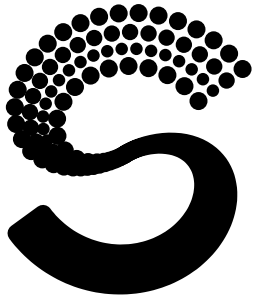


STACIE

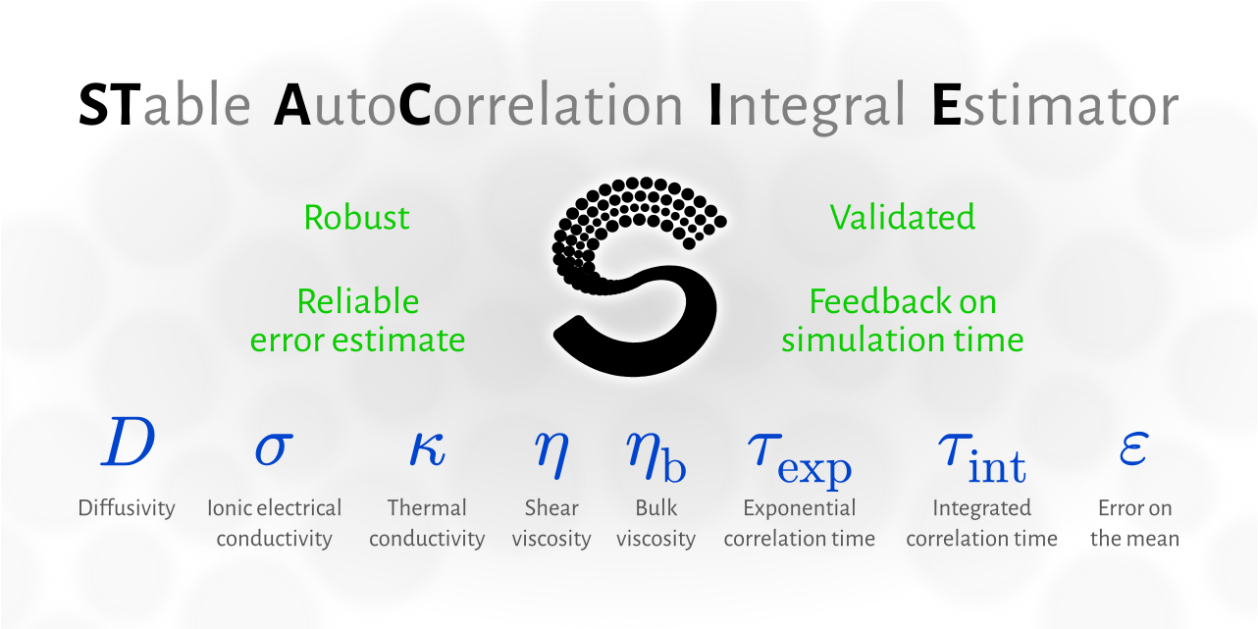
Release 1.2

Gözdenur Toraman, Toon Verstraelen

Jan 27, 2026

Contents

STACIE is a Python package and algorithm that computes time integrals of autocorrelation functions. It is primarily designed for post-processing molecular dynamics simulations. However, it can also be used for more general analysis of time-correlated data. Typical applications include estimating transport properties and the uncertainty of averages over time-correlated data, as well as analyzing characteristic timescales.



STACIE is developed in the context of a collaboration between the Center for Molecular Modeling and the tribology group of Labo Soete at Ghent University. STACIE is open-source software (LGPL-v3 license) and is available on GitHub and PyPI.

This is a PDF version of the online documentation of STACIE. The latest version of the documentation can be found at https://molmod.github.io/stacie/.

Please cite the following in any publication that relies on STACIE:

> Gözdenur Toraman, Dieter Fauconnier, and Toon Verstraelen "STable AutoCorrelation Integral Estimator (STACIE): Robust and accurate transport properties from molecular dynamics simulations" *Journal of Chemical Information and Modeling* **Article ASAP** 2025, 65 (19), 10445–10464, doi:10.1021/acs.jcim.5c01475, arXiv:2506.20438.

A follow-up paper is nearly completed that will describe in detail the calculation of shear viscosity with STACIE:

> Toraman, G.; Fauconnier, D.; Verstraelen, T. "Reliable Viscosity Calculation from High-Pressure Equilibrium Molecular Dynamics: Case Study of 2,2,4-Trimethylhexane.", in preparation.

In addition, we are preparing another follow-up paper showing how to estimate diffusion coefficients with proper uncertainty quantification using STACIE, which is currently not fully documented yet.

Copy-pasteable citation records in various formats are provided in *How to Cite*.

CHAPTER 1

Getting Started

STACIE is a Python software library that can be used interactively in a Jupyter Notebook or embedded non-interactively in larger computational workflows.

To get started:

- Install the `stacie` Python library (see *installation*).
- Understand and accept the *open source licenses* we use.
- Get a bird's-eye view of *how to use* STACIE.
- Know how to *cite* STACIE when using it for your research.

A basic understanding of the *theory* is highly recommended to ensure that you get the right result for the right reason. The documentation contains several *worked examples* that you can use as a starting point for your own research.

1.1 Installation

Before you begin, ensure that you have the following installed:

- Python version 3.10 or higher
- Pip

Additional dependencies will be installed using the `pip` command below. Familiarity with Pip is assumed. We recommend performing the installation within a Python virtual environment.

To install STACIE in a Python virtual environment, use the following shell command:

```
pip install stacie
```

Alternatively, you can install STACIE in a Conda environment from the `conda-forge` channel:

```
conda install -c conda-forge stacie
```

1.2 License

STACIE is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

STACIE is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

STACIE's documentation is located in the `docs/` directory of its source tree and files under this directory are distributed under a choice of license: either the Creative Commons Attribution-ShareAlike 4.0 International license (CC BY-SA 4.0) or the GNU Lesser General Public License, version 3 or later (LGPL-v3+). The SPDX License Expression for the documentation is `CC-BY-SA-4.0 OR LGPL-3.0-or-later`.

You should have received a copy of the CC BY-SA 4.0 and LGPL-v3+ licenses along with the source code. If not, see:

- https://creativecommons.org/licenses/by-sa/4.0/
- https://www.gnu.org/licenses/

1.3 Usage Overview

This section provides an overview of how to use STACIE. More detailed information can be found in the remaining sections of the documentation.

The STACIE algorithm provides robust and reliable estimates of autocorrelation integrals without requiring extensive adjustment of its settings. Users simply provide the relevant inputs to STACIE: the time-correlated sequences in the form of a NumPy array, a few physical parameters (such as the time step), and a model to fit to the spectrum. This can be done in a Jupyter notebook for interactive work or in a Python script.

The most important inputs for STACIE are time series data on an equidistant time grid. You can provide multiple independent sequences of the same length to reduce uncertainties. The analysis returns a `result` object including the following attributes:

- `acint`: The integral of the autocorrelation function.
- `corrtime_int`: The *integrated* autocorrelation time.
- `corrtime_exp`: The *exponential* autocorrelation time (if supported by the selected model).

The estimated uncertainties are accessible through the `acint_std`, `corrtime_int_std`, and `corrtime_exp_std` attributes, respectively. In addition, intermediate results of the analysis can be accessed, e.g., to create plots using the built-in plotting functions.

Many (transport) properties are defined in terms of an autocorrelation integral. They require slightly different settings and preprocessing of the input data. STACIE's documentation contains instructions for *the properties we have tested*. In addition, we provide *worked examples* that show in detail how STACIE is used in practice.

If you plan to produce publication-quality research with STACIE, the analysis inevitably becomes an iterative process. The main difficulty is providing sufficient data for the analysis, but what constitutes "sufficient" only becomes clear after an initial analysis. STACIE's documentation contains a section on *preparing inputs* to help you with this process.

Finally, we encourage you to delve into the *theory* behind STACIE. Although we try to make STACIE usable without a full understanding of the technical details, a good understanding will help you get the most out of it.

1.4 How to Cite

When using STACIE in your research, please cite the STACIE paper in any resulting publication. The reference is provided in several formats below:

1.4.1 Main STACIE Paper

> Gözdenur Toraman, Dieter Fauconnier, and Toon Verstraelen "STable AutoCorrelation Integral Estimator (STACIE): Robust and accurate transport properties from molecular dynamics simulations" *Journal of Chemical Information and Modeling* **Article ASAP** 2025, 65 (19), 10445–10464, doi:10.1021/acs.jcim.5c01475, arXiv:2506.20438.

This paper introduces STACIE and should be cited in any publication that relies on STACIE. The manuscript has been submitted to The Journal of Chemical Information and Modeling, and the citation records below will be updated when appropriate.

- BibTeX:

```
@article{Toraman2025,
    author = {G\"{o}zdenur Toraman and Dieter Fauconnier and Toon Verstraelen},
    title = {STable AutoCorrelation Integral Estimator (STACIE): Robust and accurate␣
 ↪transport properties from molecular dynamics simulations},
    journal = {Journal of Chemical Information and Modeling},
    volume = {65},
    number = {19},
    pages = {10445--10464},
    year = {2025},
    month = {sep},
    url = {https://doi.org/10.1021/acs.jcim.5c01475},
    doi = {10.1021/acs.jcim.5c01475},
}
```

- RIS (ProCite, Reference Manager)

```
TY  - JOUR
T1  - STable AutoCorrelation Integral Estimator: Robust and Accurate Transport␣
 ↪Properties from Molecular Dynamics Simulations
AU  - Toraman, Gözdenur
AU  - Fauconnier, Dieter
AU  - Verstraelen, Toon
Y1  - 2025/10/13
PY  - 2025
DA  - 2025/10/13
N1  - doi: 10.1021/acs.jcim.5c01475
DO  - 10.1021/acs.jcim.5c01475
T2  - Journal of Chemical Information and Modeling
JF  - Journal of Chemical Information and Modeling
JO  - J. Chem. Inf. Model.
SP  - 10445
EP  - 10464
VL  - 65
IS  - 19
M3  - doi: 10.1021/acs.jcim.5c01475
UR  - https://doi.org/10.1021/acs.jcim.5c01475
```

1.4.2 Shear Viscosity Calculations

The following paper describes in detail the calculation of shear viscosity with STACIE.

- BibTeX:

```
@article{Toraman2025ShearViscosity,
    title = {Reliable Viscosity Calculation from High-Pressure Equilibrium Molecular␣
 ↪Dynamics: Case Study of 2,2,4-Trimethylhexane.},
    author = {G\"{o}zdenur Toraman and Dieter Fauconnier and Toon Verstraelen},
    year = {2025},
    note = {in preparation}
}
```

- RIS (ProCite, Reference Manager)

```
TY  - JOUR
AU  - Toraman, Gözdenur
AU  - Fauconnier, Dieter
AU  - Verstraelen, Toon
PY  - 2025
TI  - Reliable Viscosity Calculation from High-Pressure Equilibrium Molecular Dynamics:␣
 ↪Case Study of 2,2,4-Trimethylhexane.
N1  - in preparation
ER  -
```

CHAPTER 2

Theory

This section focuses solely on the autocorrelation integral itself. The (physical) *properties* associated with this integral are discussed later.

Some derivations presented here can also be found in other sources. They are included to enhance accessibility and to provide all the necessary details for implementing STACIE.

First, the *notation* is defined, and an *overview* is presented of how STACIE works. The derivation comprises three main parts:

- A *model* for the low-frequency part of the power spectrum,
- an algorithm to *estimate the parameters* of this model, from which the autocorrelation integral and its *uncertainty* can be derived,
- and an algorithm to determine the *frequency cutoff* used to identify the low-frequency part of the spectrum.

2.1 Notation

The following notation is used throughout STACIE's documentation.

2.1.1 Special functions

- $\Gamma(z)$ is the Gamma function.
- $\gamma(z, x)$ is the lower incomplete Gamma function.

2.1.2 Statistics

- Several symbols are used to denote time:
  - t is an absolute time.
  - t_0 is a reference point on the time axis.
  - Δ_t is a time difference or lag.

  - τ denotes a relaxation or autocorrelation time, and usually has a subscript int or exp to distinguish between integrated and exponential autocorrelation times.
  - h is the time step of a discretized time axis (with equal spacing between the grid points).
  - $t_{\text{sim}} = Nh$ is the total simulation time, with N the number of steps.
  - Integer steps on a discretized time axis are denoted by indices n or m; the difference between them is Δ.
- $p_x(x)$ is the probability density function of $\hat{x}$.
- A hat is used for all stochastic quantities, including functions of stochastic quantities. This is more general than the common practice of using hats for statistical estimates only. We find it useful to clearly identify all stochastic variables. For example:
  - If $\mathcal{I}$ is the ground truth of the autocorrelation integral, then $\hat{\mathcal{I}}$ is an estimate of $\mathcal{I}$.
  - The sampling variance is denoted as $\hat{\sigma}^2_{\mathcal{I}}$.
  - The sampling covariance is denoted as $\hat{C}_{a,b}$.
  - The sampling covariance matrix of two stochastic vectors is denoted as $\hat{\mathbf{C}}_{\mathbf{a},\mathbf{b}}$.
  - A sample point from a distribution $p_a(a)$ is denoted as $\hat{a}$.
  - A realization of a continuous stochastic process $p_{a(t)}[a]$ is written as $\hat{a}(t)$.
  - Similarly, a sample from a discrete stochastic process $p_{a_n}[a]$ is written as $\hat{a}_n$.
- Expected values are denoted as:
  - E[·] is the mean operator.
  - VAR[·] is the variance operator.
  - STD[·] is the standard deviation operator.
  - COV[·, ·] is the covariance operator.
- The Gamma distribution with shape α and scale θ is denoted as:

$$p_{\text{Gamma}(\alpha,\theta)}(x) = \frac{1}{\theta^\alpha \Gamma(\alpha)} x^{\alpha-1} e^{-x/\theta}$$

- The Chi-squared distribution with ν degrees of freedom is a special case of the Gamma distribution:

$$p_{\chi^2_\nu}(x) = \frac{1}{2^{\nu/2}\Gamma(\nu/2)} x^{\nu/2-1} e^{-x/2} = p_{\text{Gamma}(\nu/2,2)}(x)$$

2.1.3 Discrete Fourier Transform

- x_n is an element of a real periodic sequence $\mathbf{x}$ with period N.
- $\mathbf{X} = \mathcal{F}[\mathbf{x}]$ is the discrete Fourier transform of the sequence, complex and periodic with period N.
- When M samples of the sequence are considered, they are denoted as $\mathbf{x}^{(m)}$ with elements $x^{(m)}_n$. Their discrete Fourier transforms are $\mathbf{X}^{(m)}$ with elements $X^{(m)}_k$.
- The grid spacing on the frequency axis is $1/hN$, where h is the spacing of the time axis.
- Frequency grid points are labeled by an index k, such that the kth frequency is k/hN.
- Hats are added if the sequences are stochastic.

2.2 STACIE Algorithm Overview

The goal of STACIE is to estimate the integral of the *ACF* of a physical, continuous, time-dependent function with an infinite domain. In practice, due to inherently finite computational resources, however, we resort to discrete and finite time-dependent sequences. We first formulate STACIE's goal in the continuous case and then reformulate it in the discrete case.

2.2.1 Continuous time, infinite domain

Consider an observation, $\hat{x}(t)$, of a time-dependent stochastic process. The integral of the ACF is defined as:

$$\mathcal{I} = \frac{F}{2} \int_{-\infty}^{\infty} c(\Delta_t)\, \mathrm{d}\Delta_t$$

with

$$c(\Delta_t) = \mathrm{COV}\left[\hat{x}(t)\,,\ \hat{x}(t+\Delta_t)\right]$$

A prefactor F is usually present, containing factors such as the temperature and/or the cell volume in Green–Kubo formalisms [Gre52, Gre54, Kub57]. The integrand is the ACF, $c(\Delta_t)$, of the time-dependent input $\hat{x}(t)$. It is common to integrate only from 0 to ∞, but we prefer to use the full range and compensate with the factor $\frac{1}{2}$ in front of the integral. (The integrand is an even function of Δ_t.) The expected value is obtained by averaging over all times t and all observations $\hat{x}(t)$.

Let $C(f)$ be the Fourier transform of the ACF, which is also known as the *PSD*:

$$C(f) = \mathcal{F}[c](f) = \int_{-\infty}^{\infty} c(\Delta_t) e^{-i2\pi f \Delta_t} \mathrm{d}\Delta_t$$

Then $\mathcal{I}$ is simply proportional to the DC component of the PSD, i.e., the zero-frequency limit of the Fourier transform of the ACF:

$$\mathcal{I} = \frac{F\, C(0)}{2}$$

At first glance, this result seems trivial, with no added value over the original form of the integral. For numerical applications, this is actually a useful identity: the sampling ACF is practically computed using the sampling PSD as an intermediate step. When $\mathcal{I}$ is derived from the PSD, the inverse transform to derive the ACF from the PSD can be skipped. As we will see later, there are other advantages to using this zero-frequency limit to compute the integral.

> **Note**
>
> Some derivations of Green–Kubo relations of transport properties, conventionally formulated as integrals of autocorrelation functions, also express them as the zero-frequency limit of an appropriate spectrum [HM13].

One can always rewrite the autocorrelation integral as a so-called Einstein–Helfand relation [Hel60], i.e., as the limit of the time derivative of the mean-square displacement [HM13]:

$$\mathcal{I} = F \frac{1}{2} \lim_{\Delta_t \to \infty} \frac{\mathrm{d}}{\mathrm{d}\Delta_t} \left\langle \left| \hat{y}(t_0 + \Delta_t) - \hat{y}(t_0) \right|^2 \right\rangle$$

where $\hat{y}$ is the antiderivative of $\hat{x}$:

$$\hat{x} = \frac{\mathrm{d}\hat{y}}{\mathrm{d}t}$$

STACIE can also be used to evaluate such limits by using samples of the time derivatives of y as input to the computation of the PSD.

2.2.2 Discretized time, periodic sequences

For simplicity, we first discuss the basic identities in terms of the ensemble average of the discrete ACF, which has no statistical uncertainties. Further below, we comment on how to deal with the uncertainties, and refer to the following sections for the details.

In terms of ensemble averages

In analogy with the continuous infinite-time case, the autocorrelation integral can be expressed in terms of discrete and periodic sequences, $\hat{x}_n$. For example, such a sequence is obtained by discretizing the time axis with a time step h and a time origin t_0:

$$\hat{x}_n = \hat{x}(t_0 + nh) \quad \forall\, n = 0 \ldots N-1$$

The underlying continuous function $\hat{x}(t)$, and thus $\hat{x}_n$, are not necessarily periodic. However, because we intend to use the discrete Fourier transform and rely on its well-known properties, we will assume in the derivations that $\hat{x}_n$ is periodic with period N. In practice, this assumption has negligible effects and is only noticeable at higher frequencies, far away from the zero-frequency limit of interest.

Due to the discretization in time, the autocorrelation integral must be approximated with a simple quadrature rule:

$$\mathcal{I} = Fh\frac{1}{2}\sum_{\Delta=0}^{N-1} \mathrm{COV}\left[\hat{x}_n\,,\, \hat{x}_{n+\Delta}\right]$$

The summand is the discrete ACF, c_Δ. The covariance is an expected value over all n and all possible realizations of the input sequence.

Let C_k be the discrete Fourier transform of the autocorrelation function:

$$C_k = \sum_{\Delta=0}^{N-1} c_\Delta \omega^{-k\Delta}$$

with $\omega = \exp(i2\pi/N)$. According to (the discrete version of) the *Wiener–Khinchin theorem* [OSB99], this Fourier transform can be written in terms of the discrete PSD:

$$C_k = \frac{1}{N}\,\mathrm{E}\left[\left|\hat{X}_k\right|^2\right]$$

with

$$\hat{X}_k = \sum_{n=0}^{N-1} (\hat{x}_n - \mathrm{E}[\hat{x}_n])\omega^{-kn}$$

In STACIE, we always work with a rescaled version of the PSD, including the factor $Fh/2$:

$$I_k = \frac{Fh}{2} C_k$$

In this notation, the autocorrelation integral is simply the zero-frequency limit of the PSD: $\mathcal{I} = I_0$.

In terms of sampling estimates

So far, we have worked with ensemble averages to define the discrete ACF and PSD. In practice, however, we must work with sampling estimates of these quantities. To keep the notation simple, we will assume that $\mathrm{E}[\hat{x}_n] = 0$. Furthermore, we will assume that we can use M independent sequences of length N:

$$\hat{x}_n^{(m)} \quad \forall\, n = 0 \ldots N-1 \quad \forall\, m = 1 \ldots M$$

In this case, the discrete sampling ACF is estimated as:

$$c_\Delta \approx \hat{c}_\Delta = \frac{1}{NM} \sum_{m=1}^{M} \sum_{n=0}^{N-1} \hat{x}_n^{(m)} \hat{x}_{n+\Delta}^{(m)}$$

The discrete sampling PSD, rescaled with STACIE's conventions, becomes:

$$I_k \approx \hat{I}_k = \frac{Fh}{2N} \sum_{m=1}^{M} \left| \hat{X}_k^{(m)} \right|^2$$

where $X_k^{(m)}$ is the discrete Fourier transform of the m-th sequence:

$$\hat{X}_k^{(m)} = \sum_{n=0}^{N-1} \hat{x}_n^{(m)} \omega^{-kn}$$

Plotting the low-frequency part of $\hat{I}_k$ will already give a quick visual estimate of $\mathcal{I}$ with the appropriate units.

A direct computation of $\hat{I}_0$ for a limited number of input sequences would at best yield a high-variance estimate of $\mathcal{I}$. (This is only possible if $\mathrm{E}[\hat{x}_n] = 0$, which is not always the case.)

To reduce the variance of the estimate of $\mathcal{I}$, STACIE derives the zero-frequency limit by fitting a model to the low-frequency part of the power spectrum $\hat{I}_k$. The following theory sections explain how this estimate can be made robustly. In summary, STACIE introduces a few *models* for the low-frequency spectrum. The parameters in such a model are estimated with likelihood maximization, and the parameter covariance is estimated with the Laplace approximation [Mac05]. To write out the likelihood function, the *statistical distribution* of the sampling PSD amplitudes must be derived. Finally, STACIE determines up to which *cutoff* frequency the model will be fitted. For cutoffs that are too high, the model becomes too simple to describe all the features in the spectrum, which leads to significant underfitting. When the cutoff is too low, too few data points are included to obtain a low-variance estimate of $\mathcal{I}$. This is solved by considering a grid of cutoff frequencies and assigning weights to each grid point based on the bias-variance trade-off of the regression. The final parameters are obtained by averaging over the grid of cutoff frequencies.

2.3 Model Spectrum

STACIE supports three models for fitting the low-frequency part of the power spectrum. In all models, the value at zero frequency corresponds to the autocorrelation integral.

1. The *ExpPolyModel* is the most general; it is an exponential function of a linear combination of simple monomials of the frequency. You can specify the degrees of the monomials, and typically a low degree works fine:
    - Degree 0 is suitable for a white noise spectrum.

- Degrees $\{0, 1\}$ can be used to extract useful information from a noisy spectrum.
- Degrees $\{0, 1, 2\}$ are applicable to spectra with low statistical uncertainty, e.g., averaged over more than 100 inputs.
- An even polynomial with degrees $\{0, 2\}$ is suitable for spectra that are expected to have a vanishing derivative at zero frequency.

The main advantage of this model is its broad applicability, as it requires little prior knowledge of the functional form of the spectrum.

2. The *PadeModel* approximates the spectrum as a rational function, i.e., the ratio of two polynomials. Such rational functions can be parameterized to have well-behaved high-frequency tails, which facilitates the regression.
3. The *LorentzModel* is a special case of the Padé model with a Lorentzian peak at the origin plus some white noise. It is equivalent to the Padé model with numerator degrees $\{0, 2\}$ and denominator degrees $\{2\}$. This special case is not only implemented for convenience, since it is the most common way of using the Padé model, but also because it allows STACIE to derive the exponential correlation time.

2.3.1 1. ExpPoly Model

The *ExpPolyModel* is defined as:

$$I^{\text{exppoly}}(f) = \exp\left(\sum_{s\in S} b_s f^s\right)$$

where S is the set of polynomial degrees, which must include 0. With this form, $\exp(b_0)$ corresponds to the integral of the autocorrelation function. When one obtains an estimate $\hat{b}_0$ and its variance $\hat{\sigma}^2_{b_0}$, the autocorrelation integral is log-normally distributed with estimated mean and variance:

$$\hat{I} = \exp\left(\hat{b}_0 + \frac{1}{2}\hat{\sigma}^2_{b_0}\right)$$
$$\hat{\sigma}^2_I = \exp\left(2\hat{b}_0 + \hat{\sigma}^2_{b_0}\right)\left(\exp(\hat{\sigma}^2_{b_0}) - 1\right)$$

To construct this model, you can create an instance of the `ExpPolyModel` class as follows:

```
from stacie import ExpPolyModel
model = ExpPolyModel([0, 1, 2])
```

This model is identified as `exppoly(0, 1, 2)` in STACIE's screen output and plots.

2.3.2 2. Pade Model

The *PadeModel* is defined as:

$$I^{\text{pade}}(f) = \frac{\sum_{s\in S_{\text{num}}} p_s f^s}{1 + \sum_{s\in S_{\text{den}}} q_s f^s}$$

where S_{num} contains the polynomial degrees in the numerator, which must include 0, and S_{den} contains the polynomial degrees in the denominator, which must exclude 0. With this model, p_0

corresponds to the integral of the autocorrelation function, for which we simply have:

$$\hat{I} = \hat{p}_0$$
$$\hat{\sigma}_I^2 = \hat{\sigma}_{p_0}^2$$

To construct this model, you can create an instance of the `PadeModel` class as follows:

```
from stacie import PadeModel
model = PadeModel([0, 2], [2])
```

This model is identified as `pade(0, 2; 2)` in STACIE's screen output and plots.

2.3.3 3. Lorentz Model

The *LorentzModel* assumes that for large time lags, Δ_t, the autocorrelation function (ACF) exhibits an exponential decay superimposed with a white noise background:

$$C(\Delta_t) = C_0\,\delta(\Delta_t) + C_1 \exp\left(-\frac{|\Delta_t|}{\tau_{\text{exp}}}\right)$$

where C_0 is the white noise level, C_1 is the amplitude of the exponential decay, and τ_{exp} is the exponential correlation time. The Lorentz model is essentially the corresponding power spectral density:

$$I^{\text{lorentz}}(f) = C_0 + \frac{2C_1\tau_{\text{exp}}}{1 + (2\pi f \tau_{\text{exp}})^2}$$

This model can be expressed as a special case of the Padé model, with numerator degrees $\{0, 2\}$ and denominator degrees $\{2\}$, and is fitted accordingly. The Padé model corresponds to a Lorentzian peak only if $q_2 > 0$ and $p_0\,q_2 > p_2$. In this case, τ_{exp} is related to the width of the peak ($2\pi\tau_{\text{exp}}$) in the power spectrum. When these conditions are met after the regression, the parameters of the Padé model are converted as follows:

$$\hat{C}_0 = \frac{\hat{p}_2}{\hat{q}_2}$$
$$\hat{C}_1 = \frac{\pi}{\sqrt{\hat{q}_2}}\left(\hat{p}_0 - \frac{\hat{p}_2}{\hat{q}_2}\right)$$
$$\hat{\tau}_{\text{exp}} = \frac{\sqrt{\hat{q}_2}}{2\pi}$$

The covariance of these parameters is derived from the covariance of the Padé parameters using first-order error propagation. We first introduce the following Jacobian matrix:

$$\mathbf{J} = \begin{pmatrix} \frac{\partial C_0}{\partial p_0} & \frac{\partial C_0}{\partial p_2} & \frac{\partial C_0}{\partial q_2} \\ \frac{\partial C_1}{\partial p_0} & \frac{\partial C_1}{\partial p_2} & \frac{\partial C_1}{\partial q_2} \\ \frac{\partial \tau_{\text{exp}}}{\partial p_0} & \frac{\partial \tau_{\text{exp}}}{\partial p_2} & \frac{\partial \tau_{\text{exp}}}{\partial q_2} \end{pmatrix} = \begin{pmatrix} 0 & \frac{1}{q_2} & -\frac{p_2}{q_2^2} \\ \frac{\pi}{\sqrt{q_2}} & -\frac{\pi}{q_2^{3/2}} & \frac{\pi}{2q_2^{3/2}}\left(\frac{3p_2}{q_2} - p_0\right) \\ 0 & 0 & \frac{1}{4\pi\sqrt{q_2}} \end{pmatrix}$$

and use it to compute the covariance of the Lorentz parameters:

$$\hat{\mathbf{C}}_{\text{lorentz}} = \mathbf{J}\,\hat{\mathbf{C}}_{\text{pade(0, 2; 2)}}\,\mathbf{J}^T$$

Note that this model is also applicable to data whose short-time correlations are not exponential, as long as the tail of the ACF decays exponentially. Such deviating short-time correlations will only affect the white noise level $\hat{C}_0$ and features in the PSD at higher frequencies, which are ignored by STACIE.

The implementation of the Lorentz model has the following advantages over the equivalent Padé model:

- The exponential correlation time and its uncertainty are computed.
- If no exponential correlation time can be computed, i.e. when $\hat{q}_2 \leq 0$ and $\hat{p}_0\,\hat{q}_2 \leq \hat{p}_2$, the fit is not retained for the final average over all cutoff grid points. (These are typically poor fits.)
- After the optimization of the model parameters at a given frequency cutoff, the following two heuristics are applied to exclude or downweight fits with a high relative error of the exponential correlation time:
  - If the relative error of the exponential correlation time is 100 times larger than that of the autocorrelation integral, the fit is discarded.
  - In all other cases, a penalty is added to the cutoff criterion, which is proportional to the ratio of the relative error of the exponential correlation time and the autocorrelation integral.

