[ndarray]

Get number_of_species

property nframes: Optional[int]

Get number of frames

property kinetic_energy_per_molecule: Optional[ndarray]

Get kinetic_energy per molecule - used internally

15.6 physical_validation.data.system_data module

Data structure carrying information on the simulated system.

class physical_validation.data.system_data.**SystemData**(*natoms: Optional[int] = None*, *nconstraints: Optional[float] = None*, *ndof_reduction_tra: Optional[float] = None*, *ndof_reduction_rot: Optional[float] = None*, *mass: Optional[ndarray] = None*, *molecule_idx: Optional[ndarray] = None*, *nconstraints_per_molecule: Optional[ndarray] = None*)

SystemData: Information about the atoms and molecules in the system.

The information stored in SystemData objects describes the atoms and molecules in the system as far as the physical validation tests need it.

The system is described in terms of

- natoms: the total number of atoms in the system
- nconstraints: the total number of constraints in the system
- ndof_reduction_tra: global reduction of translational degrees of freedom (e.g. due to constraining the center of mass of the system)
- ndof_reduction_rot: global reduction of rotational degrees of freedom (e.g. due to constraining the center of mass of the system)

The atoms are described in terms of

- mass: a list of the mass of every atom in the system

The molecules are described by

- molecule_idx: a list with the indices first atoms of every molecule (this assumes that the atoms are sorted by molecule)

- nconstraints_per_molecule: a list with the number of constraints in every molecule

Only used internally:

- ndof_per_molecule: a list with the number of degrees of freedom of every molecule

Reserved for future use:

- bonds
- constrained_bonds

Notes:

- kinetic_energy.distribution() only requires information on the system (natoms, nconstraints, ndof_reduction_tra, ndof_reduction_rot)
- kinetic_energy.equipartition() additionally requires information on the atoms and molecules (mass, molecule_idx, nconstraints_per_molecule)

All other tests do not require and information from SystemData.

property natoms: int

Number of atoms in the system

Type
int

property nconstraints: float

Total number of constraints in the system

Does not include the reduction of degrees of freedom in the absence of external forces.

Type
float

property ndof_reduction_tra: float

Number of translational degrees of freedom deducted from 3*[# of molecules]

Type
float

property ndof_reduction_rot: float

Number of rotational degrees of freedom deducted from 3*[# of molecules]

Type
float

property mass: ndarray

Mass vector for the atoms

Setter accepts array-like objects.

Type
nd-array

property molecule_idx: ndarray

List of index of first atom of each molecule

Setter accepts array-like objects.

Type
nd-array

property nconstraints_per_molecule: ndarray

List of number of constraints per molecule

Setter accepts array-like objects.

Type
nd-array

property ndof_per_molecule: Optional[List[Dict[str, float]]]

List of number of degrees of freedom per molecule

Type
nd-array

property bonds: List[List[int]]

List of bonds per molecule

Type
List[List[int]]

property constrained_bonds: List[List[int]]

List of constrained bonds per molecule

Type
List[List[int]]

15.7 physical_validation.data.parser module

Parsers read output files from MD simulation packages and create SimulationData objects with their contents.

class physical_validation.data.parser.**Parser**

Parser base class

get_simulation_data() → *SimulationData*

15.8 physical_validation.data.gromacs_parser module

gromacs_parser.py

class physical_validation.data.gromacs_parser.**GromacsParser**(*exe: Optional[str] = None*, *includepath: Optional[Union[str, List[str]]] = None*)

static units() → *UnitData*

get_simulation_data(*mdp: Optional[str] = None*, *top: Optional[str] = None*, *edr: Optional[str] = None*, *trr: Optional[str] = None*, *gro: Optional[str] = None*) → *SimulationData*

Parameters

- **mdp** (*str, optional*) – A string pointing to a .mdp file
- **top** (*str, optional*) – A string pointing to a .top file
- **edr** (*str, optional*) – A string pointing to a .edr file
- **trr** (*str, optional*) – A string pointing to a .trr file

- **gro** (*str, optional*) – A string pointing to a .gro file (Note: if also trr is given, gro is ignored)

Returns

result – A SimulationData filled with the results of the simulation as described by the provided GROMACS files.