  This heuristic is based on the observation that large relative errors of the exponential correlation time often indicate poor fits for which the MAP estimate and the Laplace approximation of the posterior distribution tend to be unreliable. By taking a ratio of relative errors, the penalty is dimensionless and insensitive to the overall uncertainty of the spectrum.

  The result of these two heuristics is that the final estimate of the exponential correlation time, after averaging over all frequency cutoffs, is more robust and reliable.

  Note that the hyperparameters of the penalty can be altered, but it is recommended to keep their default values.

To construct this model, you can create an instance of the `LorentzModel` class as follows:

```
from stacie import LorentzModel
model = LorentzModel()
```

This model is identified as `lorentz()` in STACIE's screen output and plots.

2.4 Parameter Estimation

Before discussing how to fit a model to spectral data, we first review the statistics of the sampling *PSD*. Given these statistical properties, we can derive the likelihood that certain model parameters explain the observed PSD.

2.4.1 Statistics of the Sampling Power Spectral Distribution

When constructing an estimate of a discrete PSD from a finite amount of data, it is bound to contain some *uncertainty*, which will be characterized below.

The estimate of the PSD is sometimes also called the periodogram or the (empirical) power spectrum.

Consider a periodic random real sequence $\hat{\mathbf{x}}$ with elements $\hat{x}_n$ and period N. For practical purposes, it is sufficient to consider one period of this infinitely long sequence. The mean of the sequence is zero, and its covariance is $\mathrm{COV}[\hat{x}_n, \hat{x}_m]$. The distribution of the sequence is stationary, i.e., each time translation of a sequence results in an equally probable sample. As a result, the

covariance has a circulant structure:

$$\mathrm{COV}[\hat{x}_n\,,\,\hat{x}_m] = c_\Delta = c_{-\Delta}$$

with $\Delta = n - m$. Thus, we can express the covariance with a single index and treat it as a real periodic sequence, albeit not stochastic. c_Δ is also known as the autocovariance or autocorrelation function of the stochastic process because it expresses the covariance of a sequence $\hat{\mathbf{x}}$ with itself translated by Δ steps.

The discrete Fourier transform of the sequence is:

$$\hat{X}_k = \sum_{n=0}^{N-1} \hat{x}_n \omega^{-kn}$$

with $\omega = e^{2\pi i/N}$.

A well-known property of circulant matrices is that their eigenvectors are sine- and cosine-like basis functions. As a result, the covariance of the discrete Fourier transform $\hat{\mathbf{X}}$ becomes diagonal. To make this derivation self-contained, we write out the mean and covariance of $\hat{X}_k$ explicitly. Note that the operators $\mathrm{E}[\cdot]$, $\mathrm{VAR}[\cdot]$, and $\mathrm{COV}[\cdot,\cdot]$ are expected values over all possible realizations of the sequence.

For the expected value of the Fourier transform, we take advantage of the fact that all time translations of $\hat{\mathbf{x}}$ belong to the same distribution. We can explicitly compute the average over all time translations, in addition to computing the mean, without loss of generality. In the last steps, the index n is relabeled to $n-m$, and some factors are rearranged, after which the sums can be worked out.

$$\begin{aligned}
E[\hat{X}_k] &= \mathrm{E}\left[\sum_{n=0}^{N-1} \hat{x}_n \omega^{-kn}\right] \\
&= \mathrm{E}\left[\frac{1}{N}\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} \hat{x}_{n+m} \omega^{-kn}\right] \\
&= \mathrm{E}\left[\frac{1}{N}\underbrace{\left(\sum_{m=0}^{N-1} \omega^{km}\right)}_{=0}\sum_{n=0}^{N-1} \hat{x}_n \omega^{-kn}\right] \\
&= 0
\end{aligned}$$

The derivation of the covariance uses similar techniques. In the following derivation, $*$ stands for complex conjugation. Halfway through, the summation index n is written as $n = \Delta + m$.

$$\begin{aligned}
\mathrm{COV}[\hat{X}_k^*\,,\,\hat{X}_\ell] &= \mathrm{COV}\left[\sum_{m=0}^{N-1} \hat{x}_m \omega^{km}\,,\,\sum_{n=0}^{N-1} \hat{x}_n \omega^{-\ell n}\right] \\
&= \sum_{m=0}^{N-1}\sum_{n=0}^{N-1} \omega^{km-\ell n} c_{n-m} \\
&= \sum_{m=0}^{N-1} \omega^{km-\ell m} \sum_{\Delta=0}^{N-1} \omega^{-\ell\Delta} c_\Delta \\
&= N\delta_{k,\ell}\,\mathcal{F}[c]_\ell
\end{aligned}$$

To finalize the result, we need to work out the discrete Fourier transform of the autocorrelation function, c_Δ. Again, we make use of the freedom to insert a time average when computing a mean. Note that this derivation assumes $\mathrm{E}[\hat{x}_n] = 0$ to keep the notation bearable.

$$\begin{aligned}
C_k = \mathcal{F}[\mathbf{c}]_k &= \sum_{\Delta=0}^{N-1} \omega^{-k\Delta} \,\mathrm{E}\left[\frac{1}{N}\sum_{n=0}^{N-1} \hat{x}_n\, \hat{x}_{n+\Delta}\right] \\
&= \frac{1}{N}\,\mathrm{E}\left[\sum_{n=0}^{N-1} \omega^{kn}\hat{x}_n \sum_{\Delta=0}^{N-1} \omega^{-k\Delta-kn}\hat{x}_{n+\Delta}\right] \\
&= \frac{1}{N}\,\mathrm{E}\left[|\hat{X}_k|^2\right]
\end{aligned}$$

This is the discrete version of the Wiener–Khinchin theorem [OSB99].

By combining the previous two results, we can write the covariance of the Fourier transform of the input sequence as:

$$\mathrm{COV}[\hat{X}_k^*\,,\, \hat{X}_\ell] = \delta_{k,\ell}\,\mathrm{E}\left[|\hat{X}_k|^2\right] = N\delta_{k,\ell}C_k$$

For the real component of $\hat{X}_k$ ($= \hat{X}_{-k}^*$), we find:

$$\begin{aligned}
\mathrm{VAR}[\Re(\hat{X}_k)] &= \frac{1}{4}\,\mathrm{VAR}[\hat{X}_k + \hat{X}_k^*] \\
&= \frac{1}{4}\Big(\mathrm{COV}[\hat{X}_k\,,\,\hat{X}_k] + \mathrm{COV}[\hat{X}_k\,,\,\hat{X}_k^*] + \mathrm{COV}[\hat{X}_k^*\,,\,\hat{X}_k] + \mathrm{COV}[\hat{X}_k^*\,,\,\hat{X}_k^*]\Big) \\
&= \frac{1}{4}\Big(\mathrm{COV}[\hat{X}_{-k}^*\,,\,\hat{X}_k] + \mathrm{COV}[\hat{X}_k\,,\,\hat{X}_k^*] + \mathrm{COV}[\hat{X}_k^*\,,\,\hat{X}_k] + \mathrm{COV}[\hat{X}_k^*\,,\,\hat{X}_{-k}]\Big) \\
&= \begin{cases} NC_0 & \text{if } k = 0 \\ \frac{N}{2}C_k & \text{if } 0 < k < N \end{cases}
\end{aligned}$$

Similarly, for the imaginary component (which is zero for $k = 0$):

$$\begin{aligned}
\mathrm{VAR}[\Im(\hat{X}_k)] &= \frac{1}{4}\,\mathrm{VAR}[\hat{X}_k - \hat{X}_k^*] \\
&= \frac{1}{4}\Big(\mathrm{COV}[\hat{X}_k\,,\,\hat{X}_k] - \mathrm{COV}[\hat{X}_k\,,\,\hat{X}_k^*] - \mathrm{COV}[\hat{X}_k^*\,,\,\hat{X}_k] + \mathrm{COV}[\hat{X}_k^*\,,\,\hat{X}_k^*]\Big) \\
&= \begin{cases} 0 & \text{if } k = 0 \\ \frac{N}{2}C_k & \text{if } 0 < k < N \end{cases}
\end{aligned}$$

The real and imaginary components have no covariance:

$$\begin{aligned}
\mathrm{COV}[\Re(\hat{X}_k)\,,\Im(\hat{X}_k)] &= \frac{1}{4}\,\mathrm{COV}[\hat{X}_k + \hat{X}_k^*\,,\,\hat{X}_k - \hat{X}_k^*] \\
&= \frac{1}{4}\Big(\mathrm{COV}[\hat{X}_k\,,\,\hat{X}_k] - \mathrm{COV}[\hat{X}_k\,,\,\hat{X}_k^*] \\
&\qquad + \mathrm{COV}[\hat{X}_k^*\,,\,\hat{X}_k] - \mathrm{COV}[\hat{X}_k^*\,,\,\hat{X}_k^*]\Big) \\
&= 0
\end{aligned}$$

In summary, the Fourier transform of a stationary stochastic process consists of uncorrelated real and imaginary components at each frequency. Furthermore, the variance of the Fourier transform

is proportional to the power spectrum. This simple statistical structure makes the spectrum a convenient starting point for further analysis and uncertainty quantification. In comparison, the ACF has non-trivial correlated uncertainties [Bar80, Bos96, FZ09], making it difficult to fit models directly to the ACF (or its running integral).

If we further assume that the sequence $\hat{\mathbf{x}}$ is the result of a periodic Gaussian process, the Fourier transform is normally distributed. In this case, the empirical power spectrum follows a scaled Chi-squared distribution [EMB17, Ful95, Pri82, SS17]. For notational consistency, we will use the Gamma(α, θ) distribution with shape parameter α and scale parameter θ:

$$\begin{aligned}
\hat{C}_0 &= \frac{1}{N}|\hat{X}_0|^2 \sim \text{Gamma}(\tfrac{1}{2}, 2C_0) \\
\hat{C}_{N/2} &= \frac{1}{N}|\hat{X}_{N/2}|^2 \sim \text{Gamma}(\tfrac{1}{2}, 2C_{N/2}) \quad \text{if } N \text{ is even} \\
\hat{C}_k &= \frac{1}{N}|\hat{X}_k|^2 \sim \text{Gamma}(1, C_k) \quad \text{for } 0 < k < N \text{ and } k \neq N/2
\end{aligned}$$

Note that $\hat{X}_0$ and $\hat{X}_{N/2}$ have only a real component because the input sequence $\hat{\mathbf{x}}$ is real, which corresponds to a Chi-squared distribution with one degree of freedom. For all other frequencies, $\hat{X}_k$ has a real and imaginary component, resulting in two degrees of freedom.

Spectra are often computed by averaging them over M sequences to reduce the variance. In this case, the M-averaged empirical spectrum is distributed as:

$$\hat{C}_k = \frac{1}{NM}\sum_{s=1}^{M}|\hat{X}_0^s|^2 \sim \text{Gamma}(\tfrac{\nu_k}{2}, \tfrac{2}{\nu_k}C_k)$$

with

$$\nu_k = \begin{cases} M & \text{if } k = 0 \\ M & \text{if } k = N/2 \text{ and } N \text{ is even} \\ 2M & \text{otherwise} \end{cases}$$

The rescaled spectrum used in STACIE, $\hat{I}_k$, has the same distribution, except for the scale parameter:

$$\hat{I}_k = \frac{Fh}{2}\hat{C}_k \sim \text{Gamma}(\tfrac{\nu_k}{2}, \tfrac{2}{\nu_k}I_k)$$

2.4.2 Regression

To identify the low-frequency part of the spectrum, we introduce a smooth switching function that goes from 1 to 0 as the frequency increases:

$$w(f_k|f_{\text{cut}}) = \frac{1}{1 + (f_k/f_{\text{cut}})^\beta}$$

This switching function is 1/2 when $f_k = f_{\text{cut}}$. The hyperparameter β controls the steepness of the transition and is 8 by default. (This should be fine for most applications.) This value can be set with the `switch_exponent` argument of the *estimate_acint()* function. One can better appreciate the advantage of this switching function by rewriting with a hyperbolic tangent:

$$w(f_k|f_{\text{cut}}) = \frac{1}{2}\left[1 - \tanh\left(\frac{\beta}{2}\ln\frac{f_k}{f_{\text{cut}}}\right)\right]$$

This shows that the switching is scale invariant, i.e., it does not depend on the unit of the frequency, because the frequency appears only in a logarithm. The parameter β controls the width of the transition region on a logarithmic scale.

We derive below how to fit parameters for a given frequency cut-off f_{cut}. The *next section* describes how to find suitable cutoffs.

To fit the model, we use a form of local regression by introducing weights into the log-likelihood function. The weighted log likelihood of the model $I_k^{\text{model}}(\mathbf{b})$ with parameter vector $\mathbf{b}$ becomes:

$$\begin{aligned}\ln \mathcal{L}(\mathbf{b}) &= \sum_{k\in K} w(f_k|f_{\text{cut}}) \ln p_{\text{Gamma}(\alpha_k,\theta_k)}(\hat{C}_k) \\ &= \sum_{k\in K} w(f_k|f_{\text{cut}}) \left[-\ln\Gamma(\alpha_k) - \ln\big(\theta_k(\mathbf{b})\big) + (\alpha_k - 1)\ln\left(\frac{\hat{C}_k}{\theta_k(\mathbf{b})}\right) - \frac{\hat{C}_k}{\theta_k(\mathbf{b})}\right]\end{aligned}$$

with

$$\begin{aligned}\alpha_k &= \frac{\nu_k}{2} \\ \theta_k(\mathbf{b}) &= \frac{2I_k^{\text{model}}(\mathbf{b})}{\nu_k}\end{aligned}$$

This log-likelihood is maximized to estimate the model parameters. The zero-frequency limit of the fitted model is then the estimate of the autocorrelation integral.

Note

It is worth mentioning that the cutoff frequency is not a proper hyperparameter in the Bayesian sense. It appears in the weight factor $w(f_k|f_{\text{cut}})$, which is not part of the model. Instead, it is a concept taken from local regression methods. One conceptual limitation of this approach is that the unit of the likelihood function, $\mathcal{L}(\mathbf{b})$, depends on the cutoff frequency. As a result, one cannot compare the likelihood of two different cutoffs. This is of little concern when fitting parameters for a fixed cutoff, but it is important to keep in mind when searching for suitable cutoffs.

For compatibility with the SciPy optimizers, the cost function $\text{cost}(\mathbf{b}) = -\ln \mathcal{L}(\mathbf{b})$ is minimized. STACIE implements first and second derivatives of $\text{cost}(\mathbf{b})$, and also a good initial guess of the parameters, using efficient vectorized NumPy code. These features make the optimization of the parameters both efficient and reliable. The optimized parameters are denoted as $\hat{\mathbf{b}}$. The hats indicate that these are statistical estimates because they are derived from data statistical uncertainties.

The Hessian computed with the estimated parameters, $\text{cost}(\hat{\mathbf{b}})$, must be positive definite. (If non-positive eigenvalues are found, the optimization is treated as failed.)

$$\hat{\mathbf{H}} > 0 \quad \text{with} \quad \hat{H}_{ij} = \left.\frac{\partial^2 \text{cost}}{\partial b_i \partial b_j}\right|_{\mathbf{b}=\hat{\mathbf{b}}}$$

The estimated covariance matrix of the estimated parameters is approximated by the inverse of the Hessian, which can be justified with the Laplace approximation: [Mac05].

$$\hat{C}_{\hat{b}_i,\hat{b}_j} = \left(\hat{\mathbf{H}}^{-1}\right)_{ij}$$

This covariance matrix characterizes the uncertainties of the model parameters and thus also of the autocorrelation integral. More accurate covariance estimates can be obtained with Monte

Carlo sampling, but this is not implemented in STACIE. Note that this covariance only accounts for the uncertainty due to noise in the spectrum, which is acceptable if the cutoff frequency is a fixed value. However, in STACIE, the cutoff frequency is also fitted, meaning that the uncertainty due to the cutoff must also be accounted for. This will be discussed in the *next section*.

Note

The estimated covariance has no factor $N_{\text{fit}}/(N_{\text{fit}} - N_{\text{par}})$, where N_{fit} is the amount of data in the fit and N_{par} is the number of parameters. This factor is specific to the case of (non)linear regression with normal deviates of which the standard deviation is not known *a priori* [Mil11]. Here, the amplitudes are Gamma-distributed with a known shape parameter. Only the scale parameter at each frequency is predicted by the model.

2.4.3 Regression Cost Z-score

When a model is too simple to explain the data, the regression cost Z-score can be used to quantify the goodness of fit. This is implemented in STACIE to facilitate the detection of poorly fitted models, which can sometimes occur if the selected model cannot explain the data for any cutoff frequency. This Z-score is defined as:

$$Z_{\text{cost}}(\mathbf{b}) = \frac{\text{cost}(\mathbf{b}) - \text{E}\left[\hat{\text{cost}}(\mathbf{b})\right]}{\text{STD}\left[\hat{\text{cost}}(\mathbf{b})\right]}$$

The mean $\text{E}\left[\hat{\text{cost}}(\mathbf{b})\right]$ and standard deviation $\text{STD}\left[\hat{\text{cost}}(\mathbf{b})\right]$ are computed as expectation values over all possible spectra sampled from the Gamma distribution corresponding to the model parameters $\mathbf{b}$. For both expectation values STACIE implements computationally efficient closed-form solutions.

The Z-score is easily interpretable as a goodness of fit measure. When the model fits the data well, the Z-score has a zero mean and unit standard deviation. When the model is too simple and underfits the data, the Z-score is positive and quickly exceeds the standard deviation. For example a Z-score of 2 indicates that the model cost is two standard deviations above its mean, suggesting that the model is too simple.

The mean is derived as follows:

$$\text{E}\left[\hat{\text{cost}}(\mathbf{b})\right] = \sum_{k \in K} w(f_k | f_{\text{cut}}) \, \text{E}\left[-\ln p_{\text{Gamma}(\alpha_k, \theta_k)}(\hat{C}_k)\right]$$

where the mean is computed by sampling $\hat{C}_k$ from the distribution $\text{Gamma}(\alpha_k, \theta_k)$. This mean is also known as the entropy of the distribution, with a well-known closed-form solution. Inserting this solution into the previous equation gives:

$$\text{E}\left[\hat{\text{cost}}(\mathbf{b})\right] = \sum_{k \in K} w(f_k | f_{\text{cut}}) \Big(\alpha_k - \ln\big(\theta_k(\mathbf{b})\big) + \ln\Gamma(\alpha_k) + (1 - \alpha_k)\psi(\alpha_k) \Big)$$

where $\psi(\alpha)$ is the digamma function.

The standard deviation is best derived by first computing the variance of the cost function:

$$\text{VAR}\left[\hat{\text{cost}}(\mathbf{b})\right] = \sum_{k \in K} w(f_k | f_{\text{cut}})^2 \, \text{VAR}\left[-\ln p_{\text{Gamma}(\alpha_k, \theta_k)}(\hat{C}_k)\right]$$

The variance of the cost function is also defined as an expectation value over $\hat{C}_k$ from the distribution $\text{Gamma}(\alpha_k, \theta_k)$. This variance can also be derived analytically, but the result is not as

well-known, so we will work it out here. The logarithm of the probability density has only two terms that depend on the random variable $\hat{C}_k$, which are relevant for the variance:

$$\begin{aligned}
&\mathrm{VAR}\left[-\ln p_{\mathrm{Gamma}(\alpha_k,\theta_k)}(\hat{C}_k)\right] \\
&\quad= \mathrm{VAR}\left[(\alpha_k - 1)\ln(\hat{C}_k) - \frac{\hat{C}_k}{\theta_k(\mathbf{b})}\right] \\
&\quad= (\alpha_k - 1)^2\,\mathrm{VAR}\left[\ln(\hat{C}_k)\right] + \frac{1}{\theta_k^2(\mathbf{b})}\,\mathrm{VAR}\left[\hat{C}_k\right] - 2\frac{\alpha_k - 1}{\theta_k(\mathbf{b})}\,\mathrm{COV}\left[\ln(\hat{C}_k), \hat{C}_k\right]
\end{aligned}$$

The first two terms are well-known results, i.e. the variance of the log-Gamma and Gamma distributions, respectively.

$$\begin{aligned}
\mathrm{VAR}\left[\ln(\hat{C}_k)\right] &= \psi_1(\alpha_k) \\
\mathrm{VAR}\left[\hat{C}_k\right] &= \alpha_k\,\theta_k^2(\mathbf{b})
\end{aligned}$$

where $\psi_1(\alpha)$ is the trigamma function. The only term that requires some more work is the third term:

$$\mathrm{COV}\left[\ln(\hat{C}_k), \hat{C}_k\right] = \mathrm{E}\left[\ln(\hat{C}_k)\,\hat{C}_k\right] - \mathrm{E}\left[\ln(\hat{C}_k)\right]\,\mathrm{E}\left[\hat{C}_k\right]$$

A derivation of the first term can be found in the wonderful online book of statistical proofs. The second term contains well-known expectation values of the Gamma distribution. The results are:

$$\begin{aligned}
\mathrm{E}\left[\ln(\hat{C}_k)\,\hat{C}_k\right] &= \alpha_k\,\theta_k(\mathbf{b})\Big(\psi(\alpha_k + 1) + \ln\big(\theta_k(\mathbf{b})\big)\Big) \\
\mathrm{E}\left[\ln(\hat{C}_k)\right] &= \psi(\alpha_k) + \ln\big(\theta_k(\mathbf{b})\big) \\
\mathrm{E}\left[\hat{C}_k\right] &= \alpha_k\,\theta_k(\mathbf{b})
\end{aligned}$$

The covariance can now be worked out by making using of the well-known recurrence relation of the digamma function:

$$\begin{aligned}
\mathrm{COV}\left[\ln(\hat{C}_k), \hat{C}_k\right] &= \alpha_k\,\theta_k(\mathbf{b})\big(\psi(\alpha_k + 1) - \psi(\alpha_k)\big) \\
&= \theta_k(\mathbf{b})
\end{aligned}$$

Putting it all together, we find the variance of the logarithm of the probability density of the Gamma distribution:

$$\mathrm{VAR}\left[-\ln p_{\mathrm{Gamma}(\alpha_k,\theta_k)}(\hat{C}_k)\right] = (\alpha_k - 1)^2\psi_1(\alpha_k) - \alpha_k + 2$$

The standard devation in the Z-score finally becomes:

$$\mathrm{STD}\left[\hat{\mathrm{cost}}(\mathbf{b})\right] = \sqrt{\sum_{k\in K} w(f_k|f_{\mathrm{cut}})^2\Big((\alpha_k - 1)^2\psi_1(\alpha_k) - \alpha_k + 2\Big)}$$

It is noteworthy that the standard deviation is independent of the model parameters $\mathbf{b}$.

2.5 Frequency Cutoff

In STACIE, a model is fitted to the low-frequency part of the sampling *PSD*. This low-frequency part is defined by a cutoff frequency, f_{cut}, above which the model is not expected to explain the data. The *previous section* discussed how to implement a local regression using a smooth switching function parameterized by such a cutoff frequency, f_{cut}. A good choice for the cutoff seeks a trade-off between two conflicting goals:

1. If too much data is included in the fit, the model may be too simple to explain all features of the spectrum. It underfits the data, and the estimates are generally biased.
2. If too little data is included in the fit, the variance of the estimated parameters is larger than necessary, meaning that not all relevant information is used.

Finding a good compromise between these two can be done in several ways, and similar difficulties arise in other approaches to compute transport coefficients. For example, in the direct quadrature of the *ACF*, the truncation of the integral faces a similar trade-off.

Because the *model* is fitted to a sampling PSD with known and convenient statistical properties, as discussed in the *previous section*, it is possible to determine the cutoff frequency systematically. As also explained in the *previous section*, the cutoff frequency is not a proper hyperparameter in the Bayesian sense, meaning that a straightforward marginalization over the cutoff frequency is not possible [RW05]. Instead, STACIE uses cross-validation to find a good compromise between bias and variance. As explained below, a model likelihood is constructed based on cross-validation, whose unit is independent of the cutoff frequency. This model is then used to marginalize estimated parameters over the cutoff frequency.

2.5.1 Effective number of fitting points

The concept of "effective number of fitting points" is used regularly in the following subsections. For a given cutoff frequency, it is defined as:

$$N_{\text{eff}}(f_{\text{cut}}) = \sum_{k=1}^{M} w(f_k | f_{\text{cut}})$$

This is simply the sum of the weights introduced in the section on *regression*.

2.5.2 Grid of Cutoff Frequencies

STACIE uses a logarithmic grid of cutoff frequencies and fits model parameters for each cutoff. The grid is defined as:

$$f_{\text{cut},j} = f_{\text{cut},0}\, r^j$$

where $f_{\text{cut},0}$ is the lowest cutoff frequency in the grid, and r is the ratio between two consecutive cutoff frequencies. The following parameters define the grid:

- The lowest cutoff is determined by solving:

  $$N_{\text{eff}}(f_{\text{cut,min}}) = N_{\text{eff, min}}$$

  where $N_{\text{eff, min}}$ is a user-defined parameter, and P is the number of model parameters. In STACIE, the default value is $N_{\text{eff, min}} = 5P$, which reduces the risk of numerical issues in the regression. The value of $N_{\text{eff, min}}$ can be adjusted using the `neff_min` option in the function *`estimate_acint()`*.

- The maximum cutoff frequency is determined by solving:

  $$N_{\text{eff}}(f_{\text{cut,max}}) = N_{\text{eff, max}}$$

  where $N_{\text{eff, max}}$ is a user-defined parameter. In STACIE, the default value is $N_{\text{eff, max}} = 1000$. This value can be modified using the `neff_max` option in the function *`estimate_acint()`*. The purpose of this parameter is to limit the computational cost of the regression. (For short inputs, the highest cutoff frequency is also constrained by the Nyquist frequency.)

- The ratio between two consecutive cutoff frequencies is:

$$r = \exp(g_{\text{sp}}/\beta)$$

  where g_{sp} is a user-defined parameter, and β controls the steepness of the switching function $w(f\,|\,f_{\text{cut}})$. In STACIE, the default value is $g_{\text{sp}} = 0.5$. This value can be adjusted using the `fcut_spacing` option in the function *`estimate_acint()`*. By incorporating the parameter β into the definition of r, a steeper switching function automatically requires a finer grid of cutoff frequencies.

Parameters are fitted for all cutoffs, starting from the lowest one. As shown below, the scan of the cutoff frequencies can be stopped before reaching $f_{\text{cut,max}}$.

2.5.3 Cross-Validation

Given a cutoff frequency, $f_{\text{cut},j}$, STACIE estimates model parameters $\hat{\mathbf{b}}^{(j)}$ and their covariance matrix $\hat{\mathbf{C}}_{\mathbf{b}^{(j)},\mathbf{b}^{(j)}}$. To quantify the degree of over- or underfitting, the model parameters are further refined by fitting them to the first and second halves of the low-frequency part of the sampling PSD. To make these refinements robust, the two halves are defined using smooth switching functions:

$$w_{\text{left}}(f\,|\,f_{\text{cut},j}) = w(f\,|\,g_{\text{cv}} f_{\text{cut},j}/2)$$
$$w_{\text{right}}(f\,|\,f_{\text{cut},j}) = w(f\,|\,g_{\text{cv}} f_{\text{cut},j}) - w_{\text{left}}(f\,|\,f_{\text{cut},j})$$

The parameter g_{cv} is a user-defined parameter that controls the amount of data used in the refinements. In STACIE, the default value is $g_{\text{cv}} = 1.25$, meaning that 25% more data is used compared to the original fit. (This makes the cross-validation more sensitive to underfitting, which has been found beneficial in practice.) This parameter can be controlled using the `fcut_factor` option in the *`CV2LCriterion`* class. An instance of this class can be passed to the `cutoff_criterion` argument of the function *`estimate_acint()`*.

Instead of performing two full non-linear regressions of the parameters for the two halves, linear regression is used to make first-order approximations of the changes in parameters. For cutoffs leading to well-behaved fits, these corrections are small, justifying the use of a linear approximation.

The design matrix of the linear regression is:

$$D_{kp} = \left.\frac{\partial I^{\text{model}}(f_k; \mathbf{b})}{\partial b_p}\right|_{\mathbf{b}=\hat{\mathbf{b}}^{(j)}}$$

The expected values are the residuals between the sampling PSD and the model:

$$y_k = \hat{I}_k - I^{\text{model}}(f_k; \hat{\mathbf{b}}^{(j)})$$

The measurement error is the standard deviation of the Gamma distribution, using the model spectrum in the scale parameter and the shape parameter of the sampling PSD:

$$\sigma_k = \frac{I^{\text{model}}(f_k; \hat{\mathbf{b}}^{(j)})}{\sqrt{\alpha_k}}$$

The weighted regression to obtain first-order corrections to the parameters $\hat{\mathbf{b}}^{(j)}$ solves the following linear system in the least-squares sense:

$$\frac{w_k}{\sigma_k} \sum_{p=1}^{P} D_{kp}\, \hat{b}^{(j)}_{\text{corr},p} = \frac{w_k}{\sigma_k} y_k$$

where w_k is the weight of the k-th frequency point. This system is solved once with weights for the left half and once for the right half.

The function `linear_weighted_regression()` provides a robust pre-conditioned implementation of the above linear regression. It can handle multiple weight vectors simultaneously and can directly compute linear combinations of parameters for different weight vectors. It is used to directly compute the difference between the corrections for the left and right halves, denoted as $\hat{\mathbf{d}}$, and its covariance matrix $\hat{\mathbf{C}}_{\mathbf{d},\mathbf{d}}$. Normally, the model parameters fitted to both halves must be the same, and the negative log-likelihood of the fitted parameters being identical is given by:

$$\text{criterion}^{\text{CV2L}} = -\ln \mathcal{L}^{\text{CV2L}}\left(\hat{\mathbf{d}}^{(j)}, \hat{\mathbf{C}}_{\mathbf{d}}^{(j)}\right) = \frac{P}{2}\ln(2\pi) + \underbrace{\frac{1}{2}\ln\left|\hat{\mathbf{C}}_{\mathbf{d}}^{(j)}\right|}_{\text{variance}} + \underbrace{\frac{1}{2}\left(\hat{\mathbf{d}}^{(j)}\right)^{\top}\left(\hat{\mathbf{C}}_{\mathbf{d}}^{(j)}\right)^{-1}\hat{\mathbf{d}}^{(j)}}_{\text{bias}}$$

When starting from the lowest cutoff grid point, the second term of the criterion (the variance term) will be high because the parameters are poorly constrained by the small amount of data used in the fit. As the cutoff frequency and the effective number of fitting points increase, the model becomes better constrained. The second term will decrease, but as soon as the model underfits the data, the third term (the bias term) will steeply increase. Practically, the cutoff scan is interrupted when the criterion exceeds the incumbent by g_{incr}. The default value is $g_{\text{incr}} = 100$, but this can be changed using the `criterion_high` option in the function `estimate_acint()`.

A good cutoff frequency is the one that minimizes the criterion, thereby finding a good compromise between bias and variance.

> **Note**
>
> In the description above, we assume that a cutoff exists for which the model can explain the spectrum. With unfortunate model choices, this may not be the case. The cutoff scan will then try to find a compromise between the bias and variance, but this will not be useful if the model can not be describe the spectrum at all. This situation can be detected by checking the Regression Cost Z-score, derived in the previous section.

2.5.4 Marginalization Over the Cutoff Frequency

Any method to deduce the cutoff frequency from the spectrum, whether it is human judgment or an automated algorithm, introduces some *uncertainty* in the final result because the cutoff is based on a sampling PSD spectrum with statistical uncertainty.

In STACIE, this uncertainty is accounted for by marginalizing the model parameters over the cutoff frequency, using $\mathcal{L}^{\text{CV2L}}$ as a model for the likelihood. This approach naturally incorporates the uncertainty in the cutoff frequency and is preferred over fixing the cutoff frequency at a single value.

Practically, the final estimate of the parameters and their covariance is computed using standard expressions for mixture distributions:

$$\hat{\mathbf{b}} = \sum_{j=1}^{J} W_j\, \hat{\mathbf{b}}^{(j)}$$

$$\hat{C}_{\mathbf{b},\mathbf{b}} = \sum_{j=1}^{J} W_j\, \left(\hat{C}_{\mathbf{b}^{(j)},\mathbf{b}^{(j)}} + (\hat{\mathbf{b}}^{(j)} - \hat{\mathbf{b}})(\hat{\mathbf{b}}^{(j)} - \hat{\mathbf{b}})^{\top}\right)$$

Here, $\hat{\mathbf{b}}^{(j)}$ and $\hat{C}_{\mathbf{b}^{(j)},\mathbf{b}^{(j)}}$ represent the parameters and their covariance, respectively, for cutoff j. The weights W_j sum to 1 and are proportional to $\mathcal{L}^{\mathrm{CV2L}}$.

Note that STACIE also computes weighted averages of other quantities in the same way, including:

- The effective number of fitting points, N_{eff}
- The cutoff frequency, f_{cut}
- The switching function, $w(f\,|\,f_{\mathrm{cut}})$
- The regression cost Z-score, Z_{cost}
- The cutoff criterion Z-score, $Z_{\mathrm{criterion}}$ (defined below)

2.5.5 Cutoff Criterion Z-score

In the far majority of cases, the cutoff criterion will be dominated by the bias term: it typically increases steeply as soon as the model underfits the data. In contrast, the variance term decreases relatively slowly. As a result, the minimum of the cutoff criterion is well-defined at the onset of underfitting. This works particularly well when the spectrum can be computed with a high frequency resolution. Unfortunately, there are cases where this ideal pattern does not hold, usually when providing too little data to STACIE, in which the cutoff criterion exhibits statistical fluctuations. To detect such ill-constrained cases, STACIE computes a Z-score for the cutoff criterion. This Z-score is defined as in the same spirit as the Regression Cost Z-score, but now using $\mathrm{criterion}^{\mathrm{CV2L}}$ instead of the regression cost function.

The cutoff criterion Z-score is defined as:

$$Z_{\mathrm{criterion}} = \frac{\mathrm{criterion}^{\mathrm{CV2L}} - \mathrm{E}\left[\hat{\mathrm{criterion}}^{\mathrm{CV2L}}\right]}{\mathrm{STD}\left[\hat{\mathrm{criterion}}^{\mathrm{CV2L}}\right]}$$

The mean and standard deviation are computed by averaging over all vectors $\mathbf{d}$ from the likelihood $\mathcal{L}^{\mathrm{CV2L}}$.

For cutoff frequencies that minimize the criterion, the Z-score should be close to zero, because the difference between two parameters fitted to the left and right halves of the spectrum should be zero within the statistical uncertainty. When the bias term in the cutoff criterion is noisy, it may feature minima dominated by the variance term, which are not useful and may produce unreliable estimates (with misleading error bars). In such cases, the cutoff criterion Z-score will be significantly larger than zero.

The mean in the Z-score can be worked out easily because it corresponds to the entropy of a multivariate normal distribution:

$$\mathrm{E}\left[\hat{\mathrm{criterion}}^{\mathrm{CV2L}}\right] = \mathrm{E}_{\mathbf{d}}\left[-\ln \mathcal{L}^{\mathrm{CV2L}}\left(\hat{\mathbf{d}}, \hat{\mathbf{C}}_{\mathbf{d},\mathbf{d}}\right)\right] = \frac{P}{2}\ln(2\pi e) + \frac{1}{2}\ln|\hat{\mathbf{C}}_{\mathbf{d},\mathbf{d}}|$$

For the standard deviation, we first work out the variance of the cutoff criterion:

$$\mathrm{VAR}\left[\hat{\mathrm{criterion}}^{\mathrm{CV2L}}\right] = \mathrm{VAR}_{\mathbf{d}}\left[-\ln \mathcal{L}^{\mathrm{CV2L}}\left(\hat{\mathbf{d}}, \hat{\mathbf{C}}_{\mathbf{d},\mathbf{d}}\right)\right]$$

Only the bias term contributes to the variance of the cutoff criterion. This term can be rewritten as one half the sum of P squared standard normal distributed variables. By making use of the properties of the chi-squared distribution, we can work out the variance of the bias term and take the square root to obtain the standard deviation:

$$\mathrm{STD}\left[\hat{\mathrm{criterion}}^{\mathrm{CV2L}}\right] = \sqrt{\frac{P}{2}}$$

CHAPTER 3

Preparing Inputs

This section explains how to prepare input sequences for STACIE to ensure high-quality results. It consists of three parts:

- Guidelines for planning and preparing *sufficient input sequences* for STACIE.
- Instructions for efficiently storing sequences on disk using *block averages*.
- Recommendations for performing *molecular dynamics* simulations that generate suitable inputs for STACIE.

3.1 How to Prepare Sufficient Inputs for STACIE?

This section explains how to achieve a desired relative error ϵ_{rel} of the autocorrelation integral estimate, $\hat{I}$. The preparation of sufficient inputs consists of two steps:

1. First, we guesstimate the number of independent sequences, M, required to achieve the desired relative error.
2. Second, a test is proposed to verify that the number of steps in the input sequences, N, is sufficient to achieve the desired relative error. Because this second step requires information that is not available *a priori*, it involves an analysis with STACIE of a preliminary set of input sequences. This will reveal whether the number of steps in the input sequences is sufficient. If not, the inputs must be extended, e.g., by running additional simulations or measurements.

3.1.1 Step 1: Guesstimate the Number of Independent Sequences

Because the amplitudes of the (rescaled) sampling PSD are Gamma-distributed, one can show that the relative error of the PSD (mean divided by the standard deviation) is given by:

$$\frac{\text{STD}[\hat{I}_k]}{\text{E}[\hat{I}_k]} = \sqrt{\frac{2}{\nu_k}}$$

where ν_k is the number of degrees of freedom of the sampling PSD at frequency k. For most frequencies, we have $\nu_k = 2M$. (See *Parameter Estimation* for details.) Because we are only

interested in an coarse estimate of the required number of independent sequences, we will use $\nu_k = 2M$ for all frequencies.

Let us assume for simplicity that we want to fit a white noise spectrum, which can be modeled with a single parameter, namely the amplitude of the spectrum. In this case, this single parameter is also the autocorrelation integral. By taking the average of the PSD over the first N_{eff} frequencies, the relative error of the autocorrelation integral is approximately given by:

$$\epsilon_{\text{rel}} = \frac{1}{\sqrt{M N_{\text{eff}}}}$$

In general, for any model, we recommend fitting to at least $N_{\text{eff}} = 20\,P$ points. Substituting this good practice into the equation above, we find the following estimate of the number of independent sequences M:

$$M \approx \frac{1}{20\,P\,\epsilon_{\text{rel}}^2}$$

Given the simplicity and the drastic assumptions made, this is only a guideline and should not be seen as a strict rule.

From our practical experience, $M = 10$ is a low number and $M = 500$ is quite high. For $M < 10$, the results are often rather poor and possibly a bit confusing. In this low-data regime, the sampling PSD is extremely noisy. While we have validated STACIE in this low-data regime with the ACID test set, the visualization of the spectrum will not be very informative for low M.

A single molecular dynamics simulation often provides more than one independent sequence. The following table lists M (for a single simulation) for the transport properties discussed in the *Properties* section.

Transport Property	M
Bulk Viscosity	1
Thermal Conductivity	3
Ionic Electrical Conductivity	3
Shear Viscosity	5
Diffusivity	$3N_{\text{atom}}$

This means that in most cases (except for diffusivity), multiple independent simulations are required to achieve a good estimate of the transport property. While diffusivity may seem to be a very forgiving case, it is important to note that displacements of particles in a liquid are often highly correlated. STACIE assumes its inputs to be independent, which is not the case for particle velocities when studying self-diffusivity in a liquid. A correct treatment of uncertainty quantification in this case is a topic of ongoing research.

3.1.2 Step 2: Test the Sufficiency of the Number of Steps and Increase if Necessary

There is no simple way to know *a priori* the required number of steps in the input sequences. Hence, we recommend first generating inputs with about $400\,P$ steps, where P is the number of model parameters, and analyzing these inputs with STACIE. With this choice, the first $20P$ points that are ideally used for fitting will be a factor 10 below the Nyquist frequency, which is a minimal first attempt to identify the low-frequency part of the spectrum. Using these data as inputs, you will obtain a first estimate of the autocorrelation integral and its relative error. If the relative error is larger than the desired value, you can extend the input sequences with additional steps and repeat the analysis.

Note that for some applications, 400 P steps may be far too short, meaning that you will need to extend your inputs a few times before you get a clear picture of the relative error. It is not uncommon to run into problems with storage quota in this scenario. To reduce the storage requirements, *block averages* can be helpful.

In addition to the relative error, there are other indicators to monitor the quality of the results. If any of the following criteria are not met, we recommend extending the input sequences with additional steps and repeating the analysis with STACIE:

- The effective number of points used in the fit, which is determined by the cutoff frequency, should be larger than 20 times the number of model parameters.
- The Z-score computed for the regression cost and the cutoff criterion should be smaller than 2. Note that the Z-scores may also be large for other reasons than insufficient data. This may also occur when the functional form of the model can never match the data, e.g. fitting a white noise model to a spectrum that has a non-zero slope.
- When using the Lorentz model, the total simulation time should be sufficient to resolve the zero-frequency peak of the spectrum. The width of the peak can be derived from *the Lorentz model* and is $1/2\pi\tau_{\text{exp}}$, where τ_{exp} is the exponential correlation time. Because the resolution of the frequency axis of the power spectrum is $1/t_{\text{sim}}$, where t_{sim} is the total simulation time, ample frequency grid points in this first peak are guaranteed when:

  $$t_{\text{sim}} \gg 2\pi\tau_{\text{exp}}$$

  For example, $t_{\text{sim}} \approx 20\pi\tau_{\text{exp}}$ will provide a decent resolution. When using a discrete time step h, the corresponding number of steps is:

  $$N \approx 20\pi\tau_{\text{exp}}/h$$

  When STACIE estimates a large exponential correlation time, e.g. $\tau_{\text{exp}} > t_{\text{sim}}/(20\pi)$, it has derived this value from a very sharp spectral peak at zero frequency. In this case, the peak width is artificially broadened due to spectral leakage, which results in an underestimation of the correlation time. Hence, the true exponential correlation time is then even larger than the estimated value.

Finally, it is recommended that you use sequences whose length is a power of two, or at least a product of small prime numbers. NumPy's FFT algorithm used in STACIE is optimized for such sequences and becomes significantly slower for sequence lengths with large prime factors. A good strategy for adhering to this recommendation is to start with a sequence length equal to the first power of two greater than $400P$, where P is the number of model parameters. Then repeatedly double the sequence length until the analysis with STACIE indicates that the number of steps is sufficient. This approach also facilitates increasing the block size by factors of 2 *a posteriori*, when working with *block averages* to reduce storage requirements.

3.2 Reducing Storage Requirements with Block Averages

When computer simulations generate time-dependent data, they often use a discretization of the time axis with a resolution (much) higher than needed for computing the autocorrelation integral with STACIE. Storing (and processing) all these data may require excessive resources. To reduce the amount of data, we recommend taking block averages. These block averages form a new time series with a time step equal to the block size multiplied by the original time step. They reduce storage requirements by a factor equal to the block size. If the program generating the sequences does not support block averages, you can use `stacie.utils.block_average()`.

If the blocks are sufficiently small compared to the decay rate of the autocorrelation function, STACIE will produce virtually the same results. The effect of block averages can be understood by inserting them into the discrete power spectrum, using STACIE's normalization convention to

obtain the proper zero-frequency limit. Let $\hat{a}_\ell$ be the ℓ'th block average of L blocks with block size B. We can start from the power spectrum of the original sequence, $\hat{x}_n$, and then introduce approximations to rewrite it in terms of the block averages:

$$\begin{aligned}
\hat{I}_k &= Fh\frac{1}{2}\sum_{\Delta=0}^{N-1}\hat{c}_\Delta\omega_N^{-k\Delta} \\
&= \frac{Fh}{N}\frac{1}{2}\left|\sum_{n=0}^{N-1}\hat{x}_n\omega_N^{-kn}\right|^2 \\
&\approx \frac{Fh}{N}\frac{1}{2}\left|\sum_{n=0}^{N-1}\hat{a}_{\lfloor n/B\rfloor}\omega_N^{-kn}\right|^2 \\
&\approx \frac{Fh}{L}\frac{1}{2}\left|\sum_{\ell=0}^{L-1}B\hat{a}_\ell\omega_N^{-k\ell B}\right|^2 \\
&= \frac{FhB}{L}\frac{1}{2}\left|\sum_{\ell=0}^{L-1}\hat{a}_\ell\omega_L^{-k\ell}\right|^2
\end{aligned}$$

with

$$\omega_N = \exp(i2\pi/N) \qquad \omega_L = \exp(i2\pi/L) = \omega_N^B$$

The final result is the power spectrum of the block averages, where hB is the new time step and L is the sequence length.

The approximations assume that ω_N^{kn} is nearly the same $\forall\, n \in [0, B]$. Put differently, the approximation is small when

$$\omega_N^{kB} \approx 1 \quad \text{or} \quad \frac{kB}{N} \ll 1$$

The larger the block size B, the smaller the range of frequencies for which $\hat{I}_k$ is well approximated.

Depending on the model fitted to the spectrum, there are two ways to determine the appropriate block size.

1. For any model, the number of points fitted to the spectrum is recommended to be about 20 P, where P is the number of parameters in the model. This means that the block size B should be chosen such that

   $$B \ll \frac{N}{20\,P}$$

   E.g., $B = \frac{N}{400P}$ is a good choice. This practically means that there should be at least 400 P blocks. Fewer blocks will inevitably lead to significant aliasing effects.

2. When using the *Pade model*, one should ensure that the spectrum amplitudes $\hat{I}_k$ in the peak at zero frequency are not distorted by the block averages. The width of this peak in the Pade model is $1/2\pi\tau_{\text{exp}}$, and the resolution of the frequency axis of the power spectrum is $1/t_{\text{sim}}$, where $t_{\text{sim}} = Nh$ is the total simulation time. These equations can be combined with $kB/N \ll 1$ to find:

   $$B \ll \frac{2\pi\tau_{\text{exp}}}{h}$$

For example, $B = \frac{\pi \tau_{\text{exp}}}{10h}$ will ensure that the relevant spectral features are reasonably preserved in the spectrum derived from the block averages.

Just as with the required length of the input sequences, a good choice of the block size cannot be determined *a priori*. Also for the block size, a preliminary analysis with STACIE is recommended, i.e., initially without block averages.

An application of STACIE with block averages can be found in the following example notebook: *Diffusion on a Surface with Newtonian Dynamics*.

3.3 Recommendations for MD Simulations

3.3.1 Finite Size Effects

Transport properties derived from *MD* simulations of periodic systems can be affected by finite-size effects. Finite-size effects are particularly significant for diffusion coefficients. This systematic error is known to be proportional to $1/L$, where L is the length scale of the simulation box. The $1/L$ dependence allows for extrapolation to infinite box size by linear regression or by applying analytical corrections, such as the Yeh-Hummer correction [MMC+20, YH04].

3.3.2 Choice of Ensemble

The *NVE* ensemble is generally recommended for computing transport coefficients, as thermostats and barostats (used for simulations in the *NVT* and *NpT* ensembles) can interfere with system dynamics and introduce bias in transport properties [MMC+20]. For production runs, the NpT ensemble has an additional drawback: barostats introduce coordinate scaling, which directly perturbs the atomic mean squared displacements.

A good approach is to first equilibrate the system using NVT or NpT, before switching to NVE for transport property calculations. The main difficulty is that a single NVE simulation does not fully represent an NVT or NpT ensemble, even if the average temperature and pressure match perfectly. (They become equivalent in the thermodynamic limit, but we always simulate finite systems.)

NVE simulations lack the proper variance in the kinetic energy and/or volume. This issue can be addressed by performing an ensemble of independent NVE simulations that are, as a whole, representative of the NVT or NpT ensemble. Practically, this can be achieved by first performing multiple NVT or NpT equilibration runs, depending on the ensemble of interest. The final state of each equilibration run then serves as a starting point for an NVE run, **without rescaling the volume or kinetic energy**, since rescaling to the mean would artificially lower the variance in these quantities.

Note that correctly simulating multiple independent NVE runs can be technically challenging. It is not a widely used approach, not all MD codes are properly tested for it, and the default settings of some MD codes are not suitable for NVE simulations. Hence, one must always carefully check the validity of the simulations:

- First, check the conserved quantity (total energy) for drift or large fluctuations. Compared to the fluctuations of the kinetic energy, these deviations should be small.
- For the NVE simulations as a whole, the temperature distribution should be consistent with the NVT or NpT ensemble.
- Even if the NVE runs are performed correctly, one must ensure that the number of NVE runs is large enough to obtain a representative sample of the total energy distribution.

An additional challenge is the complexity of the MD workflow with restarts in different ensembles and multiple independent runs. All examples in the STACIE documentation work with NVE

production runs, show how to manage the workflow and validate the temperature distribution in detail.

3.3.3 Thermostat and Barostat Settings

For the equilibration runs discussed above, the choice of thermostat and barostat time constants is not critical, as long as the algorithms are valid (i.e., no Berendsen thermo- or barostats) and the simulations are long enough to allow for full equilibration of the system within the equilibration run. A local thermostat can be used to make the equilibration more efficient.

In some cases, e.g., to remain consistent with historical results, or because some of the challenges of NVE simulations cannot be overcome, one may still prefer to run production runs for transport properties in the NVT ensemble. When you start a new project, however, always consider using NVE production runs. If you must use NVT, studies suggest that well-tuned NVT simulations yield comparable results to NVE simulations [BS13, FMP12, KGL+22]. Basconi *et al.* recommended using a thermostat with slow relaxation times, global coupling, and continuous rescaling (as opposed to random force contributions) [BS13]. These are typically the opposite of the settings that are used for efficient equilibration runs. A drawback of slow relaxation times is that longer simulations are required to fully sample the correct ensemble.

3.3.4 Block Averages

As discussed in the *block averages* section, the use of block averages is recommended for storing simulation data. In the case of MD simulations, a safe initial block size is 10 time steps. Usually, the integration time step in MD is small enough to ensure that the fastest oscillations are sampled with 10 steps per period. It is unlikely that transport properties are affected by the dynamics at shorter time scales, so a block size of 10 time steps is a good starting point. Once you have performed an initial analysis of the data, you can adjust (increase) the block size further to optimize the data storage. If you take multiples of 10, it is easy to reprocess the initial block averages and convert them to averages over larger blocks.

CHAPTER 4

Properties Derived from the Autocorrelation Function

This section outlines the statistical and physical quantities that can be computed as the integral of an autocorrelation function. For each property, a code skeleton is provided as a starting point for your calculations. All skeletons assume that you can load the relevant input data into NumPy arrays.

First, we discuss a few properties that may be relevant to multiple scientific disciplines:

- *The uncertainty of the mean of time-correlated data*
- The exponential and integrated *autocorrelation time*

The following physicochemical transport properties can be computed as autocorrelation integrals of outputs from molecular dynamics simulations, using the so-called Green-Kubo relations [Gre52, Gre54, Hel60, Kub57]. These properties have recently been referred to as diagonal transport coefficients [PDGB25].

- *Shear viscosity*, η
- *Bulk viscosity*, η_b
- *Thermal conductivity*, κ
- *Electrical conductivity*, σ
- *Diffusion coefficient*, D

4.1 Uncertainty of the Mean of Time-Correlated Data

When data exhibits time correlations, the error of the average cannot be computed by assuming that all data are statistically independent. Because of time correlations, there are fewer independent values than the number of elements in the data.

Quantifying the *uncertainty* of averages over time-correlated data is discussed in several textbooks, e.g., Appendix D of "Understanding Molecular Simulation" by Frenkel and Smit [FS02], or Section 8.4 in the book "Computer Simulation of Liquids" (second edition) by Allen and Tildesley [AT17].

4.1.1 Derivation

The sample mean of the time-dependent sequence $\hat{\mathbf{x}}$ is:

$$\hat{x}_{\mathrm{av}} = \frac{1}{N} \sum_{n=0}^{N-1} \hat{x}_n$$

The variance of this sample mean is:

$$\mathrm{VAR}[\hat{x}_{\mathrm{av}}] = \frac{1}{N^2} \sum_{n=0}^{N-1} \sum_{m=0}^{N-1} \mathrm{COV}[\hat{x}_n\,,\, \hat{x}_m]$$

We assume that the sequence is drawn from a stationary process, such that the covariance depends only on $\Delta = n - m$:

$$c_\Delta = \mathrm{COV}[\hat{x}_n\,,\, \hat{x}_m]$$

leading to:

$$\mathrm{VAR}[\hat{x}_{\mathrm{av}}] = \frac{1}{N^2} \sum_{n=0}^{N-1} \sum_{\Delta=-n}^{N-1-n} \mathrm{COV}[\hat{x}_n\,,\, \hat{x}_{n+\Delta}]$$

To simplify this expression, we must further assume that the second summation can be extended from $\Delta = -\infty$ to $\Delta = +\infty$. This approximation is acceptable if the correlation time of the sequence is small compared to N. (In other words, we assume that c_Δ decays to zero in a small number of steps compared to N.) With this assumption, we find:

$$\mathrm{VAR}[\hat{x}_{\mathrm{av}}] \approx \frac{1}{N} \sum_{\Delta=-\infty}^{+\infty} c_\Delta$$

Analogously, when the sample mean is defined over a continuous function:

$$\hat{x}_{\mathrm{av}} = \frac{1}{t_{\mathrm{sim}}} \int_0^{t_{\mathrm{sim}}} \hat{x}(t)\,\mathrm{d}t$$

the variance of this sample mean is:

$$\mathrm{VAR}[\hat{x}_{\mathrm{av}}] = \frac{1}{t_{\mathrm{sim}}} \int_{-\infty}^{\infty} c(\Delta_t)\,\mathrm{d}\Delta_t$$

4.1.2 How to Compute with STACIE?

Because no factor 1/2 is present in the expression for the variance of the mean, the factor F must compensate for the factor 1/2 in the autocorrelation integral. Hence, we must use $F = 2$.

It is assumed that you can load the time-dependent sequences into a 2D NumPy array, where each row is a sequence and each column a time step. If you have a physical time step (in some unit of time), it is recommended that you use it as shown below, as it will result in more meaningful plots and time scales. If not available, you can set `timestep=1` or remove it from the script altogether.

```
from stacie import compute_spectrum, estimate_acint, plot_results, PadeModel

# Load your sequences and the time step.
# The details depend on your use case.
sequences, timestep = ...

# The sequences must be an array with shape (nseq, nstep).
# Each row represents one time-dependent sequence with length nstep.
# Get the total simulation time (sum over all sequences)
total_time = timestep * sequences.size

# The factor 2 is just compensating for the factor 1/2 in the autocorrelation integral.
spectrum = compute_spectrum(
    sequences,
    prefactors=2.0 / total_time,
    timestep=timestep,
    # It is assumed that the average of the sequences is not zero,
    # so we discard the DC component from the spectrum:
    include_zero_freq=False,
)
result = estimate_acint(spectrum, PadeModel([0, 2], [2]))
print("The mean", sequences.mean())
print("Error of the mean", np.sqrt(result.acint))
plot_results("error.pdf", result)
```

The spectrum at zero frequency must be excluded because it contains contributions from the mean, i.e., not only from the autocorrelation integral.

The Pade model is used here because it is nearly always a good choice for error estimates. However, if the data does not feature an exponential decay of the ACF, this model may not be appropriate. In such cases, you can use the `ExpPolyModel` instead. For more details, see the section on *spectrum models*.

A worked example can be found in the notebook *the error of the mean of a sequence generated by a Metropolis Monte Carlo algorithm*.

4.2 Integrated and Exponential Autocorrelation Time

4.2.1 Definitions

There are two definitions of the autocorrelation time [Sok97]:

1. The *integrated* autocorrelation time is derived from the autocorrelation integral:

$$\tau_{\text{int}} = \frac{\int_{-\infty}^{+\infty} c(\Delta_t)\,\mathrm{d}\Delta_t}{2c(0)} = \frac{\mathcal{I}}{Fc(0)}$$

   where $c(\Delta_t)$ is the autocorrelation function, $\mathcal{I}$ is the ACF defined with STACIE's conventions, and F is the prefactor of the autocorrelation integral, introduced in the *overview of the autocorrelation integral*.

2. The *exponential* autocorrelation time is defined as the limit of the exponential decay rate of the autocorrelation function. In STACIE's notation, this means that for large Δ_t, we have:

$$c(\Delta_t) \propto \exp\left(-\frac{|\Delta_t|}{\tau_{\text{exp}}}\right)$$

The exponential autocorrelation time characterizes the slowest mode in the input. The parameter τ_{exp} can be estimated with the *Pade model*.

Both correlation times are the same if the autocorrelation is nothing more than a two-sided exponentially decaying function:

$$c(\Delta_t) = c_0 \exp\left(-\frac{|\Delta_t|}{\tau_{\text{exp}}}\right)$$

In practice, however, the two correlation times may differ. This can happen if the input sequences are a superposition of signals with different relaxation times, or when they contain non-diffusive contributions such as oscillations at certain frequencies. It is even not guaranteed that the exponential autocorrelation time is always well-defined, e.g., when the ACF decays as a power law.

4.2.2 Which Definition Should I Use?

There is no right or wrong. Both definitions are useful and relevant for different applications.

1. The integrated correlation time is related to *the variance of the mean of a time-correlated sequence*:

   $$\text{VAR}[\hat{x}_{\text{av}}] = \frac{\text{VAR}[\hat{x}_n]}{N} \frac{2\tau_{\text{int}}}{h}$$

   The first factor is the "naive" variance of the mean, assuming that all N inputs are uncorrelated. The second factor corrects for the presence of time correlations and is called the statistical inefficiency [AT17, FC70]:

   $$s = \frac{2\tau_{\text{int}}}{h}$$

   where h is the time step. s can be interpreted as the spacing between two independent samples.

2. The exponential correlation time can be used to estimate the required length of the input sequences when computing an autocorrelation integral. The resolution of the frequency axis of the power spectrum is $1/t_{\text{sim}}$, where $t_{\text{sim}} = Nh$ is the total simulation time, h is the time step, and N the number of steps. This resolution must be fine enough to resolve the zero-frequency peak associated with the exponential decay of the autocorrelation function. The width of the peak can be derived from *the Pade model* and is $1/2\pi\tau_{\text{exp}}$. To have ample frequency grid points in this first peak, the simulation time must be sufficiently long:

   $$t_{\text{sim}} \gg 2\pi\tau_{\text{exp}}$$

   For example, $t_{\text{sim}} = 20\pi\tau_{\text{exp}}$ will provide a decent resolution.