Return type

SimulationData

15.9 physical_validation.data.flatfile_parser module

flatfile_parser.py

class physical_validation.data.flatfile_parser.**FlatfileParser**

get_simulation_data(*units: Optional[UnitData] = None*, *ensemble: Optional[EnsembleData] = None*, *system: Optional[SystemData] = None*, *dt: Optional[float] = None*, *position_file: Optional[str] = None*, *velocity_file: Optional[str] = None*, *kinetic_ene_file: Optional[str] = None*, *potential_ene_file: Optional[str] = None*, *total_ene_file: Optional[str] = None*, *volume_file: Optional[str] = None*, *pressure_file: Optional[str] = None*, *temperature_file: Optional[str] = None*, *const_of_mot_file: Optional[str] = None*, *number_of_species_file: Optional[str] = None*) → *SimulationData*

Read simulation data from flat files

Returns a SimulationData object created from (optionally) provided UnitData, EnsembleData and SystemData, as well as TrajectoryData and ObservableData objects created from flat files. The files are expected to be in one of the following formats:

- xyz-format trajectory files (position_file, velocity_file) - three numbers per line, separated by white space - frames delimited by a completely blank line - any character after (and including) a '#' are ignored
- 1d-format all other files - one number per line - any character after (and including) a '#' are ignored

Parameters

- **units** (UnitData, *optional*) – A UnitData object representing the units used in the simulation
- **ensemble** (EnsembleData, *optional*) – A EnsembleData object representing the ensemble the simulation has been performed in
- **system** (SystemData, *optional*) – A SystemData object representing the atoms and molecules in the system
- **dt** (*float, optional*) – The time step used in the simulation
- **position_file** (*str, optional*) – Path to a file in xyz-format containing the position trajectory
- **velocity_file** (*str, optional*) – Path to a file in xyz-format containing the velocity trajectory
- **kinetic_ene_file** (*str, optional*) – Path to a file in 1d-format containing the kinetic energy trajectory

- **potential_ene_file** (*str, optional*) – Path to a file in 1d-format containing the potential energy trajectory
- **total_ene_file** (*str, optional*) – Path to a file in 1d-format containing the total energy trajectory
- **volume_file** (*str, optional*) – Path to a file in 1d-format containing the volume trajectory
- **pressure_file** (*str, optional*) – Path to a file in 1d-format containing the pressure trajectory
- **temperature_file** (*str, optional*) – Path to a file in 1d-format containing the temperature trajectory
- **const_of_mot_file** (*str, optional*) – Path to a file in 1d-format containing the constant of motion trajectory
- **number_of_species_file** (*str, optional*) – Path to a file in nd-format containing the number of species trajectory

Returns

result – A SimulationData filled with the provided ensemble and system objects as well as the trajectory data found in the edr and trr / gro files.

Return type

SimulationData

CHAPTER

SIXTEEN

PHYSICAL_VALIDATION.UTIL SUBPACKAGE

Warning: This subpackage is intended for internal use. Documentation and input validation is reduced.

16.1 physical_validation.util.kinetic_energy module

This module contains low-level functionality of the *physical_validation.kinetic_energy* module. The functions in this module should generally not be called directly. Please use the high-level functions from *physical_validation.kinetic energy*.

physical_validation.util.kinetic_energy.**is_close**(*value1: float*, *value2: float*, *relative_tolerance: float = 1e-09*, *absolute_tolerance: float = 1e-09*) → bool

Whether two float values are close, as defined by a given relative and absolute tolerance.

physical_validation.util.kinetic_energy.**check_distribution**(*kin: ndarray*, *temp: float*, *ndof: float*, *kb: float*, *verbosity: int*, *screen: bool*, *filename: Optional[str]*, *ene_unit: Optional[str]*, *temp_unit: Optional[str]*, *data_is_uncorrelated: bool*) → float

Checks if a kinetic energy trajectory is Maxwell-Boltzmann distributed.

Parameters

- **kin** – Kinetic energy snapshots of the system.
- **temp** – Target temperature of the system. Used to construct the Maxwell-Boltzmann distribution.
- **ndof** – Number of degrees of freedom in the system. Used to construct the Maxwell-Boltzmann distribution.
- **kb** – Boltzmann constant k_B.
- **verbosity** – 0: Silent. 1: Print minimal information. 2: Print result details. 3: Print additional information.
- **screen** – Plot distributions on screen.
- **filename** – Plot distributions to *filename*. If *None*, no plotting.
- **ene_unit** – Energy unit - used for output only.
- **temp_unit** – Temperature unit - used for output only.

- **data_is_uncorrelated** – Whether the provided data is uncorrelated. If this option is set, the equilibration, decorrelation and tail pruning of the trajectory is skipped. This can speed up the analysis, but note that if the provided data is correlated, the results of the physical validation checks might be invalid.

Returns
The p value of the test.

Return type
result

See also:

physical_validation.kinetic_energy.distribution
High-level version

physical_validation.util.kinetic_energy.**check_mean_std**(*kin: ndarray*, *temp: float*, *ndof: float*, *kb: float*, *verbosity: int*, *bs_repetitions: int*, *bootstrap_seed: Optional[int]*, *screen: bool*, *filename: Optional[str]*, *ene_unit: Optional[str]*, *temp_unit: Optional[str]*, *data_is_uncorrelated: bool*) → Tuple[float, float]

Calculates the mean and standard deviation of a trajectory (+ bootstrap error estimates), and compares them to the theoretically expected values.

Parameters

- **kin** – Kinetic energy snapshots of the system.
- **temp** – Target temperature of the system. Used to construct the Maxwell-Boltzmann distribution.
- **ndof** – Number of degrees of freedom in the system. Used to construct the Maxwell-Boltzmann distribution.
- **kb** – Boltzmann constant k_B.
- **verbosity** – 0: Silent. 1: Print minimal information. 2: Print result details. 3: Print additional information.
- **bs_repetitions** – Number of bootstrap samples used for error estimate.
- **bootstrap_seed** – Sets the random number seed for bootstrapping. If set, bootstrapping will be reproducible. If *None*, bootstrapping is non-reproducible.
- **screen** – Plot distributions on screen.
- **filename** – Plot distributions to *filename*. If *None*, no plotting.
- **ene_unit** – Energy unit - used for output only.
- **temp_unit** – Temperature unit - used for output only.
- **data_is_uncorrelated** – Whether the provided data is uncorrelated. If this option is set, the equilibration, decorrelation and tail pruning of the trajectory is skipped. This can speed up the analysis, but note that if the provided data is correlated, the results of the physical validation checks might be invalid.

Returns
Distance of the estimated T(mu) and T(sigma) from the expected temperature, measured in standard deviations of the estimates.

Return type
result

See also:

physical_validation.kinetic_energy.distribution
High-level version

physical_validation.util.kinetic_energy.**check_equipartition**(*positions: ndarray*, *velocities: ndarray*, *masses: ndarray*, *molec_idx: ndarray*, *molec_nbonds: ndarray*, *natoms: int*, *nmolecs: int*, *temp: float*, *kb: float*, *strict: bool*, *ndof_reduction_tra: float*, *ndof_reduction_rot: float*, *molec_groups: Optional[List[ndarray]]*, *random_divisions: int*, *random_groups: int*, *random_division_seed: Optional[int]*, *ndof_molec: Optional[List[Dict[str, float]]]*, *kin_molec: Optional[List[List[Dict[str, float]]]]*, *verbosity: int*, *screen: bool*, *filename: Optional[str]*, *ene_unit: Optional[str]*, *temp_unit: Optional[str]*, *bootstrap_seed: Optional[int]*, *data_is_uncorrelated: bool*) → Tuple[Union[List[float], List[Tuple[float, float]]], List[Dict[str, float]], List[List[Dict[str, float]]]]

Checks the equipartition of a simulation trajectory.

Parameters

- **positions** (*array-like (nframes x natoms x 3)*) – 3d array containing the positions of all atoms for all frames
- **velocities** (*array-like (nframes x natoms x 3)*) – 3d array containing the velocities of all atoms for all frames
- **masses** (*array-like (natoms x 1)*) – 1d array containing the masses of all atoms
- **molec_idx** (*array-like (nmolecs x 1)*) – Index of first atom for every molecule
- **molec_nbonds** (*array-like (nmolecs x 1)*) – Number of bonds for every molecule
- **natoms** – Number of atoms in the system
- **nmolecs** – Number of molecules in the system
- **temp** – Target temperature of the simulation.
- **kb** – Boltzmann constant k_B.
- **strict** – If True, check full kinetic energy distribution via K-S test. Otherwise, check mean and width of kinetic energy distribution.
- **ndof_reduction_tra** – Number of center-of-mass translational degrees of freedom to remove.