   Of course, before you start generating the data (e.g., through simulations), the value of τ_{exp} is yet unclear. Without prior knowledge of τ_{exp}, you should first analyze preliminary data to get a first estimate of τ_{exp}, after which you can plan the data generation more carefully. More details can be found in the section on *data sufficiency*.

   If you notice that your input sequences are many orders of magnitude longer than τ_{exp}, the number of relevant frequency grid points in the spectrum can become impractically large. In this case, you can split up the input sequences into shorter parts with `stacie.utils.split()`. However, a better solution is to plan ahead more carefully and avoid sequences that are far longer than necessary. It is more efficient to generate more fully independent and shorter sequences instead.

Note that τ_{exp} is also related to the block size when working with *block averages* to reduce storage requirements of production simulations.

4.2.3 How to Compute with STACIE?

It is assumed that you can load one or (ideally) more time-dependent sequences of equal length into a 2D NumPy array `sequences`. Each row in this array is a sequence, and the columns correspond to time steps. You also need to store the time step in a Python variable. (If your data does not have a time step, just omit it from the code below.)

With these data, the autocorrelation times are computed as follows:

```
import numpy as np
from stacie import compute_spectrum, estimate_acint, plot_results, PadeModel

# Load all the required inputs, the details of which will depend on your use case.
sequences = ...
timestep = ...

# Computation with STACIE.
spectrum = compute_spectrum(sequences, timestep=timestep)
result = estimate_acint(spectrum, PadeModel([0, 2], [2]))
print("Exponential autocorrelation time", result.corrtime_exp)
print("Uncertainty of the exponential autocorrelation time", result.corrtime_exp_std)
print("Integrated autocorrelation time", result.corrtime_int)
print("Uncertainty of the integrated autocorrelation time", result.corrtime_int_std)
```

A worked example can be found in the notebook *Diffusion on a Surface with Newtonian Dynamics*. It also discusses the correlation times associated with the diffusive motion of the particles.

Note that this example assumes that the average of the input sequences is zero. If this is not the case, you should add the option `include_zero_freq=False` when calling *`stacie.spectrum.compute_spectrum()`*. This will drop the DC component from the spectrum, which is the only part of the spectrum that is affected by a non-zero average.

4.3 Shear Viscosity

The shear viscosity of a fluid is related to the autocorrelation of microscopic off-diagonal pressure tensor fluctuations as follows:

$$\eta = \frac{V}{k_{\text{B}}T}\frac{1}{2}\int_{-\infty}^{+\infty} \text{COV}[\hat{P}_{xy}(t_0)\,,\ \hat{P}_{xy}(t_0+\Delta_t)]\,\text{d}\Delta_t$$

where V is the volume of the simulation cell, k_{B} is the Boltzmann constant, T is the temperature, and $\hat{P}_{xy}$ is an instantaneous off-diagonal pressure tensor element. The time origin t_0 is arbitrary: the expected value is computed over all possible time origins.

The derivation of this result can be found in several references, e.g., Appendix C.3.2 of "Understanding Molecular Simulation" by Frenkel and Smit [FS02], Section 8.4 of "Theory of Simple Liquids" by Hansen and McDonald [HM13], or Section 13.3.1 of "Statistical Mechanics: Theory and Molecular Simulation" by Tuckerman [Tuc23].

4.3.1 Five Independent Anisotropic Pressure Contributions of an Isotropic Liquid

To the best of our knowledge, there is no prior work demonstrating how to prepare five *independent* inputs with anisotropic pressure tensor contributions that can be used as inputs to the autocorrelation integral. For instance, the result below is not mentioned in a recent comparison of methods for incorporating diagonal elements of the traceless pressure tensor [MFF23]. Since a pressure tensor has six degrees of freedom, one of which corresponds to the isotropic pressure, the remaining five should be associated with anisotropic contributions.

It is well known that the viscosity of an isotropic fluid can be derived from six off-diagonal and diagonal traceless pressure tensor elements [DE94]. However, by subtracting the isotropic term, the six components of the traceless pressure tensor become statistically correlated. For a proper *uncertainty* analysis of the estimated viscosity, STACIE requires the inputs to be statistically independent, so the Daivis and Evans equation cannot be directly used. Here, we provide a transformation of the pressure tensor that yields five independent contributions, each of which can be used individually to compute the viscosity. The average of these five viscosities is equivalent to the result of Daivis and Evans.

To facilitate working with linear transformations of pressure tensors, we adopt Voigt notation:

$$\hat{\mathbf{P}} = \begin{bmatrix} \hat{P}_{xx} & \hat{P}_{yy} & \hat{P}_{zz} & \hat{P}_{yz} & \hat{P}_{zx} & \hat{P}_{xy} \end{bmatrix}^\top$$

The transformation to the traceless form then becomes $\hat{\mathbf{P}}_{\text{tl}} = \mathbf{T}\hat{\mathbf{P}}$ with:

$$\mathbf{T} = \begin{bmatrix} \frac{2}{3} & -\frac{1}{3} & -\frac{1}{3} & & & \\ -\frac{1}{3} & \frac{2}{3} & -\frac{1}{3} & & & \\ -\frac{1}{3} & -\frac{1}{3} & \frac{2}{3} & & & \\ & & & 1 & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix}$$

This symmetric matrix is an idempotent projection matrix and has an eigendecomposition $\mathbf{T} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top$ with:

$$\text{diag}(\mathbf{\Lambda}) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} \frac{1}{\sqrt{3}} & \sqrt{\frac{2}{3}} & 0 & & & \\ \frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} & & & \\ \frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{2}} & & & \\ & & & 1 & & \\ & & & & 1 & \\ & & & & & 1 \end{bmatrix}$$

The zero eigenvalue corresponds to the isotropic component being removed. Transforming the pressure tensor to this eigenvector basis constructs five anisotropic components. Since this transformation is orthonormal, the five components remain statistically uncorrelated. It can be shown that the first two anisotropic components must be rescaled by a factor of $1/\sqrt{2}$, as in $\hat{\mathbf{P}}' = \mathbf{V}\hat{\mathbf{P}}$,

with:

$$\mathbf{V} = \begin{bmatrix} \frac{1}{\sqrt{3}} & 0 & & & \\ -\frac{1}{2\sqrt{3}} & \frac{1}{2} & & & \\ -\frac{1}{2\sqrt{3}} & -\frac{1}{2} & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix}$$

to obtain five time-dependent anisotropic pressure components that can be used as inputs to the viscosity calculation:

$$\eta = \frac{V}{k_\mathrm{B} T} \frac{1}{2} \int_{-\infty}^{+\infty} \mathrm{COV}[\hat{P}_i'(t_0)\,,\ \hat{P}_i'(t_0 + \Delta_t)]\,\mathrm{d}\Delta_t \qquad \forall\, i \in \{1, 2, 3, 4, 5\}$$

For the last three components, this result is trivial. The second component, $\hat{P}_2'$, is found by rotating the Cartesian axes 45° about the x-axis. The rotation matrix that transforms the original frame of reference into the rotated one is given by:

$$\mathbf{R} = \begin{bmatrix} 1 & & \\ & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

The pressure tensor before and after the rotation is expressed as:

$$\hat{\mathbf{P}} = \begin{bmatrix} \hat{P}_{xx} & \hat{P}_{xy} & \hat{P}_{zx} \\ \hat{P}_{xy} & \hat{P}_{yy} & \hat{P}_{yz} \\ \hat{P}_{zx} & \hat{P}_{yz} & \hat{P}_{zz} \end{bmatrix}$$

$$\mathbf{R}\hat{\mathbf{P}}\mathbf{R}^\top = \begin{bmatrix} \hat{P}_{xx} & \frac{\sqrt{2}\hat{P}_{xy}}{2} - \frac{\sqrt{2}\hat{P}_{zx}}{2} & \frac{\sqrt{2}\hat{P}_{xy}}{2} + \frac{\sqrt{2}\hat{P}_{zx}}{2} \\ \frac{\sqrt{2}\hat{P}_{xy}}{2} - \frac{\sqrt{2}\hat{P}_{zx}}{2} & \frac{\hat{P}_{yy}}{2} - \hat{P}_{yz} + \frac{\hat{P}_{zz}}{2} & \frac{\hat{P}_{yy}}{2} - \frac{\hat{P}_{zz}}{2} \\ \frac{\sqrt{2}\hat{P}_{xy}}{2} + \frac{\sqrt{2}\hat{P}_{zx}}{2} & \frac{\hat{P}_{yy}}{2} - \frac{\hat{P}_{zz}}{2} & \frac{\hat{P}_{yy}}{2} + \hat{P}_{yz} + \frac{\hat{P}_{zz}}{2} \end{bmatrix}$$

In the new axes frame, the last off-diagonal element is a proper anisotropic term, expressed as $\frac{\hat{P}_{yy}}{2} - \frac{\hat{P}_{zz}}{2}$.

For the first component, $\hat{P}_1'$, the proof is slightly more intricate. There is no rotation of the Cartesian axis frame that results in this linear combination appearing as an off-diagonal element. Instead, it is simply a scaled sum of two anisotropic stress components:

$$\hat{P}_1' = \alpha \left(\hat{P}_{xx} - \frac{\hat{P}_{yy}}{2} - \frac{\hat{P}_{zz}}{2} \right) = \alpha \left(\frac{\hat{P}_{xx}}{2} - \frac{\hat{P}_{yy}}{2} \right) + \alpha \left(\frac{\hat{P}_{xx}}{2} - \frac{\hat{P}_{zz}}{2} \right)$$

By working out the autocorrelation functions of $\hat{P}_1'$ and $\hat{P}_2'$ one finds that, for the case of an isotropic liquid, they have the same expected values if $\alpha = \frac{1}{\sqrt{3}}$. To see this, we first expand the

covariances and use $t_1 = t_0 + \Delta_t$ for brevity:

$$\begin{aligned}
&\mathrm{COV}[\hat{P}_1'(t_0),\, \hat{P}_1'(t_1)] = \\
&\quad + \alpha^2\,\mathrm{COV}[\hat{P}_{xx}(t_0),\, \hat{P}_{xx}(t_1)] - \frac{\alpha^2}{2}\,\mathrm{COV}[\hat{P}_{yy}(t_0),\, \hat{P}_{xx}(t_1)] - \frac{\alpha^2}{2}\,\mathrm{COV}[\hat{P}_{zz}(t_0),\, \hat{P}_{xx}(t_1)] \\
&\quad - \frac{\alpha^2}{2}\,\mathrm{COV}[\hat{P}_{xx}(t_0),\, \hat{P}_{yy}(t_1)] + \frac{\alpha^2}{4}\,\mathrm{COV}[\hat{P}_{yy}(t_0),\, \hat{P}_{yy}(t_1)] + \frac{\alpha^2}{4}\,\mathrm{COV}[\hat{P}_{zz}(t_0),\, \hat{P}_{yy}(t_1)] \\
&\quad - \frac{\alpha^2}{2}\,\mathrm{COV}[\hat{P}_{xx}(t_0),\, \hat{P}_{zz}(t_1)] + \frac{\alpha^2}{4}\,\mathrm{COV}[\hat{P}_{yy}(t_0),\, \hat{P}_{zz}(t_1)] + \frac{\alpha^2}{4}\,\mathrm{COV}[\hat{P}_{zz}(t_0),\, \hat{P}_{zz}(t_1)]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{COV}[\hat{P}_2'(t_0),\, \hat{P}_2'(t_1)] = \\
&\quad + \frac{1}{4}\,\mathrm{COV}[\hat{P}_{yy}(t_0),\, \hat{P}_{yy}(t_1)] - \frac{1}{4}\,\mathrm{COV}[\hat{P}_{yy}(t_0),\, \hat{P}_{zz}(t_1)] \\
&\quad - \frac{1}{4}\,\mathrm{COV}[\hat{P}_{zz}(t_0),\, \hat{P}_{yy}(t_1)] + \frac{1}{4}\,\mathrm{COV}[\hat{P}_{zz}(t_0),\, \hat{P}_{zz}(t_1)]
\end{aligned}$$

Because the liquid is isotropic, permutations of Cartesian axes do not affect the expected values, which simplifies the expressions to:

$$\begin{aligned}
&\mathrm{COV}[\hat{P}_1'(t_0), \hat{P}_1'(t_1)] \\
&\quad + \frac{3\alpha^2}{4}\,\mathrm{COV}[\hat{P}_{xx}(t_0),\, \hat{P}_{xx}(t_1)] - \frac{3\alpha^2}{4}\,\mathrm{COV}[\hat{P}_{xx}(t_0),\, \hat{P}_{yy}(t_1)] \\
&\quad - \frac{3\alpha^2}{4}\,\mathrm{COV}[\hat{P}_{yy}(t_0),\, \hat{P}_{xx}(t_1)] + \frac{3\alpha^2}{4}\,\mathrm{COV}[\hat{P}_{yy}(t_0),\, \hat{P}_{yy}(t_1)]
\end{aligned}$$

$$\begin{aligned}
&\mathrm{COV}[\hat{P}_2'(t_0), \hat{P}_2'(t_1)] = \\
&\quad + \frac{1}{4}\,\mathrm{COV}[\hat{P}_{xx}(t_0),\, \hat{P}_{xx}(t_1)] - \frac{1}{4}\,\mathrm{COV}[\hat{P}_{xx}(t_0),\, \hat{P}_{yy}(t_1)] \\
&\quad - \frac{1}{4}\,\mathrm{COV}[\hat{P}_{yy}(t_0),\, \hat{P}_{xx}(t_1)] + \frac{1}{4}\,\mathrm{COV}[\hat{P}_{yy}(t_0),\, \hat{P}_{yy}(t_1)]
\end{aligned}$$

These two expected values are consistent when $\alpha^2 = 1/3$.

Finally, it is noteworthy that the average viscosity over the five components proposed here is equivalent to the equation proposed by Daivis and Evans [DE94]:

$$\eta = \frac{1}{5}\frac{V}{k_\mathrm{B}T}\frac{1}{2}\int_{-\infty}^{+\infty}\frac{1}{2}\,\mathrm{E}\left[\hat{\mathbf{P}}_\mathrm{tl}(t_0) : \hat{\mathbf{P}}_\mathrm{tl}(t_0 + \Delta_t)\right]\,\mathrm{d}\Delta_t$$

(This is Eq. A5 in their paper rewritten in our notation.) To show the equivalence, we expand the inner product of the traceless pressure tensor, using $\hat{P}_{\mathrm{tl},xx} = \frac{1}{3}(2\hat{P}_{xx} - \hat{P}_{yy} - \hat{P}_{zz})$ and similar

definitions for the two other Cartesian components:

$$\begin{aligned}\frac{1}{2}\hat{\mathbf{P}}_{\mathrm{tl}}(t_0) : \hat{\mathbf{P}}_{\mathrm{tl}}(t_1) = \\ &+ \frac{1}{3}\hat{P}_{xx}(t_0)\,\hat{P}_{xx}(t_1) - \frac{1}{6}\hat{P}_{xx}(t_0)\,\hat{P}_{yy}(t_1) - \frac{1}{6}\hat{P}_{xx}(t_0)\,\hat{P}_{zz}(t_1) \\ &- \frac{1}{6}\hat{P}_{yy}(t_0)\,\hat{P}_{xx}(t_1) + \frac{1}{3}\hat{P}_{yy}(t_0)\,\hat{P}_{yy}(t_1) - \frac{1}{6}\hat{P}_{yy}(t_0)\,\hat{P}_{zz}(t_1) \\ &- \frac{1}{6}\hat{P}_{zz}(t_0)\,\hat{P}_{xx}(t_1) - \frac{1}{6}\hat{P}_{zz}(t_0)\,\hat{P}_{yy}(t_1) + \frac{1}{3}\hat{P}_{zz}(t_0)\,\hat{P}_{zz}(t_1) \\ &+ \hat{P}_{xy}(t_0)\,\hat{P}_{xy}(t_1) + \hat{P}_{yz}(t_0)\,\hat{P}_{yz}(t_1) + \hat{P}_{zx}(t_0)\,\hat{P}_{zx}(t_1)\end{aligned}$$

We can similarly work out the products $\hat{P}'_i(t_0)\,\hat{P}'_i(t_1)$ for $i = 1, \ldots, 5$ appearing in our proposed viscosity equation:

$$\eta = \frac{1}{5}\frac{V}{k_{\mathrm{B}}T}\frac{1}{2}\int_{-\infty}^{+\infty}\sum_{i=1}^{5}\mathrm{COV}[\hat{P}'_i(t_0)\,,\ \hat{P}'_i(t_1)]\,\mathrm{d}\Delta_t$$

Working out the expansion of the five terms in Cartesian pressure tensor components yields:

$$\begin{aligned}\hat{P}'_1(t_0)\,\hat{P}'_1(t_1) = \\ &+ \frac{1}{3}\hat{P}_{xx}(t_0)\,\hat{P}_{xx}(t_1) - \frac{1}{6}\hat{P}_{xx}(t_0)\,\hat{P}_{yy}(t_1) - \frac{1}{6}\hat{P}_{xx}(t_0)\,\hat{P}_{zz}(t_1) \\ &- \frac{1}{6}\hat{P}_{yy}(t_0)\,\hat{P}_{xx}(t_1) + \frac{1}{12}\hat{P}_{yy}(t_0)\,\hat{P}_{yy}(t_1) + \frac{1}{12}\hat{P}_{yy}(t_0)\,\hat{P}_{zz}(t_1) \\ &- \frac{1}{6}\hat{P}_{zz}(t_0)\,\hat{P}_{xx}(t_1) + \frac{1}{12}\hat{P}_{zz}(t_0)\,\hat{P}_{yy}(t_1) + \frac{1}{12}\hat{P}_{zz}(t_0)\,\hat{P}_{zz}(t_1)\end{aligned}$$

$$\begin{aligned}\hat{P}'_2(t_0)\,\hat{P}'_2(t_1) = \\ &+ \frac{1}{4}\hat{P}_{yy}(t_0)\,\hat{P}_{yy}(t_1) - \frac{1}{4}\hat{P}_{yy}(t_0)\,\hat{P}_{zz}(t_1) \\ &- \frac{1}{4}\hat{P}_{zz}(t_0)\,\hat{P}_{yy}(t_1) + \frac{1}{4}\hat{P}_{zz}(t_0)\,\hat{P}_{zz}(t_1)\end{aligned}$$

$$\hat{P}'_3(t_0)\,\hat{P}'_3(t_1) = \hat{P}_{yz}(t_0)\,\hat{P}_{yz}(t_1)$$

$$\hat{P}'_4(t_0)\,\hat{P}'_4(t_1) = \hat{P}_{zx}(t_0)\,\hat{P}_{zx}(t_1)$$

$$\hat{P}'_5(t_0)\,\hat{P}'_5(t_1) = \hat{P}_{xy}(t_0)\,\hat{P}_{xy}(t_1)$$

Adding these five contributions together reproduces the exact same expansion as derived by Daivis and Evans.

Using the five anisotropic components, as proposed here, offers significant advantages. It explicitly defines the number of independent sequences used as input, enabling precise uncertainty quantification.

4.3.2 How to Compute with STACIE?

It is assumed that you can load the time-dependent pressure tensor components (both diagonal and off-diagonal) into a 2D NumPy array `pcomps`, where each column corresponds to a time step. Each row corresponds to an individual pressure tensor component in the order $\hat{P}_{xx}$, $\hat{P}_{yy}$, $\hat{P}_{zz}$, $\hat{P}_{zx}$, $\hat{P}_{yz}$, and $\hat{P}_{xy}$ (same order as in Voigt notation).

You also need to store the cell volume, temperature, Boltzmann constant, and time step in Python variables, all in consistent units. With these requirements, the shear viscosity can be computed as follows:

```
import numpy as np
from stacie import compute_spectrum, estimate_acint, plot_results, PadeModel, UnitConfig

# Load all the required inputs, the details of which will depend on your use case.
pcomps = ...
volume, temperature, boltzmann_const, timestep = ...

# Convert pressure components to five independent components.
# This is the optimal usage of pressure information
# and it informs STACIE of the number of independent inputs.
indep_pcomps = np.array([
    (pcomps[0] - 0.5 * pcomps[1] - 0.5 * pcomps[2]) / np.sqrt(3),
    0.5 * pcomps[1] - 0.5 * pcomps[2],
    pcomps[3],
    pcomps[4],
    pcomps[5],
])

# Actual computation with STACIE.
spectrum = compute_spectrum(
    indep_pcomps,
    prefactors=volume / (temperature * boltzmann_const),
    timestep=timestep,
)
result = estimate_acint(spectrum, PadeModel([0, 2], [2]))
print("Shear viscosity:", result.acint)
print("Uncertainty of the shear viscosity:", result.acint_std)

# The unit configuration assumes SI units are used systematically.
# You may need to adapt this to the units of your data.
uc = UnitConfig(
    acint_unit_str="Pa s",
    time_unit=1e-12,
    time_unit_str="ps",
    freq_unit=1e12,
    freq_unit_str="THz",
)
plot_results("shear_viscosity.pdf", result, uc)
```

This script can be trivially extended to combine data from multiple trajectories, by stacking additional independent pressure contributions as rows in the `indep_pcomps` array.

A worked example can be found in the notebook *Shear viscosity of a Lennard-Jones Liquid Near the Triple Point (LAMMPS)*

4.4 Bulk Viscosity

The bulk viscosity of a fluid is related to the autocorrelation of isotropic pressure fluctuations as follows:

$$\eta_b = \frac{V}{k_\mathrm{B}T}\frac{1}{2}\int_{-\infty}^{+\infty} \mathrm{COV}[\hat{P}_\mathrm{iso}(t_0)\,,\ \hat{P}_\mathrm{iso}(t_0+\Delta_t)]\,\mathrm{d}\Delta_t$$

where V represents the volume of the simulation cell, k_B is the Boltzmann constant, T is the temperature, and $\hat{P}_\mathrm{iso}$ is the instantaneous isotropic pressure. The time origin t_0 is arbitrary: the expected value is computed over all possible time origins.

The derivation of this result can be found in several references, e.g., Section 8.5 of "Theory of Simple Liquids" by Hansen and McDonald [HM13], or Section 2.7 of "Computer Simulation of Liquids" by Allen and Tildesley [AT17].

As will be shown below, one must take into account that the average pressure is not zero. For STACIE, there is no need to subtract the average pressure first. Instead, you can simply drop the DC component from the spectrum, by setting the `include_zero_freq=False` option when computing the spectrum.

4.4.1 How to Compute with STACIE?

It is assumed that you can load the diagonal, time-dependent pressure tensor components into a 2D NumPy array `pcomps`, where each column corresponds to a time step. Each row corresponds to a diagonal pressure tensor component: $\hat{P}_{xx}$, $\hat{P}_{yy}$, and $\hat{P}_{zz}$. The same array used for *shear viscosity* can be reused here, but only the first three rows are necessary.

You also need to store the cell volume, temperature, Boltzmann constant, and time step in Python variables, all in consistent units. With these requirements, the bulk viscosity can be computed as follows:

```
import numpy as np
from stacie import compute_spectrum, estimate_acint, plot_results, PadeModel, UnitConfig

# Load all the required inputs, the details of which will depend on your use case.
pcomps = ...
volume, temperature, boltzmann_const, timestep = ...

# Convert pressure components to the isotropic pressure
piso = (pcomps[0] + pcomps[1] + pcomps[2]) / 3

# Actual computation with STACIE.
spectrum = compute_spectrum(
    piso,
    prefactors=volume / (temperature * boltzmann_const),
    timestep=timestep,
    # Drop the DC component to account for non-zero average pressure:
    include_zero_freq=False,
)
result = estimate_acint(spectrum, PadeModel([0, 2], [2]))
print("Bulk viscosity:", result.acint)
print("Uncertainty of the bulk viscosity:", result.acint_std)

# The unit configuration assumes SI units are used systematically.
# You may need to adapt this to the units of your data.
uc = UnitConfig(
    acint_unit_str="Pa s",
    time_unit=1e-12,
    time_unit_str="ps",
    freq_unit=1e12,
    freq_unit_str="THz",
)
plot_results("bulk_viscosity.pdf", result, uc)
```

This script can be trivially extended to combine data from multiple trajectories, by defining `piso` as a 2D array, where each row corresponds to an additional independent isotropic pressure time series.

A worked example can be found in the notebook *Bulk viscosity of a Lennard-Jones Liquid Near the Triple Point (LAMMPS)*

4.5 Thermal Conductivity

The thermal conductivity of a system is related to the autocorrelation of the heat flux as follows:

$$\kappa = \frac{1}{V k_\mathrm{B} T^2} \frac{1}{d} \sum_{\alpha=1}^{d} \frac{1}{2} \int_{-\infty}^{+\infty} \mathrm{COV}[\hat{J}^\mathrm{h}_\alpha(t_0)\,,\, \hat{J}^\mathrm{h}_\alpha(t_0 + \Delta_t)]\,\mathrm{d}\Delta_t$$

where V is the volume of the simulation cell, k_B is the Boltzmann constant, T is the temperature, d is the dimensionality of the system (usually 3), and $\hat{J}^\mathrm{h}_\alpha$ is the instantaneous microscopic heat current along one of the Cartesian directions. The time origin t_0 is arbitrary: the expected value is computed over all possible time origins.

Our notation follows the convention used by most molecular dynamics packages, which output the microscopic (instantaneous) heat current vector, $\hat{\mathbf{J}}^\mathrm{h}$, as an extensive quantity, with units of power times length (Wm). This differs from the macroscopic heat flux, which is an intensive quantity with units of power per area (Wm^{-2}). If you need to post-process microscopic heat flux data provided as an intensive quantity (i.e., a flux density), use the prefactor $V/(k_\mathrm{B}T^2)$ in the formula above instead of $1/(V k_\mathrm{B} T^2)$.

The derivation of this result can be found in Section 8.5 of "Theory of Simple Liquids" by Hansen and McDonald [HM13].

> **Warning**
>
> The LAMMPS `compute/heat flux` command is reported to produce unphysical results when many-body interactions (e.g., angle, dihedral, impropers) are present [JWB+19], [SMKO19], [BBW19], [SMKO21]. This command only treats pairwise interactions correctly. If this is relevant, one should use the `compute heat/flux` command with `compute centroid/stress/atom`. For systems with only two-body interactions, the `compute heat/flux` command with the `compute stress/atom` command is sufficient. Molecular liquids are practically always simulated with some many-body terms, and thus require the `compute centroid/stress/atom` command.

4.5.1 How to Compute with STACIE?

It is assumed that you can load the time-dependent heat flux components into a 2D NumPy array `heatflux`, where each column corresponds to a time step. Each row corresponds to a single heat flux component: $\hat{J}_x$, $\hat{J}_y$, and $\hat{J}_z$.

You also need to store the cell volume, temperature, Boltzmann constant, and time step in Python variables, all in consistent units. With these requirements, the thermal conductivity can be computed as follows:

```
import numpy as np
from stacie import compute_spectrum, estimate_acint, plot_results, PadeModel, UnitConfig
```

(continues on next page)

(continued from previous page)

```
# Load all the required inputs, the details of which will depend on your use case.
heatflux = ...
volume, temperature, boltzmann_const, timestep = ...

# Actual computation with STACIE.
# Note that the average spectrum over the three components is implicit.
# There is no need to include 1/3 here.
spectrum = compute_spectrum(
    heatflux,
    prefactors=1.0 / (volume * temperature**2 * boltzmann_const),
    timestep=timestep,
)
result = estimate_acint(spectrum, PadeModel([0, 2], [2]))
print("Thermal conductivity", result.acint)
print("Uncertainty of the thermal conductivity", result.acint_std)

# The unit configuration assumes SI units are used systematically.
# You may need to adapt this to the units of your data.
uc = UnitConfig(
    acint_symbol="κ",
    acint_unit_str="W m$^{-1}$ K$^{-1}$",
    time_unit=1e-12,
    time_unit_str="ps",
    freq_unit=1e12,
    freq_unit_str="THz",
)
plot_results("thermal_conductivity.pdf", result, uc)
```

This script is trivially extended to combine data from multiple trajectories, by stacking additional heat flux components as rows in the `heatflux` array.

A worked example can be found in the notebook *Thermal Conductivity of a Lennard-Jones Liquid Near the Triple Point (LAMMPS)*.

4.6 Ionic Electrical Conductivity

The ionic electrical conductivity of a system is related to the autocorrelation of the charge current as follows:

$$\sigma = \frac{1}{V k_\mathrm{B} T} \frac{1}{d} \sum_{\alpha=1}^{d} \frac{1}{2} \int_{-\infty}^{+\infty} \mathrm{COV}[\hat{J}^\mathrm{c}_\alpha(t_0)\,,\, \hat{J}^\mathrm{c}_\alpha(t_0 + \Delta_t)]\,\mathrm{d}\Delta_t$$

where V is the volume of the simulation cell, k_B is the Boltzmann constant, T is the temperature, d is the dimensionality of the system, and $\hat{J}^\mathrm{c}_i$ is the instantaneous charge current along one of the Cartesian directions. The time origin t_0 is arbitrary: the expected value is computed over all possible time origins.

The derivation of this result can be found in Appendix C.3.1 of "Understanding Molecular Simulation" by Frenkel and Smit [FS02], or Section 7.7 of "Theory of Simple Liquids" by Hansen and McDonald [HM13].

If your simulation code does not print out the charge current, it can also be derived from the

velocities, $\hat{\mathbf{v}}_n(t)$, and the net charges, q_n, of the charge carriers as follows:

$$\hat{\mathbf{J}}(t) = \sum_{n=1}^{N_q} q_n \hat{\mathbf{v}}_n(t)$$

where N_q is the number of charge carriers. The charge current can also be interpreted as the time derivative of the instantaneous dipole moment of the system.

In the case of molecular ions, the center-of-mass velocity can be used, but this is not critical. You will get the same conductivity (possibly with slightly larger uncertainties) when using the velocity of any single atom in a molecular ion instead. The charges of ions must be integer multiples of the elementary charge [GB19].

4.6.1 Nernst-Einstein Approximation

The electrical conductivity is related to the (correlated) diffusion of the charge carriers. When correlations between the ions are neglected, one obtains the Nernst-Einstein approximation of the conductivity in terms of the self-diffusion coefficients of the ions. We include the derivation here because a consistent treatment of the pre-factors can be challenging. (Literature references are not always consistent due to differences in notation.) Our derivation is general, i.e., for an arbitrary number of different *types* of charge carriers, which are not restricted to monovalent ions.

First, insert the expression for the charge current into the conductivity and then bring the sums out of the integral:

$$\sigma = \frac{1}{V k_\mathrm{B} T} \frac{1}{d} \sum_{i=1}^{d} \sum_{n=1}^{N_q} \sum_{m=1}^{N_q} q_n q_m \frac{1}{2} \int_{-\infty}^{+\infty} \mathrm{COV}[\hat{v}_{n,i}(t_0)\,,\, \hat{v}_{m,i}(t_0 + \Delta_t)]\,\mathrm{d}\Delta_t$$

In the Nernst-Einstein approximation, all correlations between ion velocities (even of the same type) are neglected by discarding all off-diagonal terms in the double sum over n and m.

$$\sigma \approx \sigma_{NE} = \frac{1}{V k_\mathrm{B} T} \sum_{n=1}^{N_q} q_n^2 \frac{1}{d} \sum_{i=1}^{d} \frac{1}{2} \int_{-\infty}^{+\infty} \mathrm{COV}[\hat{v}_{n,i}(t_0)\,,\, \hat{v}_{n,i}(t_0 + \Delta_t)]\,\mathrm{d}\Delta_t$$

To further connect this equation to diffusion coefficients, the number of *types* of charge carriers is called K. Each type $k \in \{1, \ldots, K\}$ has a set of ions S_k with charge q_k. The number of ions in each set is $N_k = |S_k|$. With these conventions, we can rewrite the equation as:

$$\sigma_{NE} = \frac{1}{V k_\mathrm{B} T} \sum_{k=1}^{K} q_k^2 N_k \left(\frac{1}{N_k d} \sum_{i=1}^{d} \sum_{n \in S_k} \frac{1}{2} \int_{-\infty}^{+\infty} \mathrm{COV}[\hat{v}_{n,i}(t_0)\,,\, \hat{v}_{n,i}(t_0 + \Delta_t)]\,\mathrm{d}\Delta_t \right)$$

The part between parentheses is the self-diffusion coefficient of the ions of type k. Finally, we get:

$$\sigma_\mathrm{NE} = \frac{1}{k_\mathrm{B} T} \sum_{k=1}^{K} q_k^2 \rho_k D_k$$

where ρ_k and D_k are the concentration and the diffusion coefficient of charge carrier k, respectively. The Nernst-Einstein approximation may not seem useful because it neglects correlated motion between different types of charge carriers. (The effect may be large!) Nevertheless, a comparison of the Nernst-Einstein approximation to the actual conductivity can help to quantify the degree of such correlations. [SSWZ20]

4.6.2 How to Compute with STACIE?

It is assumed that you can load the time-dependent ion velocity components into a NumPy array `ionvels`. In the example below, this is a three-index array, where the first index is for the ion, the second for the Cartesian component, and the last for the time step. To compute the charge current, you need to put the charges of the ions in an array `charges`.

You also need to store the cell volume, temperature, Boltzmann constant, and time step in Python variables, all in consistent units. With these requirements, the ionic electrical conductivity can be computed as follows:

```
import numpy as np
from stacie import compute_spectrum, estimate_acint, plot_results, ExpPolyModel, UnitConfig

# Load all the required inputs, the details of which will depend on your use case.
# We assume ionvels has shape `(nstep, natom, ncart)`
# and charges is a 1D array with shape `(natom,)`
ionvels = ...
charges = ...
volume, temperature, boltzmann_const, timestep = ...

# Compute the charge current
chargecurrent = np.einsum("ijk,j->ki", ionvels, charges)

# Actual computation with STACIE.
# Note that the average spectrum over the three components is implicit.
# There is no need to include 1/3 here.
# Note that the zero-frequency component is usually not reliable
# because usually the total momentum is constrained or conserved.
spectrum = compute_spectrum(
    chargecurrent,
    prefactors=1.0 / (volume * temperature * boltzmann_const),
    timestep=timestep,
    include_zero_freq=False,
)
# The unit configuration assumes SI units are used systematically.
# You may need to adapt this to the units of your data.
uc = UnitConfig(
    acint_unit_str="S m$^{-1}$",
    time_unit=1e-12,
    time_unit_str="ps",
    freq_unit=1e12,
    freq_unit_str="THz",
)
# Actual analysis with STACIE.
result = estimate_acint(spectrum, ExpPolyModel([0, 1, 2]), unit_config=uc, verbose=True)
print("Electrical conductivity", result.acint)
print("Uncertainty of the electrical conductivity", result.acint_std)

plot_results("electrical_conductivity.pdf", result, uc)
```

A common scenario is that you have the positions of the ions instead of their velocities, or the dipole moment of the system. Even if this information is stored with large time intervals, you can still compute the ionic conductivity by first computing a *block-averaged* charge current.

$$\bar{\mathbf{J}}^{\mathrm{c}}_{i+1/2} = \frac{1}{\Delta_t} \int_{t_i}^{t_{i+1}} \hat{\mathbf{J}}^{\mathrm{c}}(t)\,\mathrm{d}t = \frac{\hat{\mathbf{p}}_{i+1} - \hat{\mathbf{p}}_i}{\Delta_t}$$

where $\hat{\mathbf{p}}_i$ is the dipole moment at time step i. Formally, this resembles a finite difference approximation of the charge current, except that Δ_t can be large.

> **Warning**
>
> If you have sampled velocities or charge currents directly with a large sampling interval, they cannot be used because they are not proper *block averages*. In this case, your data violates the Niquist-Shannon sampling theorem, and the results will be perturbed by aliasing artifacts. The block averages satisfy the sampling theorem by construction because they average out the high-frequency components.

For example, if the dipole moment is stored every 10 ps in eÅ, you can compute the charge current as follows:

```
import numpy as np
from scipy.constants import value
from stacie import compute_spectrum, estimate_acint, plot_results, ExpPolyModel, UnitConfig

# Values of units of external data in "internal" SI base units.
ELCHARGE = value("elementary charge")
PICOSECOND = 1e-12
ANGSTROM = 1e-10
TERAHERTZ = 1e12
BOLTZMANN_CONST = value("Boltzmann constant")

# Load all the required inputs, the details of which will depend on your use case.
# It is assumed that each row of `dipoles` corresponds to a single Cartesian component
# and each column to a time step.
dipoles = (...) * ELCHARGE * ANGSTROM
timestep = 10.0 * PICOSECOND
volume, temperature  = ...

# Compute charge current, as if you are using finite difference approximation.
chargecurrent = np.diff(dipoles, axis=1) / timestep

# Computation with STACIE, as before.
spectrum = compute_spectrum(
    chargecurrent,
    prefactors=1.0 / (volume * temperature * BOLTZMANN_CONST),
    timestep=timestep,
    include_zero_freq=False,
)
uc = UnitConfig(
    acint_unit_str="S m$^{-1}$",
    time_unit=PICOSECOND,
    time_unit_str="ps",
    freq_unit=TERAHERTZ,
    freq_unit_str="THz",
)
result = estimate_acint(spectrum, ExpPolyModel([0, 1, 2]), verbose=True, unit_config=uc)
plot_results("electrical_conductivity.pdf", result, uc)
```

A worked example can be found in the notebook *Ionic Conductivity and Self-diffusivity in Molten Sodium Chloride at 1100 K (OpenMM)*

4.7 Diffusion Coefficient

The diffusion coefficient (or diffusivity) of a set of N particles in d dimensions is given by:

$$D = \frac{1}{N\,d}\frac{1}{2}\int_{-\infty}^{+\infty}\sum_{n=1}^{N}\sum_{\alpha=1}^{d}\mathrm{COV}[\hat{v}_{n,\alpha}(t_0),\ \hat{v}_{n,\alpha}(t_0+\Delta_t)]\,\mathrm{d}\Delta_t$$

where $\hat{v}_{n,\alpha}(t)$ is the α-th Cartesian component of the time-dependent velocity of particle n. For molecular systems, the center-of-mass velocities are typically used.

For a simple fluid, the result is called the self-diffusion coefficient or self-diffusivity. The same expression applies to the diffusion coefficient of components of a mixture or guest molecules in porous media.

Note that this definition is valid only if the particles of interest exhibit diffusive motion. If they oscillate around a fixed center, the zero-frequency component of the velocity autocorrelation spectrum will approach zero, resulting in a diffusion coefficient of zero. This scenario may occur when the diffusion is governed by an activated hopping process, and the simulation is too short to capture such rare events.

The derivation of this result can be found in several references, e.g., Section 4.4.1 of "Understanding Molecular Simulation" by Frenkel and Smit [FS02], Section 7.7 of "Theory of Simple Liquids" by Hansen and McDonald [HM13], or Section 13.3.2 of "Statistical Mechanics: Theory and Molecular Simulation" by Tuckerman [Tuc23].

4.7.1 How to Compute with STACIE?

It is assumed that you can load the particle velocities into a 2D NumPy array `velocities`. Each row of this array corresponds to a single Cartesian component of a particle's velocity, while each column corresponds to a specific time step. You should also store the time step in a Python variable. The diffusion coefficient can then be computed as follows:

```
import numpy as np
from stacie import compute_spectrum, estimate_acint, plot_results, ExpPolyModel, UnitConfig

# Load all the required inputs, the details of which will depend on your use case.
velocities = ...
timestep = ...

# Computation with STACIE.
# Note that the factor 1/(N*d) is implied:
# the average spectrum over all velocity components is computed.
# Note that the zero-frequency component is usually not reliable
# because typically the total momentum is constrained or conserved.
spectrum = compute_spectrum(
    velocities,
    prefactors=1.0,
    timestep=timestep,
    include_zero_freq=False,
)
# The unit configuration assumes SI units are used systematically.
# You may need to adapt this to the units of your data.
uc = UnitConfig(
    acint_symbol="D",
    acint_unit_str="m$^2$/s",
```

(continues on next page)

(continued from previous page)

```
    time_unit=1e-12,
    time_unit_str="ps",
    freq_unit=1e12,
    freq_unit_str="THz",
)
# Actual analysis with STACIE.
result = estimate_acint(spectrum, ExpPolyModel([0, 1, 2]), unit_config=uc, verbose=True)
print("Diffusion coefficient", result.acint)
print("Uncertainty of the diffusion coefficient", result.acint_std)


plot_results("diffusion_coefficient.pdf", result, uc)
```

A worked example can be found in the notebook *Diffusion on a Surface with Newtonian Dynamics*.

One can also use particle positions and derive *block-averaged* velocities by applying something that looks like a finite difference approximation:

$$\bar{\mathbf{v}}_{i+1/2} = \frac{1}{\Delta_t} \int_{t_i}^{t_{i+1}} \hat{\mathbf{v}}(t)\,\mathrm{d}t = \frac{\hat{\mathbf{r}}_{i+1} - \hat{\mathbf{r}}_i}{\Delta_t}$$

where the index i runs over the recorded time steps and Δ_t is time between two recorded steps. Formally, this resembles a finite difference approximation of the velocity, except that Δ_t can be large.

Warning

If you have sampled particle velocities directly with a large sampling interval, they cannot be used because they are not proper *block averages*. In this case, your data violates the Niquist-Shannon sampling theorem, and the results will be perturbed by aliasing artifacts. The block averages satisfy the sampling theorem by construction because they average out the high-frequency components.

For example, if the trajectory data contains positions in Å, recorded every 10 ps, and you want to compute the diffusion coefficient in m²/s, the code would be:

```
import numpy as np
from stacie import compute_spectrum, estimate_acint, ExpPolyModel, UnitConfig, plot_results

# Define units of external data in "internal" SI base units.
PICOSECOND = 1e-12
ANGSTROM = 1e-10
TERAHERTZ = 1e12

# Load all the required inputs, the details of which will depend on your use case.
# It is assumed that each row of `positions` corresponds to a single Cartesian component
# of a particle's position, while each column corresponds to a specific time step.
positions = (...) * ANGSTROM
timestep = 10.0 * PICOSECOND

# Compute velocities, as if you are using finite difference approximation.
# In fact, these are block-averaged velocities.
velocities = np.diff(positions, axis=1) / timestep
```

(continues on next page)

(continued from previous page)

```
# Computation with STACIE, as before.
spectrum = compute_spectrum(
    velocities,
    prefactors=1.0,
    timestep=timestep,
    include_zero_freq=False,
)
uc = UnitConfig(
    acint_symbol="D",
    acint_unit_str="m$^2$/s",
    time_unit=PICOSECOND,
    time_unit_str="ps",
    freq_unit=TERAHERTZ,
    freq_unit_str="THz",
)
result = estimate_acint(spectrum, ExpPolyModel([0, 1, 2]), verbose=True, unit_config=uc)
plot_results("diffusion_coefficient.pdf", result, uc)
```

CHAPTER 5

Worked Examples

All the examples are also available as Jupyter notebooks and can be downloaded as one ZIP archive here:

> Gözdenur Toraman, Toon Verstraelen, "Example Trajectory Data and Jupyter Notebooks Showing How to Compute Various Properties with STACIE" June 2025 https://doi.org/10.5281/zenodo.15543902

Warning

The ZIP file will contain executable notebooks and simulation outputs used by the examples. Hyperlinks from these notebooks to the rest of the documentation and literature references will not work.

This documentation contains the rendered notebooks, including all outputs, in the following sections. We recommend starting with the minimal example, as it is the easiest to run and understand. This example thoroughly explains STACIE's output and how to interpret the plots. The other examples produce similar outputs and plots, but the meaning of all outputs is not repeated in each example.

The first few notebooks are completely self-contained. They generate the data and analyze it with STACIE:

5.1 Minimal Example

The main goal of this example is to demonstrate how to use STACIE with a minimal, self-contained example. First, the properties of a basic Markov process are discussed, and then data is generated using this process. (A detailed derivation of the analytical results is provided in the last section.) The Markov chains are analyzed using two *models*, followed by some comments on their applicability.

A secondary goal is to thoroughly discuss the plots generated by STACIE, which can help detect problems with the analysis or input data.

5.1.1 Library Imports and Matplotlib Configuration

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from stacie import (
    compute_spectrum,
    estimate_acint,
    ExpPolyModel,
    LorentzModel,
    UnitConfig,
    plot_extras,
    plot_fitted_spectrum,
)
```

```
mpl.rc_file("matplotlibrc")
%config InlineBackend.figure_formats = ["svg"]
```

5.1.2 The Markov Process

We use the following simple discrete-time Markov process:

$$\hat{x}_{n+1} = \alpha \hat{x}_n + \beta \hat{z}_n$$

where $\hat{z}_n$ are uncorrelated standard normal random variables, and α and β are real constants. The parameter α controls the autocorrelation of the process, with $0 < \alpha < 1$.

One can show that the autocorrelation function of this process is given by

$$c_\Delta = \frac{\beta^2}{1-\alpha^2} \alpha^{|\Delta|}$$

The variance, autocorrelation integral (with $F = 1$ and $h = 1$) and integrated correlation time are respectively:

$$\sigma^2 = \frac{\beta^2}{1-\alpha^2}$$

$$\mathcal{I} = \frac{1}{2}\left(\frac{\beta}{1-\alpha}\right)^2$$

$$\tau_{\text{int}} = \frac{\mathcal{I}}{F\, c_0} = \frac{1}{2}\frac{1+\alpha}{1-\alpha}$$

Derivations of these equations can be found in the final section of this notebook.

The example below uses the parameters $\alpha = 31/33$ and $\beta = \sqrt{8/1089}$, for which we expect $\mathcal{I} = 1$ and $\tau_{\text{int}} = 16$.

5.1.3 Data generation

The following code cell implements 64 independent realizations of the Markov process of 32768 steps each. This implementation vectorizes over independent sequences, which is much faster than generating them one by one.

```
nseq = 64
nstep = 1024 * 32
alpha = 31 / 33
beta = np.sqrt(8 / 1089)
std = beta / np.sqrt(1 - alpha**2)
rng = np.random.default_rng(0)
sequences = np.zeros((nseq, nstep))
sequences[:, 0] = rng.normal(0, std, nseq)
for i in range(1, nstep):
    sequences[:, i] = alpha * sequences[:, i - 1] + rng.normal(0, beta, nseq)
```

5.1.4 Analysis With STACIE, Using the ExpPoly Model

The following code cell estimates the autocorrelation integral using the `ExpPolyModel`. Because the autocorrelation decays exponentially, the spectrum features a Lorentzian peak at zero frequency. Hence, we use degrees $S = \{0, 2\}$ for the polynomial, which ensures a zero-derivative at the origin.

```
spectrum = compute_spectrum(sequences)
result_exppoly = estimate_acint(spectrum, ExpPolyModel([0, 2]), verbose=True)
```

```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion        fcut
---------  ----------  ----------
     10.0        14.4    2.83e-04
     10.6        14.4    3.01e-04
     11.3        14.4    3.20e-04
     12.0        14.4    3.41e-04
     12.7        14.3    3.63e-04
     13.5        14.2    3.86e-04
     14.3        14.0    4.11e-04
     15.2        13.7    4.38e-04
     16.2        13.5    4.66e-04
     17.2        13.3    4.96e-04
     18.2        13.1    5.28e-04
     19.4        13.1    5.62e-04
     20.6        13.1    5.98e-04
     21.9        13.3    6.37e-04
     23.3        13.4    6.78e-04
     24.8        13.5    7.21e-04
     26.3        13.6    7.68e-04
     28.0        13.5    8.18e-04
     29.8        13.1    8.70e-04
     31.6        12.5    9.26e-04
     33.6        11.7    9.86e-04
     35.8        11.0    1.05e-03
     38.1        10.5    1.12e-03
     40.5        10.1    1.19e-03
     43.1        10.0    1.27e-03
     45.8        10.0    1.35e-03
     48.7        10.0    1.43e-03
     51.8        10.0    1.53e-03
     55.2         9.7    1.63e-03
     58.7         9.3    1.73e-03
```

(continues on next page)

(continued from previous page)

```
     62.4         8.8    1.84e-03
     66.4         8.3    1.96e-03
     70.7         7.7    2.09e-03
     75.2         7.3    2.22e-03
     80.0         7.0    2.37e-03
     85.1         6.9    2.52e-03
     90.6         7.1    2.68e-03
     96.4         7.3    2.85e-03
    102.6         7.6    3.04e-03
    109.2         7.7    3.23e-03
    116.2         7.7    3.44e-03
    123.7         7.5    3.66e-03
    131.6         7.2    3.90e-03
    140.1         7.0    4.15e-03
    149.1         6.9    4.42e-03
    158.6         6.9    4.70e-03
    168.8         7.1    5.01e-03
    179.7         7.4    5.33e-03
    191.2         8.2    5.67e-03
    203.6         9.7    6.04e-03
    216.6        12.9    6.43e-03
    230.6        19.2    6.84e-03
    245.4        30.7    7.29e-03
    261.2        50.9    7.76e-03
    278.0        84.7    8.26e-03
    295.9       140.3    8.79e-03
Cutoff criterion exceeds incumbent + margin: 6.9 + 100.0.

INPUT TIME SERIES
    Time step:                      1.00e+00
    Simulation time:                3.28e+04
    Maximum degrees of freedom:     128.0

MAIN RESULTS
    Autocorrelation integral:       9.99e-01 ± 1.76e-02
    Integrated correlation time:    1.59e+01 ± 2.80e-01

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:     117.3 (ideally > 40)
    Regression cost Z-score:        -0.1 (ideally < 2)
    Cutoff criterion Z-score:       0.3 (ideally < 2)

MODEL exppoly(0, 2) | CUTOFF CRITERION cv2l(125%)
    Number of parameters:           2
    Average cutoff frequency:       3.47e-03
```

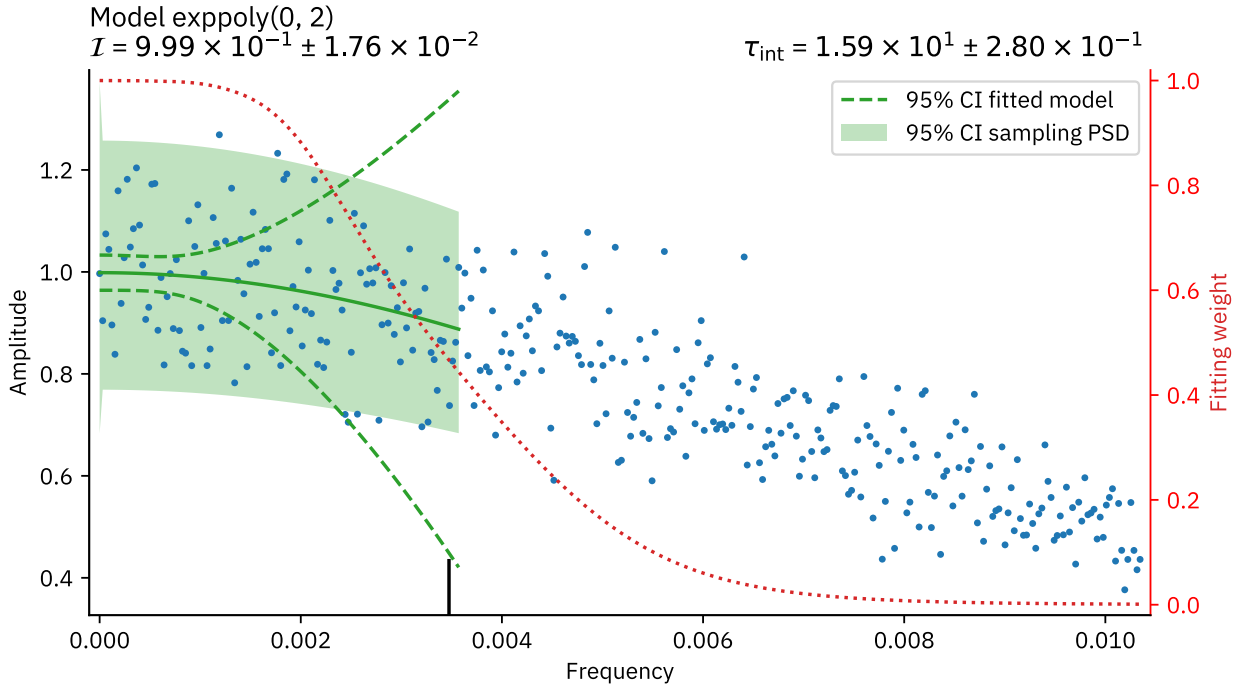
With the `verbose=True` option, STACIE prints the results of the analysis. The first section of the screen output shows the progress of the cutoff frequency scan and includes the following columns:

- `neff`: the number of effective spectrum data points used in the fit.
- `criterion`: the value of the cutoff criterion used for the weighted average over solutions at different cutoff frequencies.
- `fcut`: the cutoff frequency used for the fit.

The second section summarizes the analysis, and can be easily related to the concepts in the *theory section*. These results already reveal that STACIE reproduces the expected results.

The next code cell plots the model fitted to the spectrum.

```
uc = UnitConfig()
plt.close("fitted_exppoly")
fig, ax = plt.subplots(num="fitted_exppoly")
plot_fitted_spectrum(ax, uc, result_exppoly)
```

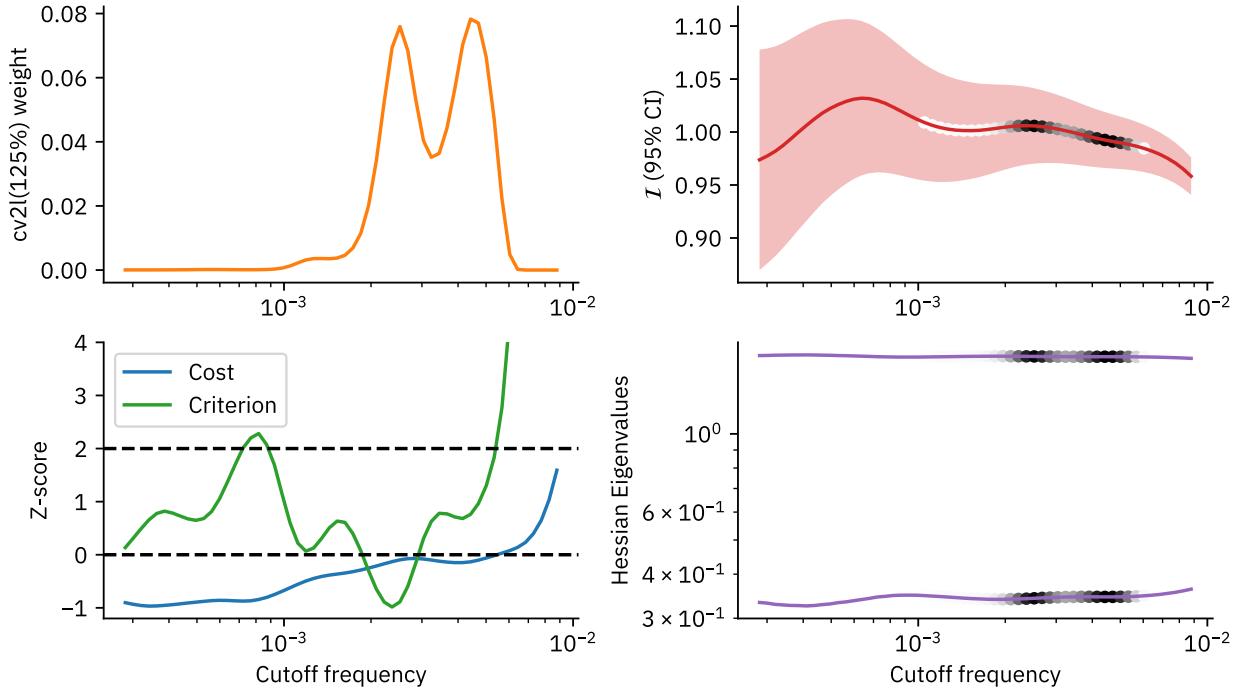


This plot displays a lot of information:

- The spectrum is shown as **blue dots**.
- The fitted model is plotted as a **solid green line**. Its parameters are the weighted average over multiple cutoff frequencies.
- The **red dotted line** shows the weighted average of the switching functions, used to identify the low-frequency region of the spectrum.
- The **green band** shows the expected *uncertainty* of the sampling spectrum, as a 95% confidence interval. Most of the blue data points should fall within this band, at least in the region where the model is fitted to the data (red dotted line close to 1.0).
- The **green dashed lines** are the 95% confidence intervals of the fitted model. This confidence interval should be narrow in the low-frequency region.
- The **black vertical line** corresponds to the weighted average of the cutoff frequencies used in the fit.
- The **plot title** summarizes key information about the analysis, including:
  - The model used to fit the data.
  - The autocorrelation integral and its uncertainty.
  - The integrated correlation time and its uncertainty.
- The **legend** shows to the confidence level used for plotting.

The next plot shows some intermediate results, which can help you understand the fitting process or detect problems.

```
plt.close("extras_exppoly")
fig, axs = plt.subplots(2, 2, num="extras_exppoly")
plot_extras(axs, uc, result_exppoly)
```



This plot contains four panels with extra results:

1. The **top left panel** shows the weight assigned to each cutoff frequency, based on the CV2L criterion. Things to look for:
   - If the cutoff weight is large for the lowest cutoffs, then the input sequences are likely too short. This typically also results in a low number of effective data points. Increasing the number of steps in the inputs will increase the frequency resolution of the spectrum, allowing for better fits with lower cutoff frequencies.
   - If the cutoff weight is large for the highest cutoffs, there is most likely also a problem with the analysis. There can be multiple causes for this:
     - The input sequences are much longer than necessary. In this case, you can increase the `neff_max` option of *`estimate_acint()`* to fit the model with higher cutoffs. However, this can be expensive, so it is recommended to use more and shorter sequences instead. This can be done by preprocessing the data with the *`split()`* function before computing the spectrum. Even better is to plan ahead and avoid this situation.
     - The data is block-averaged with a block size that is too large, which limits the available frequency range.
2. The **top right panel** shows the autocorrelation integral for each cutoff frequency. The dots indicate the extent to which each point contributes to the final result. (Black is high weight, white is low weight.) Things to look for:
   - If the autocorrelation integral shows sharp jumps at low cutoff frequencies, the model is most likely overfitting the data at low frequencies. These points are practically always given low cutoff weights, so you can ignore them. However, if you want to exclude them from the analysis, you can increase the `neff_min` option of *`estimate_acint()`*.

3. The **bottom left panel** the Z-scores of the regression cost and the cutoff criterion, as a function of the cutoff frequency. The Z-score is the number of standard deviations a value deviates from its mean. For ill-behaved fits, the Z-scores easily exceed 2. When providing sufficient inputs, high Z-scores should only occur where the cutoff weight is low. If the Z-scores are high for cutoff frequencies with high cutoff weights, the input data is insufficient for reliable error estimation or the model is not appropriate. In this case, it is recommended to use a different model or to increase the length of the input sequences.
4. The **bottom right panel** shows the eigenvalues of the Hessian matrix of the fit, in a preconditioned parameter space, at each cutoff frequency. A large spread of the eigenvalues indicates that the fit is not well constrained. Such a large spread typically results in overfitting artifacts.

5.1.5 Analysis With STACIE, Using the Lorentz Model

In this example, we know *a priori* that the autocorrelation function decays exponentially. Therefore, the `LorentzModel` should be able to perfectly explain the spectrum, up to the statistical noise in the data.