- **ndof_reduction_rot** – Number of center-of-mass rotational degrees of freedom to remove.
- **molec_groups** (*List[array-like] (ngroups x ?), optional*) – List of 1d arrays containing molecule indeces defining groups. Useful to pre-define groups of molecules (e.g. solute / solvent, liquid mixture species, ...). If None, no pre-defined molecule groups will be tested.

  Note: If an empty 1d array is found as last element in the list, the remaining molecules are collected in this array. This allows, for example, to only specify the solute, and indicate the solvent by giving an empty array.
- **random_divisions** – Number of random division tests attempted.
- **random_groups** – Number of groups the system is randomly divided in.
- **random_division_seed** – Seed making the random divisions reproducible. If *None*, random divisions not reproducible
- **ndof_molec** – Pass in the degrees of freedom per molecule. Slightly increases speed of repeated analysis of the same simulation run.
- **kin_molec** – Pass in the kinetic energy per molecule. Greatly increases speed of repeated analysis of the same simulation run.
- **verbosity** – Verbosity level, where 0 is quiet and 3 very chatty.
- **screen** – Plot distributions on screen. Default: False.
- **filename** – Plot distributions to *filename*. If *None*, no plotting.
- **ene_unit** – Energy unit - used for output only.
- **temp_unit** – Temperature unit - used for output only.
- **bootstrap_seed** – Sets the random number seed for bootstrapping (if strict=False). If set, bootstrapping will be reproducible. If *None*, bootstrapping is non-reproducible.
- **data_is_uncorrelated** – Whether the provided data is uncorrelated. If this option is set, the equilibration, decorrelation and tail pruning of the trajectory is skipped. This can speed up the analysis, but note that if the provided data is correlated, the results of the physical validation checks might be invalid.

Returns

- **result** (*List[float] or List[Tuple[float]]*) – If *strict=True*: The p value for every test. If *strict=False*: Distance of the estimated T(mu) and T(sigma) from the expected temperature, measured in standard deviations of the respective estimate, for every test.
- **ndof_molec** (*List[dict]*) – List of the degrees of freedom per molecule. Can be saved to increase speed of repeated analysis of the same simulation run.
- **kin_molec** (*List[List[dict]]*) – List of the kinetic energy per molecule per frame. Can be saved to increase speed of repeated analysis of the same simulation run.

See also:

physical_validation.kinetic_energy.check_equipartition
High-level version

physical_validation.util.kinetic_energy.**calc_ndof**(*natoms: int*, *nmolecs: int*, *molec_idx: ndarray*, *molec_nbonds: ndarray*, *ndof_reduction_tra: float*, *ndof_reduction_rot: float*) → List[Dict[str, float]]

Calculates the total / translational / rotational & internal / rotational / internal degrees of freedom per molecule.

Parameters

- **natoms** – Total number of atoms in the system
- **nmolecs** – Total number of molecules in the system
- **molec_idx** – Index of first atom for every molecule
- **molec_nbonds** – Number of bonds for every molecule
- **ndof_reduction_tra** – Number of center-of-mass translational degrees of freedom tobremove.
- **ndof_reduction_rot** – Number of center-of-mass rotational degrees of freedom to remove.

Returns

List of dictionaries containing the degrees of freedom for each molecule Keys: [‘total’, ‘translational’, ‘rotational and internal’, ‘rotational’, ‘internal’]

Return type

ndof_molec

physical_validation.util.kinetic_energy.**calc_molec_kinetic_energy**(*pos: ndarray*, *vel: ndarray*, *masses: ndarray*, *molec_idx: ndarray*, *natoms: int*, *nmolecs: int*) → Dict[str, ndarray]

Calculates the total / translational / rotational & internal / rotational / internal kinetic energy per molecule.

Parameters

- **pos** (*nd-array (natoms x 3)*) – 2d array containing the positions of all atoms
- **vel** (*nd-array (natoms x 3)*) – 2d array containing the velocities of all atoms
- **masses** (*nd-array (natoms x 1)*) – 1d array containing the masses of all atoms
- **molec_idx** (*nd-array (nmolecs x 1)*) – Index of first atom for every molecule
- **natoms** – Total number of atoms in the system
- **nmolecs** – Total number of molecules in the system

Returns

Dictionary of lists containing the kinetic energies for each molecule Keys: [‘total’, ‘translational’, ‘rotational and internal’, ‘rotational’, ‘internal’]

Return type

kin

physical_validation.util.kinetic_energy.**group_kinetic_energy**(*kin_molec: Dict[str, ndarray]*, *nmolecs: int*, *molec_group=typing.Optional[typing.Iterable[int]]*) → Dict[str, float]

Sums up the partitioned kinetic energy for a given group or the entire system.