```
# Analysis
result_lorentz = estimate_acint(spectrum, LorentzModel(), verbose=True)

# Plotting
plt.close("fitted_lorentz")
fig, ax = plt.subplots(num="fitted_lorentz")
plot_fitted_spectrum(ax, uc, result_lorentz)
plt.close("extras_lorentz")
fig, axs = plt.subplots(2, 2, num="extras_lorentz")
plot_extras(axs, uc, result_lorentz)
```

```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion        fcut
---------  ----------  ----------
     15.0         inf    4.31e-04  (No correlation time estimate available.)
     15.9         inf    4.59e-04  (No correlation time estimate available.)
     16.9         inf    4.89e-04  (No correlation time estimate available.)
     18.0         inf    5.20e-04  (No correlation time estimate available.)
     19.1         inf    5.54e-04  (No correlation time estimate available.)
     20.3         inf    5.89e-04  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     21.6         inf    6.27e-04  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     23.0         inf    6.68e-04  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     24.4       112.7    7.11e-04
     25.9        81.9    7.57e-04
     27.6        68.9    8.06e-04
     29.3        62.7    8.58e-04
     31.2        59.2    9.13e-04
     33.2        57.5    9.72e-04
     35.3        56.8    1.03e-03
     37.5        56.7    1.10e-03
     39.9        57.1    1.17e-03
     42.4        57.8    1.25e-03
     45.1        58.5    1.33e-03
     48.0        59.3    1.41e-03
     51.1        60.1    1.51e-03
     54.4         inf    1.60e-03  (opt: Hessian matrix has non-positive eigenvalues:␣
```

(continues on next page)

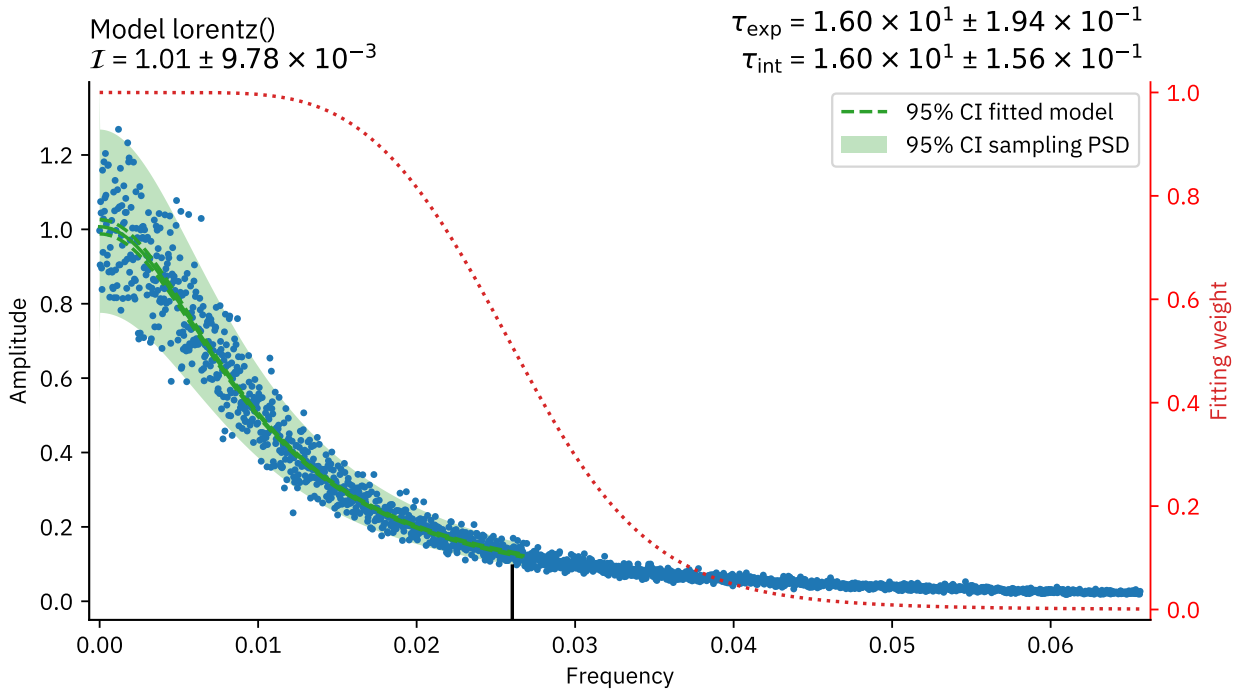
(continued from previous page)

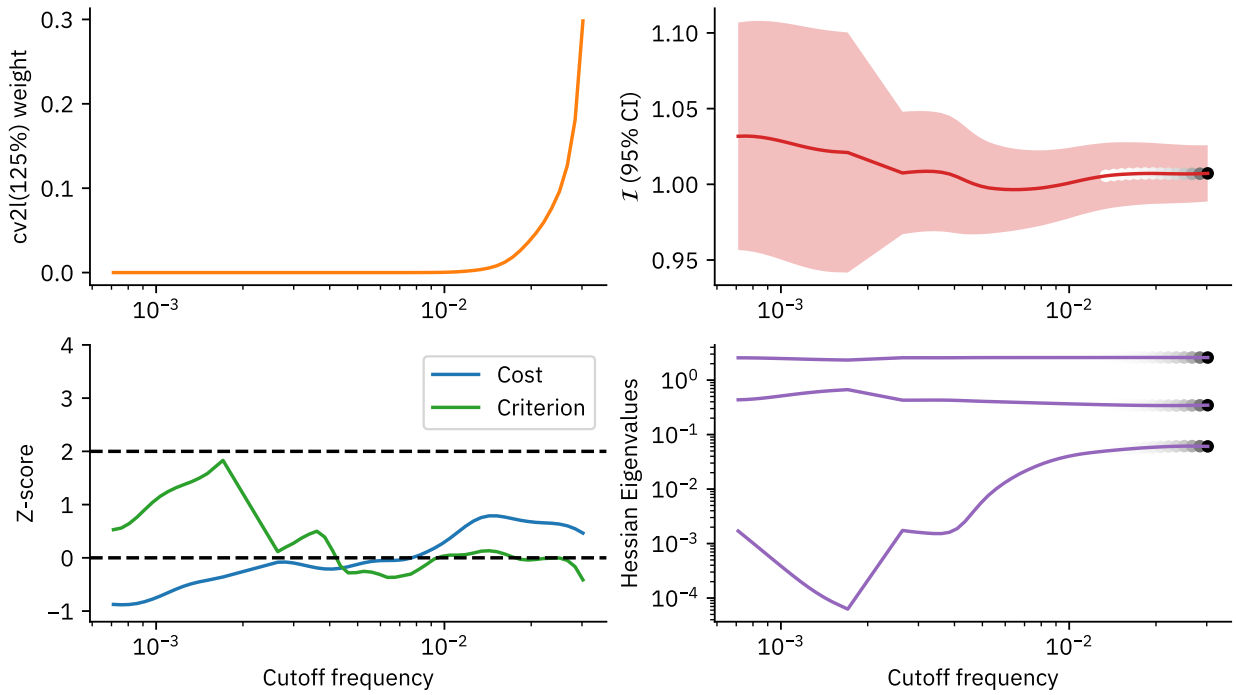
```
↪evals=array([-9.28856795e-05,  4.37431407e-01,  2.56266148e+00]))
    57.8        61.7    1.71e-03
    61.5         inf    1.82e-03  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
    65.5         inf    1.93e-03  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
    69.7         inf    2.06e-03  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
    74.1         inf    2.19e-03  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
    78.9         inf    2.33e-03  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
    83.9         inf    2.48e-03  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
    89.3       101.0    2.64e-03
    95.0        80.5    2.81e-03
   101.1        68.7    2.99e-03
   107.6        61.1    3.19e-03
   114.5        56.0    3.39e-03
   121.9        52.5    3.61e-03
   129.7        50.0    3.84e-03
   138.0        48.0    4.09e-03
   146.9        45.6    4.36e-03
   156.3        42.3    4.64e-03
   166.4        38.6    4.94e-03
   177.1        34.7    5.25e-03
   188.5        30.9    5.59e-03
   200.6        27.5    5.95e-03
   213.5        24.7    6.34e-03
   227.2        22.3    6.75e-03
   241.9        20.4    7.18e-03
   257.4        18.9    7.64e-03
   274.0        17.6    8.14e-03
   291.6        16.6    8.66e-03
   310.4        15.8    9.22e-03
   330.4        15.0    9.81e-03
   351.7        14.4    1.04e-02
   374.3        13.8    1.11e-02
   398.4        13.2    1.18e-02
   424.1        12.8    1.26e-02
   451.4        12.4    1.34e-02
   480.5        12.0    1.43e-02
   511.5        11.6    1.52e-02
   544.4        11.2    1.62e-02
   579.5        10.8    1.72e-02
   616.9        10.4    1.83e-02
   656.6        10.1    1.95e-02
   698.9         9.9    2.08e-02
   744.0         9.6    2.21e-02
   791.9         9.4    2.35e-02
   843.0         9.1    2.51e-02
   897.3         8.9    2.67e-02
   955.1         8.5    2.84e-02
  1016.7         8.0    3.02e-02
Reached the maximum number of effective points (1000).

INPUT TIME SERIES
    Time step:                     1.00e+00
    Simulation time:               3.28e+04
    Maximum degrees of freedom:    128.0
```

(continues on next page)

(continued from previous page)

```
MAIN RESULTS
    Autocorrelation integral:      1.01e+00 ± 9.78e-03
    Integrated correlation time:   1.60e+01 ± 1.56e-01

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:    875.6 (ideally > 60)
    Regression cost Z-score:       0.6 (ideally < 2)
    Cutoff criterion Z-score:      -0.2 (ideally < 2)

MODEL lorentz() | CUTOFF CRITERION cv2l(125%)
    Number of parameters:          3
    Average cutoff frequency:      2.60e-02
    Exponential correlation time:  1.60e+01 ± 1.94e-01

RECOMMENDED SIMULATION SETTINGS (EXPONENTIAL CORR. TIME)
    Block time:                    < 5.02e+00 ± 1.09e-01
    Simulation time:               > 1.00e+03 ± 1.54e+00
```





The extra plots reveal some noteworthy features:

- At the lowest cutoff frequencies, the model clearly overfits the data. There is a large spread on the eigenvalues, and the autocorrelation integral fluctuates significantly for low cutoff frequencies. Although the results at these cutoff frequencies are unreliable, they are given low weights, so you don't need to intervene manually to exclude these results.
- The cutoff weight is maximal for the highest cutoff frequencies. This is typically a sign that the input sequences are too long, but this is not the case here. While the Lorentz model would also yield excellent results with shorter sequences, this would still result in a high frequency cutoff. This is because the Lorentz model fits the data perfectly; more data points will always result in a lower cutoff criterion. However, in more realistic cases involving data from complex simulations or measurements, this is unlikely to happen.

5.1.6 Regression Tests

If you are experimenting with this notebook, you can ignore any exceptions below. The tests are only meant to pass for the notebook in its original form.

```
if abs(result_exppoly.acint - 1.0) > 0.03:
    raise ValueError(f"Wrong acint: {result_exppoly.acint:.4e}")
if abs(result_exppoly.corrtime_int - 16.0) > 0.5:
    raise ValueError(f"Wrong corrtime_int: {result_exppoly.corrtime_int:.4e}")
if abs(result_lorentz.acint - 1.0) > 0.03:
    raise ValueError(f"Wrong acint: {result_lorentz.acint:.4e}")
if abs(result_lorentz.corrtime_int - 16.0) > 0.5:
    raise ValueError(f"Wrong corrtime_int: {result_lorentz.corrtime_int:.4e}")
if abs(result_lorentz.corrtime_exp - 16.0) > 0.5:
    raise ValueError(f"Wrong corrtime_exp: {result_lorentz.corrtime_exp:.4e}")
```

5.1.7 Derivation of the Autocorrelation Integral

The Markov process is defined by the following equation:

$$\hat{x}_{n+1} = \alpha \hat{x}_n + \beta \hat{z}_n$$

The stationary distribution of the process is Gaussian with zero mean. The initial state is slowly reduced by repetitive application of the factor α. The only part that remains after a long time is the additive contributions from the normal noise $\hat{z}_n$.

Since the two terms on the right-hand side of the equation are independent, the variance of the stationary distribution can be found by solving:

$$c_0 = \alpha^2 c_0 + \beta^2$$

This gives:

$$c_0 = \frac{\beta^2}{1 - \alpha^2}$$

The covariance of two neighboring points is given by:

$$\mathrm{COV}[\hat{x}_n, \hat{x}_{n+1}] = \alpha\, \mathrm{COV}[\hat{x}_n, \hat{x}_n] = \alpha\, c_0$$

This can easily be generalized to points separated by $\Delta > 0$ steps through induction:

$$\mathrm{COV}[\hat{x}_n, \hat{x}_{n+\Delta}] = \alpha\ \mathrm{COV}[\hat{x}_n, \hat{x}_{n+\Delta-1}] = \alpha^\Delta\, c_0$$

with a similar result for $\Delta < 0$. Combining these results gives:

$$c_\Delta = \frac{\beta^2}{1 - \alpha^2}\, \alpha^{|\Delta|}$$

The autocorrelation integral is defined by a simple quadrature rule (with $F = 1$ and $h = 1$):

$$\mathcal{I} = \frac{1}{2} \sum_{\Delta=-\infty}^{\infty} \frac{\beta^2}{1 - \alpha^2}\, \alpha^{|\Delta|}$$

This can be rewritten easily using properties of geometric series. One must be careful not to double-count the $\Delta = 0$ term. This can be accomplished by isolating this term and rewriting the remaining terms in the sum with a shifted index, $n = |\Delta| - 1$. After replacing the index, we use $\sum_{\Delta=1}^{\infty} \alpha^\Delta = \alpha \sum_{n=0}^{\infty} \alpha^n$.

$$\begin{aligned}
\mathcal{I} &= \frac{\beta^2}{(1 - \alpha^2)} \left(\frac{1}{2} + \alpha \sum_{n=0}^{\infty} \alpha^n \right) \\
&= \frac{\beta^2}{(1 - \alpha^2)} \left(\frac{1}{2} + \frac{\alpha}{1 - \alpha} \right) \\
&= \frac{\beta^2}{(1 - \alpha^2)} \left(\frac{1 + \alpha}{2(1 - \alpha)} \right) \\
&= \frac{1}{2} \left(\frac{\beta}{1 - \alpha} \right)^2
\end{aligned}$$

5.2 Uncertainty of the Mean of Time-Correlated Data

This notebook shows how to use STACIE to compute the error of the mean of a time-correlated input sequence, meaning not all of its values are statistically independent.

This is a completely self-contained example that generates input sequences (with MCMC) and then analyzes them with STACIE. Atomic units are used unless otherwise noted.

We suggest experimenting with this notebook by making the following changes:

- Change the number of sequences and their length.
- Change the correlation time through `PROPOSAL_STEP`.

5.2.1 Library Imports and Matplotlib Configuration

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from scipy.integrate import quad
import scipy.constants as sc
from stacie import (
    UnitConfig,
    compute_spectrum,
    estimate_acint,
    LorentzModel,
    plot_extras,
    plot_fitted_spectrum,
    plot_spectrum,
)
```

```
mpl.rc_file("matplotlibrc")
%config InlineBackend.figure_formats = ["svg"]
```

5.2.2 Data Generation

The data for the analysis are generated by sampling a Kratzer–Feus potential of a diatomic molecule [Fue26, Kra20] at constant temperature. This potential is harmonic in $1/r$:

$$U(r) = \frac{K}{2}\left(\frac{r_0^2}{r} - r_0\right)^2$$

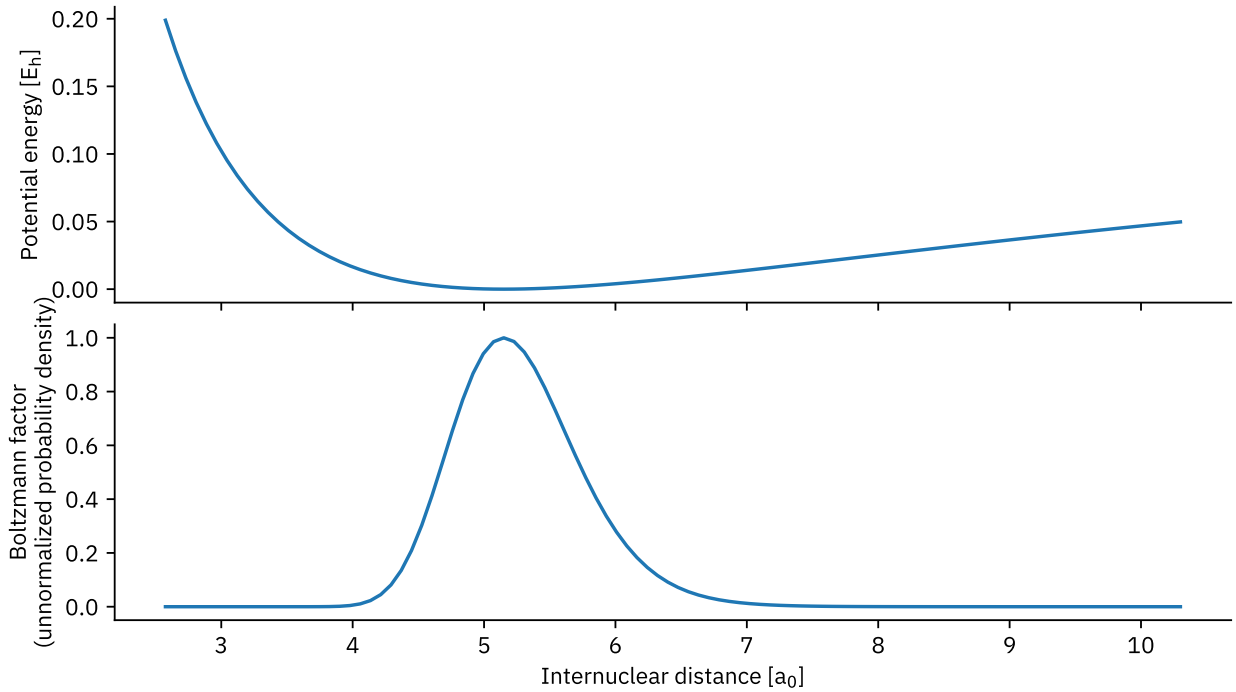
where K is the force constant and r_0 the equilibrium bond length. The sampled probability density is the Boltzmann distribution:

$$p_r(r) = \frac{1}{Z}\exp\left(-\frac{U(r)}{k_\mathrm{B}T}\right)$$

where the normalization Z is the classical partition function.

In this example, the force constant and bond length of the lithium dimer are used, with parameters from Zhao *et al.* [ZZF22] converted to atomic units. A high temperature is used to skew the distribution to larger distances.

```
K = 0.015
R0 = 5.150
TEMPERATURE = 1000
BOLTZMANN = sc.value("Boltzmann constant") / sc.value("Hartree energy")
BETA = 1 / (BOLTZMANN * TEMPERATURE)
PROPOSAL_STEP = 0.1


def logprob(r):
    """Calculate the logarithm of the probability."""
    energy = 0.5 * K * (R0**2 / r - R0) ** 2
    return -BETA * energy


def plot_potential_dist():
    plt.close("boltzmann")
    _, (ax1, ax2) = plt.subplots(2, 1, num="boltzmann", sharex=True)
    rgrid = np.linspace(0.5 * R0, 2 * R0, 100)
    ax2.sharex(ax1)
    ax1.plot(rgrid, -logprob(rgrid) / BETA)
    ax1.set_ylabel(r"Potential energy [E$_\text{h}$]")
    ax2.plot(rgrid, np.exp(logprob(rgrid)))
    ax2.set_ylabel("Boltzmann factor\n(unnormalized probability density)")
    ax2.set_xlabel("Internuclear distance [a$_0$]")


plot_potential_dist()
```



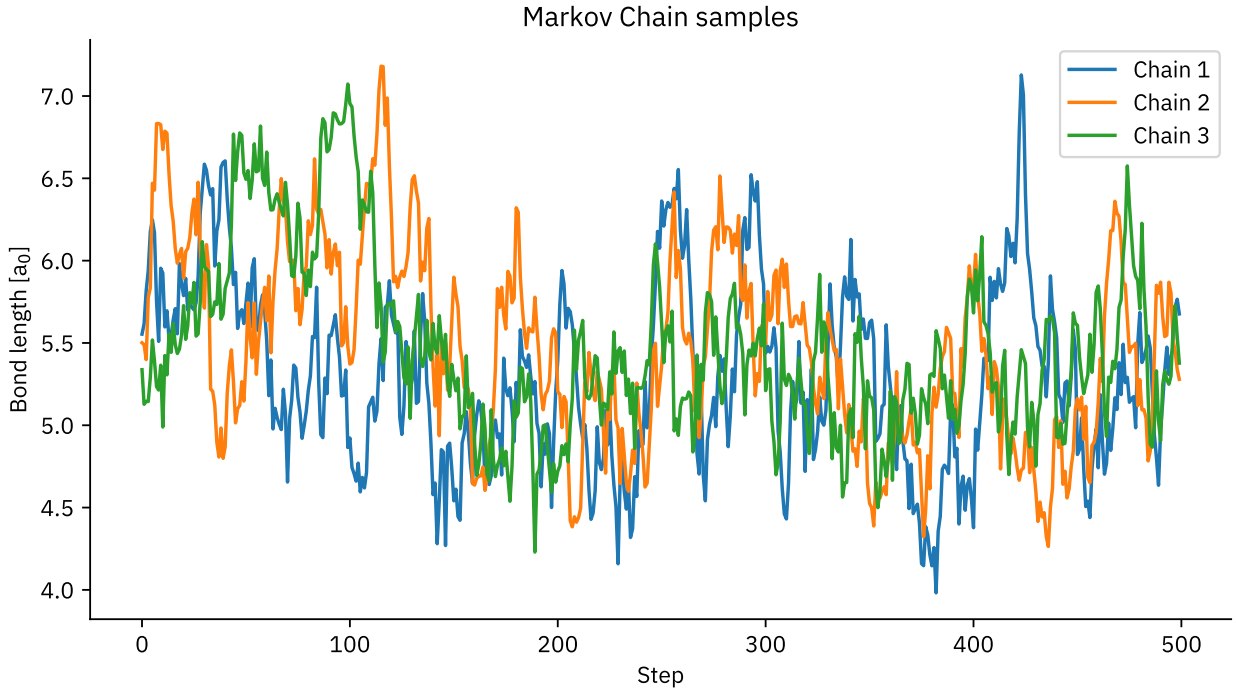
The MCMC implementation below is non-standard in the sense that it is vectorized to generate multiple sequences in parallel.

```
def sample_mcmc_chain(niter, stride, ndim, burnin, seed=42):
    """Sample independent Markov Chains with the Metropolis algorithm.

    Parameters
    ----------
    niter
        The number of MCMC iterations to run.
    stride
        The number of iterations between samples returned,
        i.e. the thinning interval.
    ndim
        The number of independent Markov chains to run.
    burnin
        The number of iterations to discard at the beginning.
    seed
        The random number generator seed.

    Returns
    -------
    result
        A 2D array of shape (ndim, niter // stride) containing the sampled sequences.
    """
    rng = np.random.default_rng(seed)
    result = np.zeros((ndim, niter // stride))
    r_old = np.full(ndim, R0)
    lp_old = logprob(r_old)
    irow = 0
    istep = 0
    while irow < result.shape[1]:
        r_new = r_old + rng.normal(0, PROPOSAL_STEP, ndim)
        lp_new = logprob(r_new)
        accept = lp_new > lp_old
        mask = ~accept
        nrnd = mask.sum()
        if nrnd > 0:
            accept[mask] = rng.uniform(0, 1, nrnd) < np.exp(lp_new[mask] - lp_old[mask])
        r_old[accept] = r_new[accept]
        lp_old[accept] = lp_new[accept]
        if burnin > 0:
            burnin -= 1
            continue
        if istep % stride == 0:
            result[:, irow] = r_new
            irow += 1
        istep += 1
    return result


sequences = sample_mcmc_chain(10240, 5, 50, 200)
print(f"(nseq, nstep) = {sequences.shape}")
mean_mc = sequences.mean()
print(f"Monte Carlo E[r] ≈ {mean_mc:.5f} > R0 = {R0:.5f}")
```

```
(nseq, nstep) = (50, 2048)
Monte Carlo E[r] ≈ 5.28666 > R0 = 5.15000
```

Because of the finite temperature and the anharmonicity of the potential, the average distance is greater than the equilibrium bond length.

```
# Plot the beginning of a few sequences.
# The atomic unit of length is the Bohr radius, $\mathrm{a}_0$.
def plot_chains():
    plt.close("chains")
    _, ax = plt.subplots(num="chains")
    ax.plot(sequences[0][:500], label="Chain 1")
    ax.plot(sequences[1][:500], label="Chain 2")
    ax.plot(sequences[2][:500], label="Chain 3")
    ax.set_xlabel("Step")
    ax.set_ylabel(r"Bond length [a$_0$]")
    ax.set_title("Markov Chain samples")
    ax.legend()


plot_chains()
```



The sequences in the plot are clearly time-correlated. The following cells show how STACIE can be used to compute the *uncertainty* of this average, taking into account that not all samples are independent due to time correlations.

5.2.3 Uncertainty Quantification

The spectrum is calculated using settings that are appropriate for error estimation. See the *Error Estimates* section for the justification of the `prefactors` and `include_zero_freq` keyword arguments. Since we are analyzing MCMC data, the `timestep` argument is not specified, corresponding to a dimensionless time step of 1.

```
# Compute and plot the power spectrum.
spectrum = compute_spectrum(
```

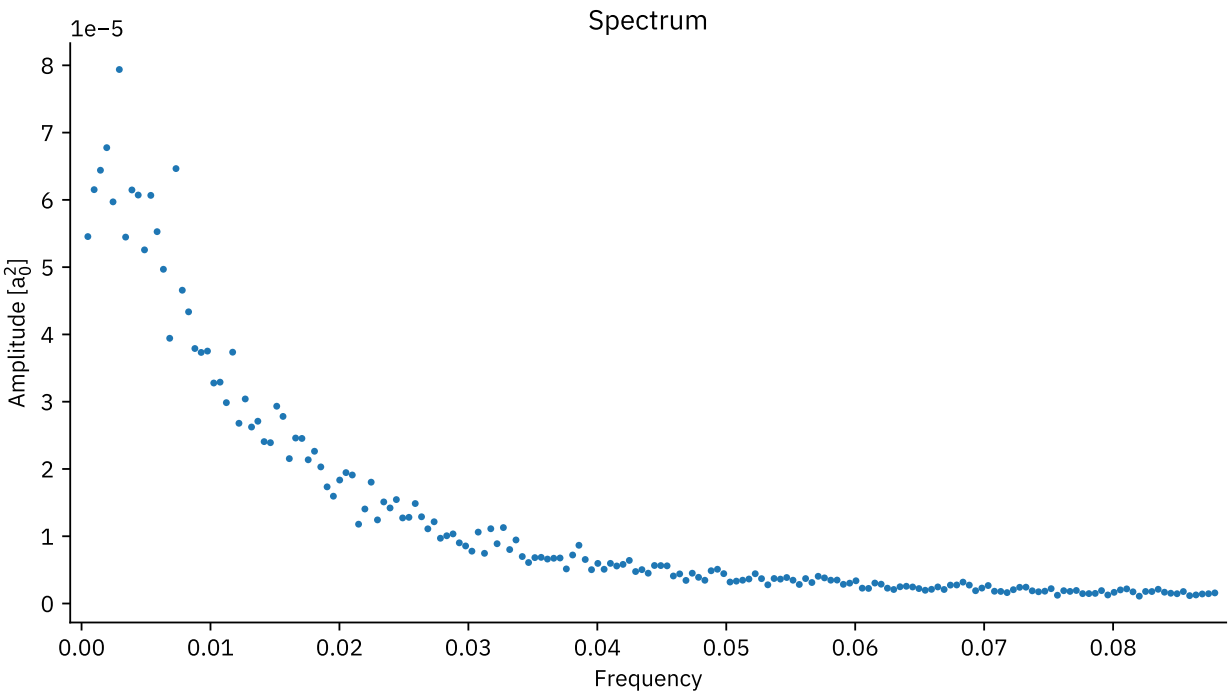
(continues on next page)

(continued from previous page)

```
    sequences,
    prefactors=2.0 / sequences.size,
    include_zero_freq=False,
)
```

The `UnitConfig` object contains settings that are reused by most plotting functions. The integral has units of length squared, a_0^2. (It is the variance of the mean.)

```
uc = UnitConfig(
    time_fmt=".1f",
    acint_fmt=".1e",
    acint_unit_str=r"a$^2_0$",
)
plt.close("spectrum")
_, ax = plt.subplots(num="spectrum")
plot_spectrum(ax, uc, spectrum, 180)
```



From the spectrum, one can already visually estimate the variance of the mean: the limit to zero frequency is about $6 \times 10^{-5}\, a_0^2$. By normalizing the spectrum with the total simulation time, the spectrum has the correct unit of length squared. In the following cell, a model is fitted to the spectrum to get a more precise estimate.

```
result = estimate_acint(spectrum, LorentzModel(), verbose=True, uc=uc)
```

```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion        fcut
---------  ----------  ----------
     15.0         7.0    7.38e-03
     16.0         4.5    7.85e-03
     17.1         2.9    8.36e-03
```

(continues on next page)

(continued from previous page)

```
     18.2         1.5    8.90e-03
     19.4         0.5    9.47e-03
     20.7        -0.3    1.01e-02
     22.0        -0.9    1.07e-02
     23.5        -1.2    1.14e-02
     25.0        -1.3    1.22e-02
     26.7        -1.3    1.29e-02
     28.4        -1.2    1.38e-02
     30.3        -1.3    1.47e-02
     32.3        -1.5    1.56e-02
     34.4        -1.9    1.66e-02
     36.7        -2.3    1.77e-02
     39.1        -2.7    1.88e-02
     41.6        -3.2    2.00e-02
     44.3        -3.7    2.13e-02
     47.2        -4.3    2.27e-02
     50.3        -5.0    2.42e-02
     53.6        -5.6    2.57e-02
     57.1        -6.1    2.74e-02
     60.8        -6.4    2.92e-02
     64.7        -6.7    3.11e-02
     68.9        -6.9    3.31e-02
     73.4        -7.0    3.52e-02
     78.2        -7.2    3.75e-02
     83.3        -7.2    3.99e-02
     88.7        -7.3    4.24e-02
     94.4        -7.4    4.52e-02
    100.5        -7.4    4.81e-02
    107.1        -7.6    5.12e-02
    114.0        -7.8    5.45e-02
    121.4        -7.9    5.80e-02
    129.2        -8.0    6.18e-02
    137.6        -8.1    6.57e-02
    146.5        -8.1    7.00e-02
    156.0        -8.0    7.45e-02
    166.1        -7.8    7.93e-02
    176.8        -7.6    8.44e-02
    188.3        -7.3    8.98e-02
    200.4        -7.2    9.56e-02
    213.4        -7.2    1.02e-01
    227.2        -7.4    1.08e-01
    241.9        -7.6    1.15e-01
    257.5        -7.8    1.23e-01
    274.2        -7.9    1.31e-01
    291.9        -7.9    1.39e-01
    310.7        -7.8    1.48e-01
    330.8        -7.7    1.58e-01
    352.2        -7.6    1.68e-01
    374.9         inf    1.79e-01  (cv2l: Insufficient data after cutoff.)
Scan stopped by cutoff criterion.

INPUT TIME SERIES
    Time step:                    1.0
    Simulation time:              2048.0
```

(continues on next page)

(continued from previous page)

```
    Maximum degrees of freedom:    100.0

MAIN RESULTS
    Autocorrelation integral:      6.3e-05 ± 2.5e-06 a$^2_0$
    Integrated correlation time:   12.2 ± 0.5

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:    178.1 (ideally > 60)
    Regression cost Z-score:       -0.0 (ideally < 2)
    Cutoff criterion Z-score:      1.1 (ideally < 2)

MODEL lorentz() | CUTOFF CRITERION cv2l(125%)
    Number of parameters:          3
    Average cutoff frequency:      8.50e-02
    Exponential correlation time:  12.5 ± 0.5

RECOMMENDED SIMULATION SETTINGS (EXPONENTIAL CORR. TIME)
    Block time:                    < 3.9 ± 0.3
    Simulation time:               > 786.2 ± 4.2
```

The spectrum is normalized such that the integral of the autocorrelation function is equal to the variance of the mean. Because STACIE estimates errors of the autocorrelation integral, it can thus also estimate errors of errors of means.