Parameters

- **kin_molec** – Partitioned kinetic energies per molecule.
- **nmolecs** – Total number of molecules in the system.

- **molec_group** – Indices of the group to be summed up. *None* defaults to all molecules in the system.

Returns
kin – Dictionary of partitioned kinetic energy for the group.

Return type
dict

physical_validation.util.kinetic_energy.**group_ndof**(*ndof_molec: List[Dict[str, float]]*, *nmolecs: int*, *molec_group=typing.Optional[typing.Iterable[int]]*) → Dict[str, float]

Sums up the partitioned degrees of freedom for a given group or the entire system.

Parameters

- **ndof_molec** – Partitioned degrees of freedom per molecule.
- **nmolecs** – Total number of molecules in the system.
- **molec_group** – Indeces of the group to be summed up. None defaults to all molecules in the system.

Returns
ndof – Dictionary of partitioned degrees of freedom for the group.

Return type
dict

physical_validation.util.kinetic_energy.**test_group**(*kin_molec: List[Dict[str, ndarray]]*, *ndof_molec: List[Dict[str, float]]*, *nmolecs: int*, *temp: float*, *kb: float*, *dict_keys: List[str]*, *strict: bool*, *group: Optional[Iterable[int]]*, *verbosity: int*, *screen: bool*, *filename: Optional[str]*, *ene_unit: Optional[str]*, *temp_unit: Optional[str]*, *bootstrap_seed: Optional[int]*, *data_is_uncorrelated: bool*) → Union[List[float], List[Tuple[float, float]]]

Tests if the partitioned kinetic energy trajectory of a group (or, if group is None, of the entire system) are separately Maxwell-Boltzmann distributed.

Parameters

- **kin_molec** – Partitioned kinetic energies per molecule for every frame.
- **ndof_molec** – Partitioned degrees of freedom per molecule.
- **nmolecs** – Total number of molecules in the system.
- **temp** – Target temperature of the simulation.
- **kb** – Boltzmann constant k_B.
- **dict_keys** – List of dictionary keys representing the partitions of the degrees of freedom.
- **strict** – If True, check full kinetic energy distribution via K-S test. Otherwise, check mean and width of kinetic energy distribution.
- **group** – Indices of the group to be tested. *None* defaults to all molecules in the system.
- **verbosity** – Verbosity level, where 0 is quiet and 3 very chatty.
- **screen** – Plot distributions on screen.

- **filename** – Plot distributions to *filename*. If *None*, no plotting.
- **ene_unit** – Energy unit - used for output only.
- **temp_unit** – Temperature unit - used for output only.
- **bootstrap_seed** – Sets the random number seed for bootstrapping (if strict=False). If set, bootstrapping will be reproducible. If *None*, bootstrapping is non-reproducible.
- **data_is_uncorrelated** – Whether the provided data is uncorrelated. If this option is set, the equilibration, decorrelation and tail pruning of the trajectory is skipped. This can speed up the analysis, but note that if the provided data is correlated, the results of the physical validation checks might be invalid.

Returns
p value for every partition (strict) or tuple of distance of estimated T(mu) and T(sigma) for every partition (non-strict)

Return type
result

16.2 physical_validation.util.ensemble module

This file reimplements most functionality of the checkensemble.py code originally published on https://github.com/shirtsgroup/checkensemble. It serves as the low-level functionality of the high-level module *physical_validation.ensemble*.

physical_validation.util.ensemble.**pymbar_bar**(*work_forward: ndarray*, *work_backward: ndarray*) → Dict[str, float]

Wrapper around the pymbar BAR functionality, allowing the remaining code to be agnostic of the pymbar version used.