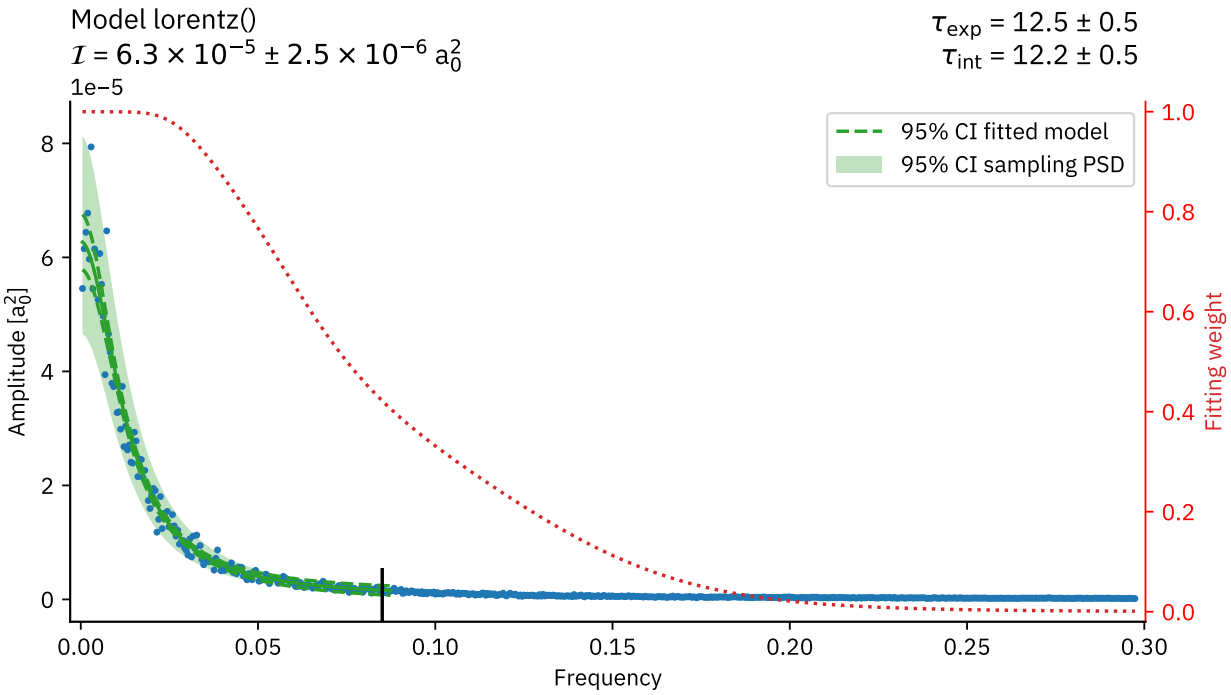
The error of the mean and its uncertainty are printed in the following cell.

```
error_mc = np.sqrt(result.acint)
print(f"Error of the mean = {error_mc:.5f}")
error_of_error_mc = 0.5 * result.acint_std / error_mc
print(f"Uncertainty of the error of the mean = {error_of_error_mc:.5f}")
```

```
Error of the mean = 0.00792
Uncertainty of the error of the mean = 0.00016
```

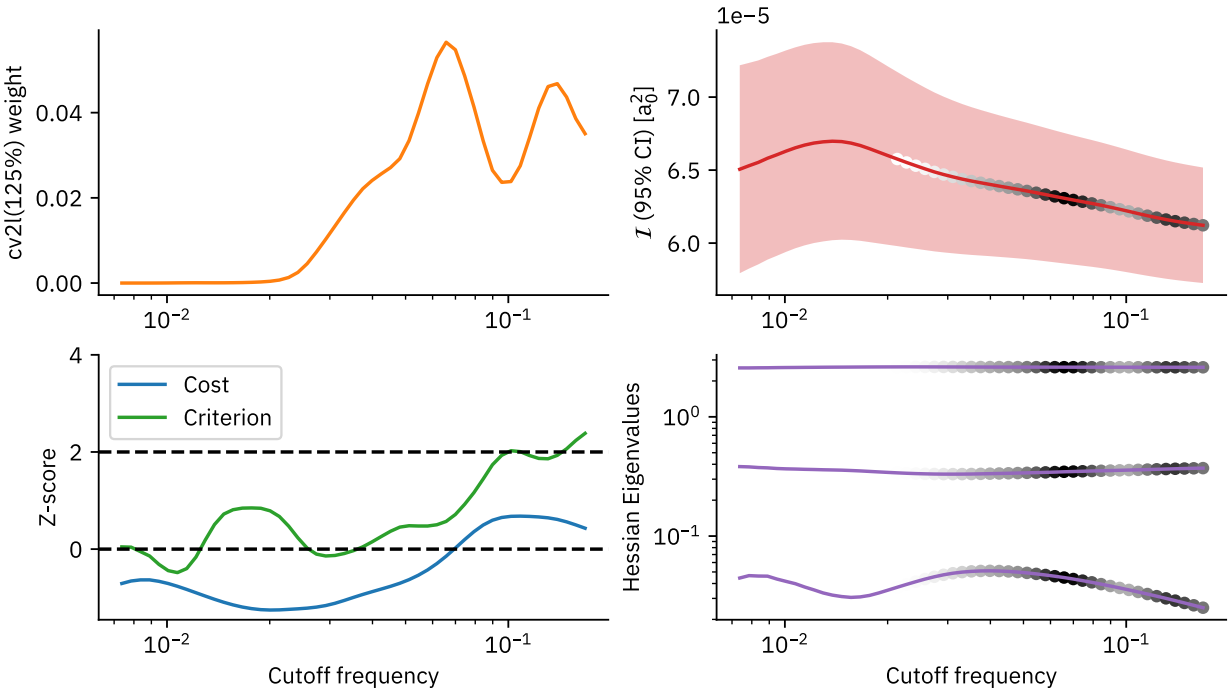
It is also interesting to visualize the fitted spectrum and some intermediate results.

```
# Plot of the sampling and fitted model spectrum.
plt.close("fitted")
_, ax = plt.subplots(num="fitted")
plot_fitted_spectrum(ax, uc, result)
```



The Lorentz model can clearly explain the spectrum, even well beyond the width of the peak at zero frequency.

```
# Plot additional intermediate results as a function of the frequency cutoff.
plt.close("extras")
_, axs = plt.subplots(2, 2, num="extras")
plot_extras(axs, uc, result)
```



The extra plots reveal several interesting challenges of the analysis:

- The cutoff weight (top left panel) remains high up to the highest cutoff frequency considered.

If one is only interested in the zero-frequency limit of the spectrum, there is little to be gained by including many data points in the fit at high frequencies, well past the width of the peak at zero frequency. These will not make the autocorrelation integral more precise, but bear the risk of introducing some bias due to underfitting. One may manually impose a maximum frequency cutoff with the `fcut_max` argument of the `estimate_acint()` function.

- The risk for some bias at high cutoff frequencies is also visible in the Z-score associated with the cutoff criterion (green curve in the lower left panel). For higher cutoff frequencies, the Z-score slowly increases to values above 2, where the cutoff weight is still significant.

  The reason for the higher Z-score is that the input time series is not normally distributed, due to the asymmetry of the Kratzer–Fues potential. As a result, the MC chain cannot be described by a Gaussian process, and the uncertainty of the spectrum amplitudes is not exactly Gamma-distributed. You can verify this hypothesis by rerunning this example with `TEMPERATURE = 100` and `PROPOSAL_STEP = 0.03`. This will result in a more symmetric distribution of bond lengths. By lowering the proposal step, the correlation time remains about the same. With these settings, a lower criterion Z-score is obtained at high cutoff frequencies.

5.2.4 Precise Mean With Numerical Quadrature

Because the probability density sampled by the MC chain is one-dimensional, it is feasible to compute the mean using numerical quadrature, which is much more accurate than the Monte Carlo estimate. (For production simulations, Monte Carlo is only advantageous for high-dimensional problems.)

As shown in the code below, the difference between the quadrature and Monte Carlo estimates is on the order of the estimated uncertainty of the MC result.

```
numer_quad = quad(lambda r: r * np.exp(logprob(r)), 0, 50)[0]
denom_quad = quad(lambda r: np.exp(logprob(r)), 0, 50)[0]
mean_quad = numer_quad / denom_quad
print(f"Quadrature  E[r]   ≈ {mean_quad:8.5f}")
print(f"Monte Carlo E[r]   ≈ {mean_mc:8.5f}")
print(f"|Difference|       = {abs(mean_quad - mean_mc):8.5f}")
print(f"Estimated MC error = {error_mc:8.5f}")
```

```
Quadrature  E[r]   ≈  5.28043
Monte Carlo E[r]   ≈  5.28666
|Difference|       =  0.00623
Estimated MC error =  0.00792
```

5.2.5 Autocorrelation time

The Lorentz model estimates the *exponential correlation time* [Sok97] from the width of the peak at zero frequency in the spectrum. It may differ from the *integrated autocorrelation time*. Only if the autocorrelation function is nothing but an exponentially decaying function, both should match.

```
print("Autocorrelation times:")
print(f"exponential: {result.corrtime_exp:5.2f} ± {result.corrtime_exp_std:5.2f}")
print(f"integrated:  {result.corrtime_int:5.2f} ± {result.corrtime_int_std:5.2f}")
```

```
Autocorrelation times:
exponential: 12.51 ±  0.52
integrated:  12.24 ±  0.48
```

Here, the deviation between the two autocorrelation times falls within the uncertainty of the estimates. This is the expected result, since the Lorentzian model is able to explain the whole spectrum.

5.2.6 Regression Tests

If you are experimenting with this notebook, you can ignore any exceptions below. The tests are only meant to pass for the notebook in its original form.

```
if abs(mean_mc - 5.28666) > 1e-3:
    raise ValueError(f"Wrong mean_mc: {mean_mc:.5f}")
if abs(error_mc - 0.00794) > 1e-3:
    raise ValueError(f"Wrong error_mc: {error_mc:.5f}")
```

5.3 Applicability of the Lorentz Model

STACIE's *Lorentz model* assumes that the autocorrelation function decays exponentially for large lag times. Not all dynamical systems exhibit this exponential relaxation. If you want to apply STACIE to systems without exponential relaxation, you can use the *exppoly model* instead.

To illustrate the applicability of the Lorentz model, this notebook applies STACIE to numerical solutions of Thomas' Cyclically Symmetric Attractor:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sin(y) - bx$$
$$\frac{\mathrm{d}y}{\mathrm{d}t} = \sin(z) - by$$
$$\frac{\mathrm{d}z}{\mathrm{d}t} = \sin(x) - bz$$

For $b < 0.208186$, this system has chaotic solutions. As a result, the system looses memory of its initial conditions rather quickly, and the autocorrelation function tends to decay exponentially. At the boundary, $b = 0.208186$, the exponential decay is no longer valid and the spectrum deviates from the Lorentzian shape. In practice, the Lorentz model is applicable for smaller values, $0 < b < 0.17$.

For $b = 0$, the solutions become random walks with anomalous diffusion [RS08]. In this case, it makes more sense to work with the spectrum of the time derivative of the solutions. However, due to the anomalous diffusion, the spectrum of these derivatives cannot be approximated well with the Lorentz model.

This example is fully self-contained: input data is generated with numerical integration and then analyzed with STACIE. Dimensionless units are used throughout.

We suggest you experiment with this notebook by changing the b parameter and replacing the Lorentz model with the ExpPoly model.

5.3.1 Library Imports and Matplotlib Configuration

```
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from numpy.typing import ArrayLike, NDArray
from stacie import (
```

(continues on next page)

(continued from previous page)

```
    UnitConfig,
    compute_spectrum,
    estimate_acint,
    LorentzModel,
    plot_extras,
    plot_fitted_spectrum,
    plot_spectrum,
)
```

```
mpl.rc_file("matplotlibrc")
%config InlineBackend.figure_formats = ["svg"]
```

5.3.2 Data Generation

The following cell implements the numerical integration of the oscillator using Ralston's method for 100 different initial configurations. The parameter b is given as an argument to the `generate()` function at the last line of the next cell.

```
NSYS = 100
NDIM = 3
NSTEP = 20000
TIMESTEP = 0.3


def time_derivatives(state: ArrayLike, b: float) -> NDArray:
    """Compute the time derivatives defining the differential equations."""
    return np.sin(np.roll(state, 1, axis=1)) - b * state


def integrate(state: ArrayLike, nstep: int, h: float, b: float) -> NDArray:
    """Integrate the System with Ralston's method, using a fixed time step h.

    Parameters
    ----------
    state
        The initial state of the system, shape `(ndim, nsys)`,
        where `ndim` is the number of dimensions and `nsys` systems to integrate in parallel.
    nstep
        The number of time steps to integrate.
    h
        The time step size.
    b
        The parameter $b$ in the differential equations.

    Returns
    -------
    trajectory
        The trajectory of the system, shape `(nstep, ndim, nsys)`.
        The first dimension is the time step, the second dimension is the state variable,
        and the third dimension is the system index.
    """
    trajectory = np.zeros((nstep, *state.shape))
    for istep in range(nstep):
```

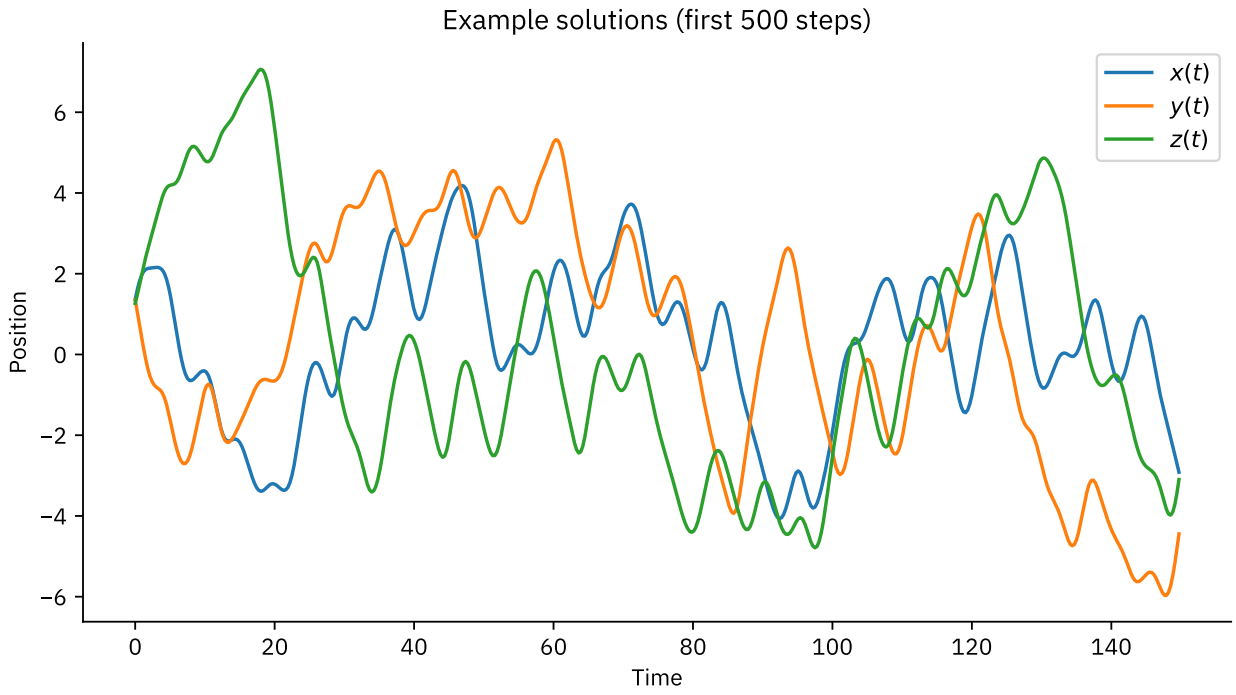
(continues on next page)

(continued from previous page)

```
        k1 = time_derivatives(state, b)
        k2 = time_derivatives(state + (2 * h / 3) * k1, b)
        state += h * (k1 + 3 * k2) / 4
        trajectory[istep] = state
    return trajectory


def generate(b: float):
    """Generate solutions for random initial states."""
    rng = np.random.default_rng(42)
    x = rng.uniform(-2, 2, (NDIM, NSYS))
    return integrate(x, NSTEP, TIMESTEP, b)


trajectory = generate(b=0.1)
```

The solutions shown below are smooth, but for low enough values of b, they are pseudo-random over longer time scales.

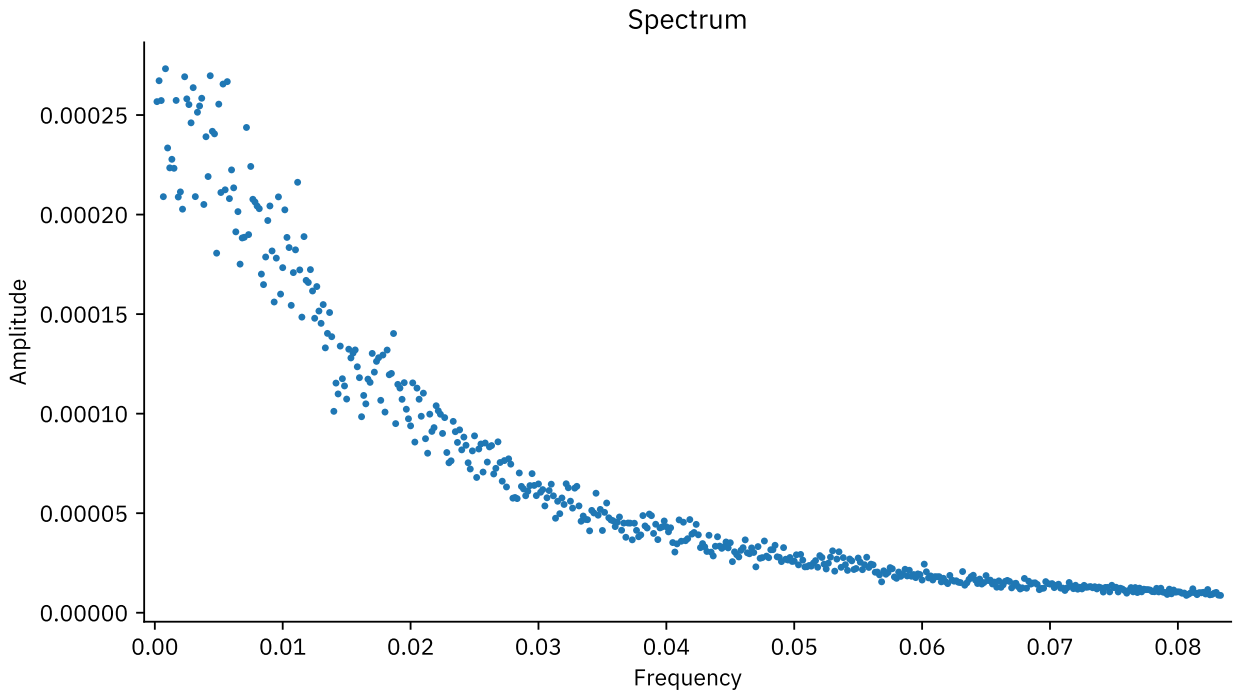
```
def plot_traj(nplot=500):
    """Show the first 500 steps of the first 10 solutions."""
    plt.close("traj")
    _, ax = plt.subplots(num="traj")
    times = np.arange(nplot) * TIMESTEP
    ax.plot(times, trajectory[:nplot, 0, 0], label="$x(t)$")
    ax.plot(times, trajectory[:nplot, 1, 0], label="$y(t)$")
    ax.plot(times, trajectory[:nplot, 2, 0], label="$z(t)$")
    ax.set_xlabel("Time")
    ax.set_ylabel("Position")
    ax.set_title(f"Example solutions (first {nplot} steps)")
    ax.legend()


plot_traj()
```



5.3.3 Spectrum

In the chaotic regime, the low-frequency spectrum indicates diffusive motion: a large peak at the origin. The spectrum is normalized so that the autocorrelation integral becomes the *variance of the mean*.

```
uc = UnitConfig(acint_fmt=".2e")
sequences = trajectory[:, 0, :].T  # use x(t) only
spectrum = compute_spectrum(
    sequences,
    timestep=TIMESTEP,
    prefactors=2.0 / (NSTEP * TIMESTEP * NSYS),
    include_zero_freq=False,
)
plt.close("spectrum")
_, ax = plt.subplots(num="spectrum")
plot_spectrum(ax, uc, spectrum, nplot=500)
```



Note that we only use component 0, i.e. $x(t)$, of each system as input for the spectra. This ensures that fully independent sequences are used in the analysis below, which is assumed by the statistical model of the spectrum used by STACIE.

5.3.4 Error of the Mean

The following cells fit the Lorentz model to the spectrum to derive the variance of the mean.

```
result = estimate_acint(spectrum, LorentzModel(), verbose=True)
```

```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion        fcut
---------  ----------  ----------
     15.0        26.7    2.52e-03
     16.0        27.3    2.68e-03
     17.1        28.1    2.85e-03
     18.2        29.0    3.04e-03
     19.4         inf    3.23e-03  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-2.13212330e-04,  4.27724835e-01,  2.57248838e+00]))
     20.7         inf    3.44e-03  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-3.42858708e-04,  4.26401259e-01,  2.57394160e+00]))
     22.0         inf    3.66e-03  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-6.28787951e-04,  4.25285484e-01,  2.57534330e+00]))
     23.5         inf    3.90e-03  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-7.74018437e-04,  4.23479196e-01,  2.57729482e+00]))
     25.0         inf    4.15e-03  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-9.36617560e-04,  4.22060298e-01,  2.57887632e+00]))
     26.7         inf    4.42e-03  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-6.89294195e-04,  4.19968631e-01,  2.58072066e+00]))
     28.4         inf    4.70e-03  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-8.22895236e-05,  4.18498803e-01,  2.58158349e+00]))
     30.3         inf    5.01e-03  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
```

(continues on next page)

(continued from previous page)

```
     32.3         inf    5.33e-03  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     34.4         inf    5.67e-03  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     36.7         inf    6.04e-03  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     39.1        80.8    6.43e-03
     41.6        43.3    6.84e-03
     44.3        29.2    7.28e-03
     47.2        20.9    7.75e-03
     50.3        15.8    8.25e-03
     53.6        12.2    8.79e-03
     57.1         9.5    9.35e-03
     60.8         7.6    9.96e-03
     64.7         6.3    1.06e-02
     68.9         5.2    1.13e-02
     73.4         3.7    1.20e-02
     78.2         1.6    1.28e-02
     83.3        -0.4    1.36e-02
     88.7        -1.9    1.45e-02
     94.4        -2.1    1.54e-02
    100.5        -0.9    1.64e-02
    107.1         1.5    1.75e-02
    114.0         3.8    1.86e-02
    121.4         4.8    1.98e-02
    129.2         4.3    2.11e-02
    137.6         3.1    2.24e-02
    146.5         2.3    2.39e-02
    156.0         1.8    2.54e-02
    166.1         1.0    2.71e-02
    176.8        -0.2    2.88e-02
    188.3        -1.1    3.07e-02
    200.4        -1.0    3.26e-02
    213.4         0.5    3.48e-02
    227.2         3.1    3.70e-02
    241.9         6.2    3.94e-02
    257.5         8.9    4.19e-02
    274.2        10.5    4.46e-02
    291.9        10.5    4.75e-02
    310.7         9.7    5.06e-02
    330.8         8.8    5.38e-02
    352.2         7.9    5.73e-02
    374.9         6.5    6.10e-02
    399.1         3.7    6.49e-02
    424.9         inf    6.91e-02  (cv2l: Linear dependencies in basis. evals=array([1.
↪83362182e-07, 2.82393010e-01, 2.71760681e+00]))
    452.3         inf    7.36e-02  (cv2l: Linear dependencies in basis. evals=array([9.
↪49923181e-07, 2.36034488e-01, 2.76396456e+00]))
    481.5         inf    7.83e-02  (cv2l: Linear dependencies in basis. evals=array([1.
↪47629017e-07, 1.81704351e-01, 2.81829550e+00]))
    512.6       140.1    8.34e-02
Cutoff criterion exceeds incumbent + margin: -2.1 + 100.0.

INPUT TIME SERIES
    Time step:                     3.00e-01
    Simulation time:               6.00e+03
    Maximum degrees of freedom:    200.0
```

(continues on next page)

(continued from previous page)

```
MAIN RESULTS
    Autocorrelation integral:      2.46e-04 ± 5.09e-06
    Integrated correlation time:   1.02e+01 ± 2.10e-01

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:    121.8 (ideally > 60)
    Regression cost Z-score:       1.3 (ideally < 2)
    Cutoff criterion Z-score:      1.2 (ideally < 2)

MODEL lorentz() | CUTOFF CRITERION cv2l(125%)
    Number of parameters:          3
    Average cutoff frequency:      1.99e-02
    Exponential correlation time:  1.02e+01 ± 8.84e-01

RECOMMENDED SIMULATION SETTINGS (EXPONENTIAL CORR. TIME)
    Block time:                    < 3.19e+00 ± 4.95e-01
    Simulation time:               > 6.39e+02 ± 7.01e+00
```

Due to the symmetry of the oscillator, the mean of the solutions should be zero. Within the *uncertainty*, this is indeed the case for the numerical solutions, as shown below.

```
mean = sequences.mean()
print(f"Mean: {mean:.3e}")
error_mean = np.sqrt(result.acint)
print(f"Error of the mean: {error_mean:.3e}")
```

```
Mean: 1.303e-02
Error of the mean: 1.569e-02
```