Parameters
- **work_forward** (*Array of forward work values*) –
- **work_backward** (*Array of backward work values*) –

Return type
A dictionary containing the free energy estimate and its error estimate

physical_validation.util.ensemble.**chemical_potential_energy**(*chemical_potential: ndarray*, *number_of_species: ndarray*) → ndarray

Calculates the chemical potential energy of a trajectory by returning the sum of the current number of species multiplied by the chemical potential.

physical_validation.util.ensemble.**generate_histograms**(*traj1: ndarray*, *traj2: ndarray*, *g1: float*, *g2: float*, *bins: ndarray*) → Tuple[ndarray, ndarray, ndarray, ndarray]

physical_validation.util.ensemble.**do_linear_fit**(*traj1: ndarray*, *traj2: ndarray*, *g1: float*, *g2: float*, *bins: ndarray*, *screen: bool*, *filename: Optional[str]*, *trueslope: float*, *trueoffset: float*, *units: Optional[str]*, *xlabel: str*, *ylabel: str*) → Tuple[ndarray, ndarray]

physical_validation.util.ensemble.**do_max_likelihood_fit**(*traj1: ndarray*, *traj2: ndarray*, *g1: Union[float, ndarray]*, *g2: Union[float, ndarray]*, *init_params: ndarray*, *verbose: bool*) → Tuple[ndarray, ndarray]

physical_validation.util.ensemble.**checkensemble_solver**(*fun*, *x0*, *args*, *jac*, *hess*, *tol=1e-10*, *maxiter=20*) → OptimizeResult

physical_validation.util.ensemble.**check_bins**(*traj1: ndarray*, *traj2: ndarray*, *bins: ndarray*) → ndarray

physical_validation.util.ensemble.**print_stats**(*title: str*, *fitvals: ndarray*, *dfitvals: Optional[ndarray]*, *kb: float*, *param1: Union[float, ndarray]*, *param2: Union[float, ndarray]*, *trueslope: Union[float, ndarray]*, *temp: Optional[float]*, *pvconvert: Optional[float]*, *dtemp: bool*, *dpress: bool*, *dmu: bool*, *dtempdpress: bool*, *dtempdmu: bool*) → None

physical_validation.util.ensemble.**estimate_interval**(*ens_string: str*, *ens_temp: float*, *energy: ndarray*, *kb: float*, *ens_press: Optional[float]*, *volume: Optional[ndarray]*, *pvconvert: Optional[float]*, *ens_mu: Optional[ndarray]*, *species_number: Optional[ndarray]*, *verbosity: int*, *cutoff: float*, *tunit: str*, *punit: str*, *munit: str*, *data_is_uncorrelated: bool*) → Dict[str, Union[float, List[float]]]

physical_validation.util.ensemble.**check_1d**(*traj1: ndarray*, *traj2: ndarray*, *param1: float*, *param2: float*, *kb: float*, *quantity: str*, *dtemp: bool*, *dpress: bool*, *dmu: bool*, *temp: Optional[float]*, *pvconvert: Optional[float]*, *nbins: int*, *cutoff: float*, *bootstrap_seed: Optional[int]*, *bootstrap_error: bool*, *bootstrap_repetitions: int*, *verbosity: int*, *screen: bool*, *filename: str*, *xlabel: str*, *xunit: Optional[str]*, *data_is_uncorrelated: bool*) → List[float]

Checks whether the energy trajectories of two simulation performed at different temperatures have sampled distributions at the analytically expected ratio.

Parameters

- **traj1** – Trajectory of the first simulation If dtemp:
  - NVT: Potential energy U or total energy E = U + K
  - NPT: Enthalpy H = U + pV or total energy E = H + K

  If dpress:
  - NPT: Volume V
- **traj2** – Trajectory of the second simulation If dtemp:
  - NVT: Potential energy U or total energy E = U + K
  - NPT: Enthalpy H = U + pV or total energy E = H + K

  If dpress:
  - NPT: Volume V
- **param1** – Target temperature or pressure of the first simulation
- **param2** – Target temperature or pressure of the second simulation

- **kb** – Boltzmann constant in same units as the energy trajectories
- **quantity** – Name of quantity analyzed (used for printing only)
- **dtemp** – Set to True if trajectories were simulated at different temperature
- **dpress** – Set to True if trajectories were simulated at different pressure
- **temp** – The temperature in equal temperature, differing pressure NPT simulations. Needed to print optimal dP.
- **pvconvert** – Conversion from pressure * volume to energy units. Needed to print optimal dP.
- **dmu** – Set to True if trajectories were simulated at different chemical potential
- **nbins** – Number of bins used to assess distributions of the trajectories
- **cutoff** – Tail cutoff of distributions.
- **bootstrap_seed** – Sets the random number seed for bootstrapping. If set, bootstrapping will be reproducible. If *None*, bootstrapping is non-reproducible.
- **bootstrap_error** – Calculate the standard error via bootstrap resampling
- **bootstrap_repetitions** – Number of bootstrap repetitions drawn
- **verbosity** – Verbosity level.
- **screen** – Plot distributions on screen.
- **filename** – Plot distributions to *filename*. If *None*, no plotting.
- **xlabel** – x-axis label used for plotting
- **xunit** – x-axis label unit used for plotting
- **data_is_uncorrelated** – Whether the provided data is uncorrelated. If this option is set, the equilibration, decorrelation and tail pruning of the trajectory is skipped. This can speed up the analysis, but note that if the provided data is correlated, the results of the physical validation checks might be invalid.

Return type

The number of quantiles the computed result is off the analytical one.

physical_validation.util.ensemble.**check_2d**(*traj1: ndarray*, *traj2: ndarray*, *param1: ndarray*, *param2: ndarray*, *kb: float*, *pvconvert: Optional[float]*, *quantity: List[str]*, *dtempdpress: bool*, *dtempdmu: bool*, *cutoff: float*, *bootstrap_seed: Optional[int]*, *bootstrap_error: bool*, *bootstrap_repetitions: int*, *verbosity: int*, *screen: bool*, *filename: Optional[str]*, *data_is_uncorrelated: bool*) → List[float]

Checks whether the energy trajectories of two simulation performed at different temperatures have sampled distributions at the analytically expected ratio.

Parameters

- **traj1** – Trajectory of the first simulation If dtempdpress:
  - traj[0,:]: Potential energy U or total energy E = U + K
  - traj[1,:]: Volume V
- **traj2** – Trajectory of the second simulation If dtempdpress:
  - traj[0,:]: Potential energy U or total energy E = U + K

  - traj[1,:]: Volume V

- **param1** –

  If dtempdpress:
  Target temperature and pressure of the first simulation

- **param2** –

  If dtempdpress:
  Target temperature and pressure of the first simulation

- **kb** – Boltzmann constant in same units as the energy trajectories
- **pvconvert** – Conversion from pressure * volume to energy units
- **quantity** – Names of quantities analyzed (used for printing only)
- **dtempdpress** – Set to True if trajectories were simulated at different temperature and pressure
- **dtempdmu** – Set to True if trajectories were simulated at different temperature and chemical potential
- **cutoff** – Tail cutoff of distributions.
- **bootstrap_seed** – Sets the random number seed for bootstrapping. If set, bootstrapping will be reproducible. If *None*, bootstrapping is non-reproducible.
- **bootstrap_error** – Calculate the standard error via bootstrap resampling
- **bootstrap_repetitions** – Number of bootstrap repetitions drawn
- **verbosity** – Verbosity level.
- **screen** – Plot distributions on screen.
- **filename** – Plot distributions to *filename*. If *None*, no plotting.
- **data_is_uncorrelated** – Whether the provided data is uncorrelated. If this option is set, the equilibration, decorrelation and tail pruning of the trajectory is skipped. This can speed up the analysis, but note that if the provided data is correlated, the results of the physical validation checks might be invalid. Default: False

Return type
The number of quantiles the computed result is off the analytical one.

16.3 physical_validation.util.integrator module

This module contains low-level functionality of the *physical_validation.integrator* module. The functions in this module should generally not be called directly. Please use the high-level functions from *physical_validation.integrator*.

physical_validation.util.integrator.**calculate_rmsd**(*data: ndarray*, *time: Optional[ndarray]*) → Tuple[float, float, float]

physical_validation.util.integrator.**max_deviation**(*dts: ndarray*, *rmsds: ndarray*) → float

physical_validation.util.integrator.**check_convergence**(*const_traj: Dict[str, ndarray]*, *convergence_test: Callable[[ndarray, ndarray], float]*, *verbose: bool*, *screen: bool*, *filename: Optional[str]*) → float

16.4 physical_validation.util.trajectory module

physical_validation.util.trajectory.**equilibrate**(*traj: ndarray*) → ndarray

physical_validation.util.trajectory.**decorrelate**(*traj: ndarray*) → ndarray

physical_validation.util.trajectory.**cut_tails**(*traj: ndarray*, *cut: float*) → ndarray

physical_validation.util.trajectory.**prepare**(*traj: ndarray*, *cut: Optional[float] = None*, *verbosity: int = 1*, *name: Optional[str] = None*, *skip_preparation: bool = False*) → ndarray

physical_validation.util.trajectory.**overlap**(*traj1: ndarray*, *traj2: ndarray*, *cut=None*) → Tuple[ndarray, ndarray, Optional[float], Optional[float]]

physical_validation.util.trajectory.**bootstrap**(*traj: ndarray*, *n_samples: int*) → Iterator[ndarray]

16.5 physical_validation.util.plot module

physical_validation.util.plot.**plot**(*res: List[Dict[str, Union[ndarray, str, Dict]]]*, *legend: Optional[str] = None*, *title: Optional[str] = None*, *xlabel: Optional[str] = None*, *ylabel: Optional[str] = None*, *xlim: Optional[Tuple[float, float]] = None*, *ylim: Optional[Tuple[float, float]] = None*, *inv_x: bool = False*, *inv_y: bool = False*, *sci_x: bool = False*, *sci_y: bool = False*, *axtext: Optional[str] = None*, *annotation_location: Optional[str] = None*, *percent: bool = False*, *filename: Optional[str] = None*, *screen: bool = True*) → None

16.6 physical_validation.util.error module

Module containing the custom exception classes for the physical_validation package.

exception physical_validation.util.error.**PhysicalValidationError**

Base class for exceptions in the physical_validation module.

exception physical_validation.util.error.**InputError**(*argument*, *message*)

Exception raised for input errors

exception physical_validation.util.error.**FileFormatError**(*argument*, *message*)

Exception raised for files not following expected format

16.7 physical_validation.util.gromacs_interface module

GROMACS python interface.

Warning: This is a mere place holder, as an official python API is currently being developed by the gromacs development team. It is probably neither especially elegant nor especially safe. Use of this module in any remotely critical application is strongly discouraged.

class physical_validation.util.gromacs_interface.**GromacsInterface**(*exe: Optional[str] = None*, *dp: bool = False*, *includepath: Optional[Union[str, List[str]]] = None*)

property exe: str

exe is a string pointing to the gmx executable.

property double: bool

double is a bool defining whether the simulation was ran at double precision

property includepath: List[str]

includepath defines a path the parser looks for system files

get_quantities(*edr: str*, *quantities: List[str]*, *cwd: Optional[str] = None*, *begin: Optional[float] = None*, *end: Optional[float] = None*, *args: Optional[List[str]] = None*) → Dict[str, ndarray]

read_trr(*trr: str*) → Dict[str, Optional[Union[ndarray, *RectangularBox*]]]

static read_gro(*gro: str*) → Dict[str, Optional[Union[ndarray, *RectangularBox*]]]

static read_mdp(*mdp: str*) → Dict[str, str]

static write_mdp(*options: Dict[str, str]*, *mdp: str*)

read_system_from_top(*top: str*, *define: Optional[str] = None*, *include: Optional[str] = None*) → List[Dict[str, Any]]

grompp(*mdp: str*, *top: str*, *gro: str*, *tpr: Optional[str] = None*, *cwd: str = '.'*, *args: Optional[List[str]] = None*, *stdin: Union[None, int, IO[Any]] = None*, *stdout: Union[None, int, IO[Any]] = None*, *stderr: Union[None, int, IO[Any]] = None*) → int

mdrun(*tpr: str*, *edr: Optional[str] = None*, *deffnm: Optional[str] = None*, *cwd: str = '.'*, *args: Optional[List[str]] = None*, *stdin: Union[None, int, IO[Any]] = None*, *stdout: Union[None, int, IO[Any]] = None*, *stderr: Union[None, int, IO[Any]] = None*, *mpicmd: Optional[str] = None*) → int

CHAPTER

SEVENTEEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Merz2018] Merz PT, Shirts MR (2018) "Testing for physical validity in molecular simulations", PLOS ONE 13(9): e0202764. https://doi.org/10.1371/journal.pone.0202764

[Shirts2013] Shirts, M.R. "Simple Quantitative Tests to Validate Sampling from Thermodynamic Ensembles", J. Chem. Theory Comput., 2013, 9 (2), pp 909–926, http://dx.doi.org/10.1021/ct300688p

PYTHON MODULE INDEX

p

INDEX

O

P

R

S

T

U

V

W