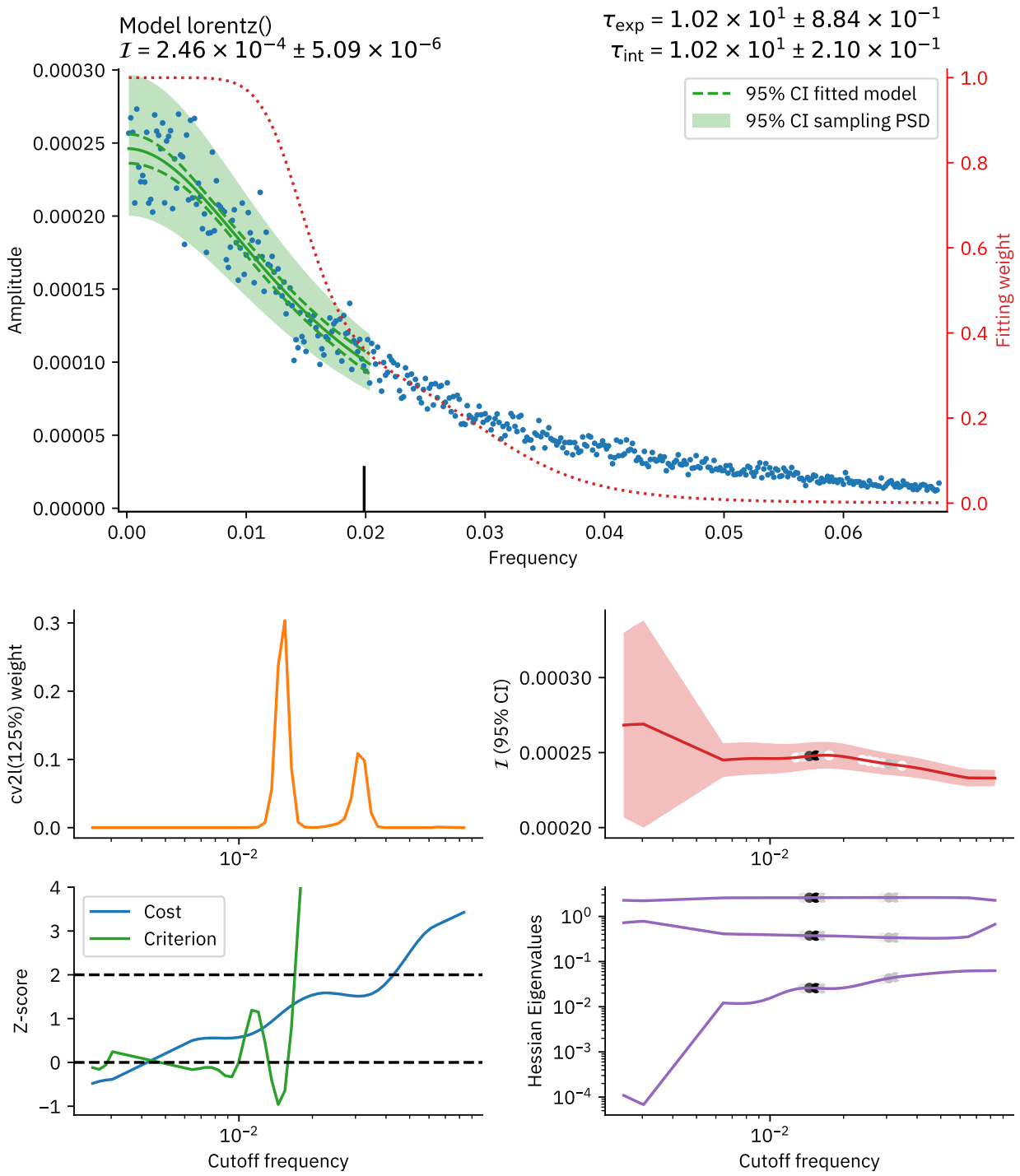
For sufficiently small values of b, the autocorrelation function decays exponentially, so that the two *autocorrelation times* are very similar:

```
print(f"corrtime_exp = {result.corrtime_exp:.3f} ± {result.corrtime_exp_std:.3f}")
print(f"corrtime_int = {result.corrtime_int:.3f} ± {result.corrtime_int_std:.3f}")
```

```
corrtime_exp = 10.166 ± 0.884
corrtime_int = 10.172 ± 0.210
```

To further gauge the applicability of the Lorentz model, it is useful to plot the fitted spectrum and the intermediate results as a function of the cutoff frequency, as shown below.

```
plt.close("fitted")
fig, ax = plt.subplots(num="fitted")
plot_fitted_spectrum(ax, uc, result)
plt.close("extras")
fig, axs = plt.subplots(2, 2, num="extras")
plot_extras(axs, uc, result)
```



It is clear that at higher cutoff frequencies, which are given a negligible weight, the spectrum deviates from the Lorentzian shape. Hence, at shorter time scales, the autocorrelation function does not decay exponentially. This was to be expected, as the input sequences are smooth functions. To further confirm this, we recommend rerunning this notebook with different values of b:

- For lower value, such as $b = 0.05$, the Lorentz model will fit the spectrum better, which is reflected in lower Z-score values.
- Up to $b = 0.17$, the Lorentz model is still applicable, but the Z-scores will increase.
- For $b = 0.2$, the Lorentz model will not be able to assign an exponential correlation time. To

be able to run the notebook until the last plot, you need to comment out the line that prints the exponential correlation time.

5.3.5 Regression Tests

If you are experimenting with this notebook, you can ignore any exceptions below. The tests are only meant to pass for the notebook in its original form.

```
if abs(result.acint - 2.47e-4) > 2e-5:
    raise ValueError(f"Wrong acint: {result.acint:.4e}")
if abs(result.corrtime_exp - 10.166) > 1e-1:
    raise ValueError(f"Wrong corrtime_exp: {result.corrtime_exp:.4e}")
```

5.4 Diffusion on a Surface with Newtonian Dynamics

This example shows how to compute the diffusion coefficient of a particle adsorbed on a crystal surface. For simplicity, the motion of the adsorbed particle is described by Newton's equations (without thermostat), i.e. in the *NVE* ensemble, and the particle can only move in two dimensions.

This is a completely self-contained example that generates the input sequences (with numerical integration) and then analyzes them with STACIE. Unless otherwise noted, atomic units are used.

5.4.1 Library Imports and Matplotlib Configuration

```
import attrs
import matplotlib as mpl
import matplotlib.pyplot as plt
import numdifftools as nd
import numpy as np
import scipy.constants as sc
from numpy.typing import ArrayLike, NDArray
from stacie import (
    UnitConfig,
    compute_spectrum,
    estimate_acint,
    LorentzModel,
    plot_extras,
    plot_fitted_spectrum,
)
from utils import compute_msds
```

```
mpl.rc_file("matplotlibrc")
%config InlineBackend.figure_formats = ["svg"]
```

5.4.2 Data Generation

Potential energy surface

The first cell below defines the potential energy of a particle on a surface, as well as the force that the surface exerts on the particle. The potential energy model is a superposition of cosine

functions:

$$U(\mathbf{r}) = -A \sum_{n=1}^{N} \cos(2\pi \mathbf{r} \cdot \mathbf{e}_n / \lambda)$$

with

$$\mathbf{e}_n = \mathbf{e}_x \cos(n\alpha) + \mathbf{e}_y \sin(n\alpha)$$

The default settings for this notebook result in a hexagonal lattice: $A = 0.2\,\mathrm{eV}$, $\lambda = 5\,\mathrm{a}_0$, $N = 3$, and $\alpha = 2\pi/3$. One may change these parameters to construct different types of surfaces:

- A square lattice: $N = 2$ and $\alpha = \pi/2$.
- A quasi-periodic pentagonal lattice: $N = 5$ and $\alpha = 2\pi/5$.

The rest of the notebook is set up to work well with the default parameters. If you change the potential energy model, remaining settings will also need to be adapted.

```
WAVELENGTH = 5.0
ALPHA = 2 * np.pi / 3
ANGLES = np.arange(3) * ALPHA
EV = sc.value("electron volt") / sc.value("atomic unit of energy")
AMPLITUDE = 0.2 * EV
ANGSTROM = 1e-10 / sc.value("atomic unit of length")


def potential_energy_force(coords: ArrayLike) -> tuple[NDArray, NDArray]:
    """Compute the potential energies for given particle positions.

    Parameters
    ----------
    coords
        A NumPy array with one or more particle positions.
        The last dimension is assumed to have size two.
        Index 0 and 1 of the last axis correspond to x and y coordinates,
        respectively.

    Returns
    -------
    energy
        The potential energies for the given particle positions.
        An array with shape `pos.shape[:-1]`.
    force
        The forces acting on the particles.
        Same shape as `pos`, with same index conventions.
    """
    coords = np.asarray(coords, dtype=float)
    x = coords[..., 0]
    y = coords[..., 1]
    energy = 0
    force = np.zeros(coords.shape)
    wavenum = 2 * np.pi / WAVELENGTH
    for angle in ANGLES:
        arg = (x * np.cos(angle) + y * np.sin(angle)) * wavenum
        energy -= np.cos(arg)
```

(continues on next page)

(continued from previous page)

```
        sin_wave = np.sin(arg) * wavenum
        force[..., 0] -= sin_wave * np.cos(angle)
        force[..., 1] -= sin_wave * np.sin(angle)
    return AMPLITUDE * energy, AMPLITUDE * force
```

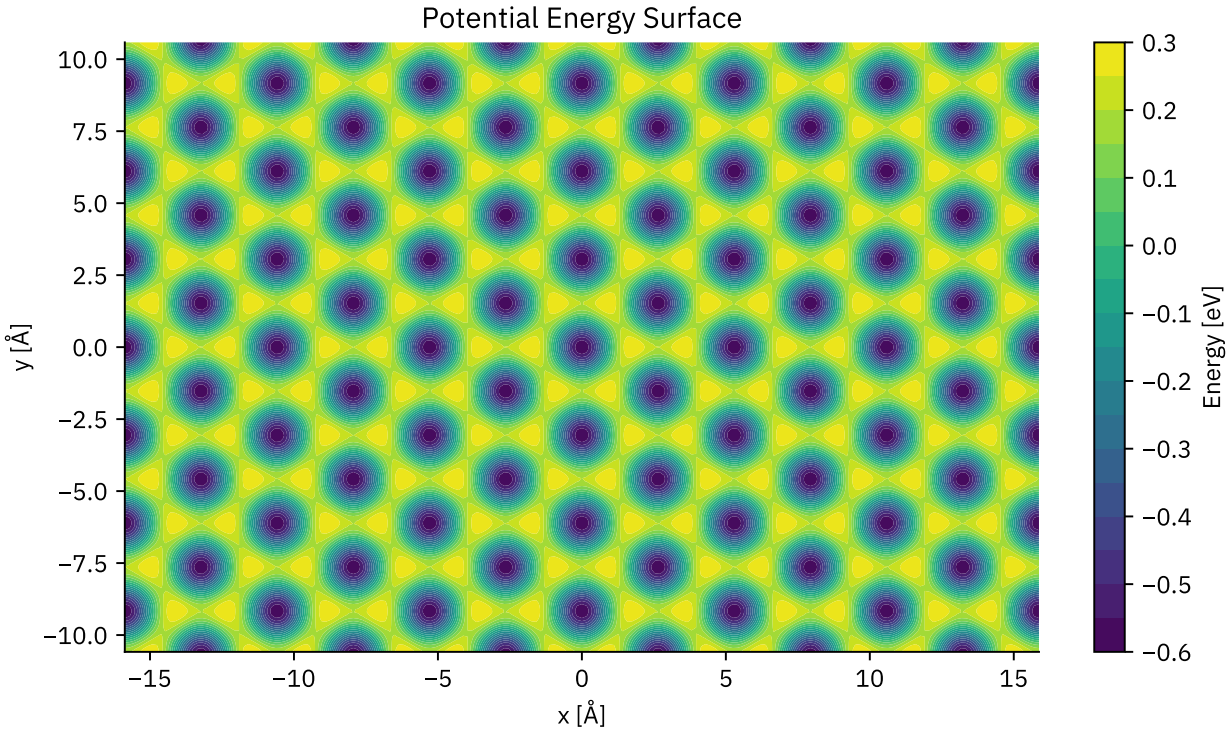
The following code cell provides a quick visual test of the forces using numdifftools. (The force is equal to minus the energy gradient.)

```
print(potential_energy_force([1, 2]))
print(nd.Gradient(lambda coords: potential_energy_force(coords)[0])([1, 2]))
```

```
(np.float64(0.0045001339646275546), array([-0.00569293, -0.01063935]))
[0.00569293 0.01063935]
```

Finally, the following code cell plots the potential energy surface.

```
def plot_pes():
    plt.close("pes")
    fig, ax = plt.subplots(num="pes")
    xs = np.linspace(-30, 30, 201)
    ys = np.linspace(-20, 20, 201)
    coords = np.array(np.meshgrid(xs, ys)).transpose(1, 2, 0)
    energies = potential_energy_force(coords)[0]
    cf = ax.contourf(xs / ANGSTROM, ys / ANGSTROM, energies / EV, levels=20)
    ax.set_aspect("equal", "box")
    ax.set_xlabel("x [Å]")
    ax.set_ylabel("y [Å]")
    ax.set_title("Potential Energy Surface")
    fig.colorbar(cf, ax=ax, label="Energy [eV]")


plot_pes()
```



Newtonian Dynamics

The following code cell implements a vectorized Velocity Verlet integrator, which can integrate multiple independent trajectories at the same time. Some parameters, like mass and time step are fixed as global constants. The mass is that of an Argon atom converted to atomic units. The timestep is five femtosecond converted to atomic units.

```
MASS = sc.value("unified atomic mass unit") * 39.948 / sc.value("atomic unit of mass")
FEMTOSECOND = 1e-15 / sc.value("atomic unit of time")
PICOSECOND = 1e-12 / sc.value("atomic unit of time")
TERAHERTZ = 1e12 * sc.value("atomic unit of time")
TIMESTEP = 5 * FEMTOSECOND


@attrs.define
class Trajectory:
    """Bundle dynamics trajectory results.

    The first axis of all array attributes corresponds to time steps.
    """

    timestep: float = attrs.field()
    """The spacing between two recorded time steps."""

    coords: NDArray = attrs.field()
    """The time-dependent particle positions, with shape `(natom, 2, nstep)`.

    The last index is used for time steps, of which only every `block_size` step is recorded.
    """
```

(continues on next page)

(continued from previous page)

```
    vels: NDArray = attrs.field()
    """The time-dependent particle velocities, with shape `(natom, 2, nstep)`.

    If block_size is larger than 1,
    this attribute contains the block-averaged velocity.
    """

    potential_energies: NDArray = attrs.field()
    """The time-dependent potential energies."""

    kinetic_energies: NDArray = attrs.field()
    """The time-dependent potential energies."""

    @classmethod
    def empty(cls, shape: tuple[int, ...], nstep: int, timestep: float):
        """Construct an empty trajectory object."""
        return cls(
            timestep,
            np.zeros((*shape, 2, nstep)),
            np.zeros((*shape, 2, nstep)),
            np.zeros((*shape, nstep)),
            np.zeros((*shape, nstep)),
        )

    @property
    def nstep(self) -> int:
        """The number of time steps."""
        return self.coords.shape[-1]


def integrate(coords: ArrayLike, vels: ArrayLike, nstep: int, block_size: int = 1):
    """Integrate Newton's equation of motion for the given initial conditions.

    Parameters
    ----------
    coords
        The initial particle positions.
        Index 0 and 1 of the last axis correspond to x and y coordinates.
    vels
        The initial particle velocities.
        Index 0 and 1 of the last axis correspond to x and y coordinates.
    nstep
        The number of MD time steps.
    block_size
        The block_size with which to record the trajectory data.

    Returns
    -------
    trajectory
        A Trajectory object holding all the results.
    """
    traj = Trajectory.empty(coords.shape[:-1], nstep // block_size, TIMESTEP * block_size)
    energies, forces = potential_energy_force(coords)
    delta_vels = forces * (0.5 * TIMESTEP / MASS)
```

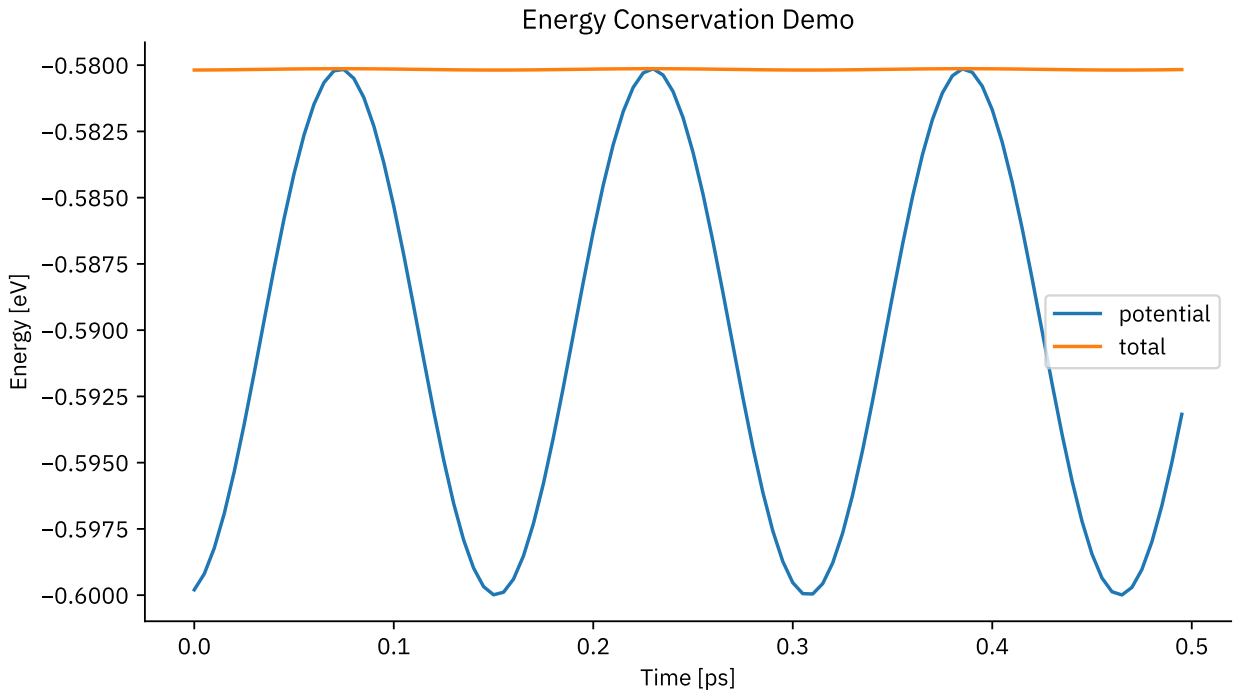
(continues on next page)

(continued from previous page)

```
    vels_block = 0
    for istep in range(traj.nstep * block_size):
        vels += delta_vels
        coords += vels * TIMESTEP
        energies, forces = potential_energy_force(coords)
        delta_vels = forces * (0.5 * TIMESTEP / MASS)
        vels += delta_vels
        vels_block += vels
        if istep % block_size == block_size - 1:
            itraj = istep // block_size
            traj.coords[..., itraj] = coords
            traj.vels[..., itraj] = vels_block / block_size
            traj.potential_energies[..., itraj] = energies
            traj.kinetic_energies[..., itraj] = (0.5 * MASS) * (vels**2).sum(axis=-1)
            vels_block = 0
    return traj
```

As a quick test, the following code cell integrates the equations of motion for a single particle with a small initial velocity. In this case, the particle oscillates around the origin and one can easily verify that the total energy is conserved.

```
def demo_energy_conservation():
    """Simple demo of the approximate energy conservation.

    The initial velocity is small enough
    to let the particle vibrate around the origin.
    """
    nstep = 100
    traj = integrate(np.zeros(2), np.full(2, 1e-4), nstep)
    plt.close("energy")
    _, ax = plt.subplots(num="energy")
    times = np.arange(traj.nstep) * traj.timestep
    ax.plot(times / PICOSECOND, traj.potential_energies / EV, label="potential")
    ax.plot(
        times / PICOSECOND,
        (traj.potential_energies + traj.kinetic_energies) / EV,
        label="total",
    )
    ax.set_title("Energy Conservation Demo")
    ax.set_xlabel("Time [ps]")
    ax.set_ylabel("Energy [eV]")
    ax.legend()


demo_energy_conservation()
```



Demonstration of Deterministic Choas

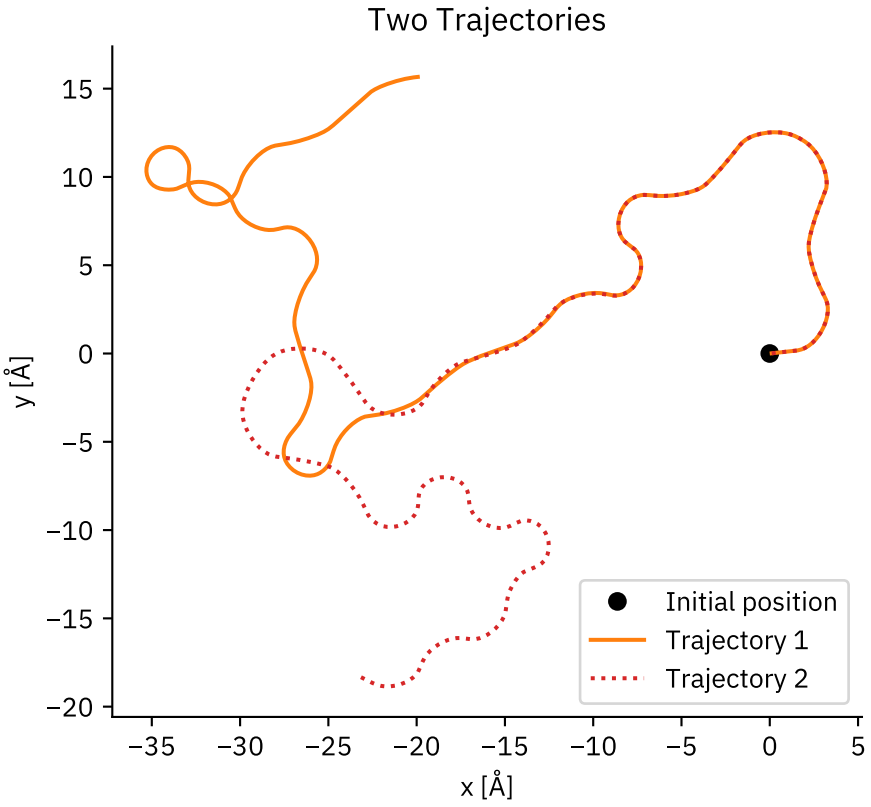
Newtonian dynamics is deterministic, but has chaotic solutions for many systems. The particle on a surface in this notebook is no exception. The following cell shows two trajectories for nearly identical initial conditions, but they slowly drift apart over time. After sufficient time, any information about their nearly identical initial conditions is lost.

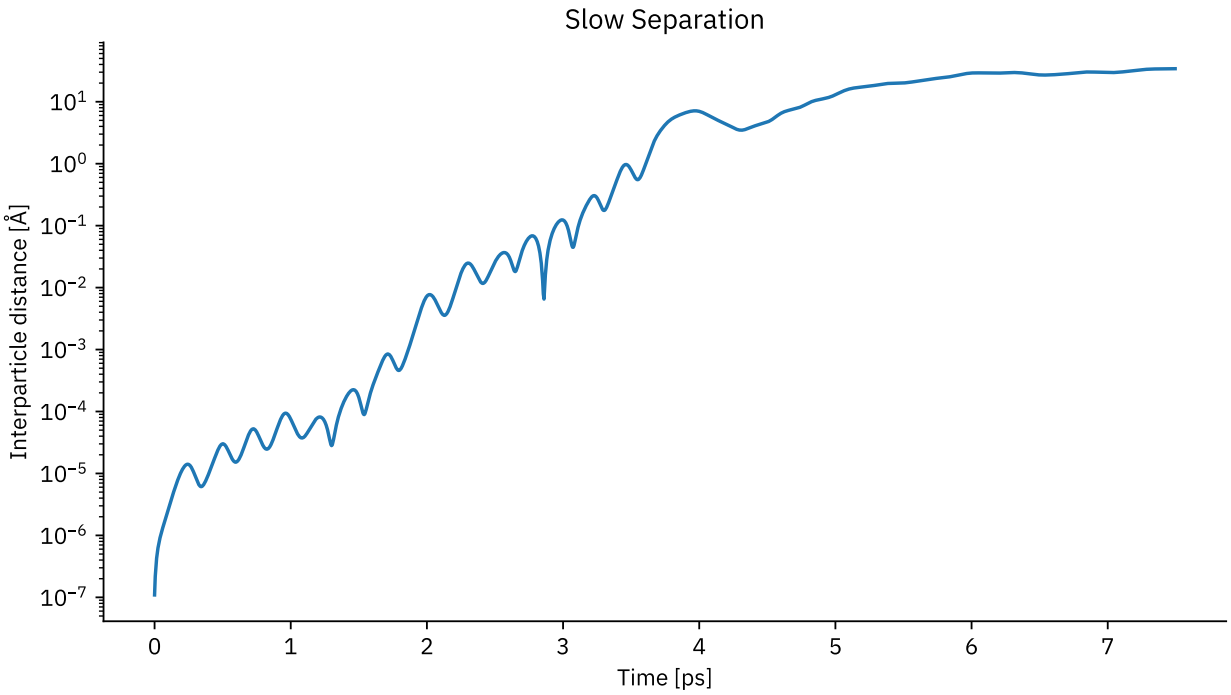
```
def demo_chaos():
    vels = np.array([[1e-3, 1e-4], [1.000001e-3, 1e-4]])
    traj = integrate(np.zeros((2, 2)), vels, 1500)
    plt.close("chaos")
    _, ax = plt.subplots(num="chaos")
    ax.plot([0], [0], "o", color="k", label="Initial position")
    ax.plot(
        traj.coords[0, 0] / ANGSTROM,
        traj.coords[0, 1] / ANGSTROM,
        color="C1",
        label="Trajectory 1",
    )
    ax.plot(
        traj.coords[1, 0] / ANGSTROM,
        traj.coords[1, 1] / ANGSTROM,
        color="C3",
        ls=":",
        label="Trajectory 2",
    )
    ax.set_aspect("equal", "box")
    ax.set_xlabel("x [Å]")
    ax.set_ylabel("y [Å]")
    ax.legend()
    ax.set_title("Two Trajectories")
```

(continues on next page)

(continued from previous page)

```
    plt.close("chaos_dist")
    _, ax = plt.subplots(num="chaos_dist")
    times = np.arange(traj.nstep) * traj.timestep
    ax.semilogy(
        times / PICOSECOND,
        np.linalg.norm(traj.coords[0] - traj.coords[1], axis=0) / ANGSTROM,
    )
    ax.set_xlabel("Time [ps]")
    ax.set_ylabel("Interparticle distance [Å]")
    ax.set_title("Slow Separation")


demo_chaos()
```





Because the trajectories are chaotic, the short term motion is ballistic, while the long term motion is a random walk.

Note that the random walk is only found in a specific energy window. If the energy is too small, the particles will oscillate around a local potential energy minimum. If the energy is too large, or just high enough to cross barriers, the particles will follow almost linear paths over the surface.

5.4.3 Surface diffusion without block averages

This section considers 100 independent particles whose initial velocities have the same magnitude but whose directions are random. The time-dependent particle velocities are used as inputs for STACIE to compute the diffusion coefficient.

```
def demo_stacie(block_size: int = 1):
    """Simulate particles on a surface and compute the diffusion coefficient.

    Parameters
    ----------
    block_size
        The block size for the block averages.
        If 1, no block averages are used.

    Returns
    -------
    result
        The result of the STACIE analysis.
    """
    natom = 100
    nstep = 20000
    rng = np.random.default_rng(42)
    vels = rng.normal(0, 1, (natom, 2))
    vels *= 9.7e-4 / np.linalg.norm(vels, axis=1).reshape(-1, 1)
```

(continues on next page)

(continued from previous page)

```
    traj = integrate(np.zeros((natom, 2)), vels, nstep, block_size)

    plt.close(f"trajs_{block_size}")
    _, ax = plt.subplots(num=f"trajs_{block_size}", figsize=(6, 6))
    for i in range(natom):
        ax.plot(traj.coords[i, 0], traj.coords[i, 1])
    ax.set_aspect("equal", "box")
    ax.set_xlabel("x [a$_0$]")
    ax.set_ylabel("y [a$_0$]")
    ax.set_title(f"{natom} Newtonian Pseudo-Random Walks")

    spectrum = compute_spectrum(
        traj.vels.reshape(2 * natom, traj.nstep),
        timestep=traj.timestep,
    )

    # Define units and conversion factors used for screen output and plotting.
    # This does not affect numerical values stored in the result object.
    uc = UnitConfig(
        acint_symbol="D",
        acint_unit=sc.value("atomic unit of time")
        / sc.value("atomic unit of length") ** 2,
        acint_unit_str="m$^2$/s",
        acint_fmt=".2e",
        freq_unit=TERAHERTZ,
        freq_unit_str="THz",
        time_unit=PICOSECOND,
        time_unit_str="ps",
        time_fmt=".2f",
    )

    # The maximum cutoff frequency is chosen to be 1 THz,
    # by inspecting the first spectrum plot.
    # Beyond the cutoff frequency, the spectrum has resonance peaks that
    # the Lorentz model is not designed to handle.
    result = estimate_acint(
        spectrum, LorentzModel(), fcut_max=TERAHERTZ, verbose=True, uc=uc
    )

    # Plotting
    plt.close(f"spectrum_{block_size}")
    _, ax = plt.subplots(num=f"spectrum_{block_size}")
    plot_fitted_spectrum(ax, uc, result)
    plt.close(f"extras_{block_size}")
    _, axs = plt.subplots(2, 2, num=f"extras_{block_size}")
    plot_extras(axs, uc, result)
    return traj, result
```

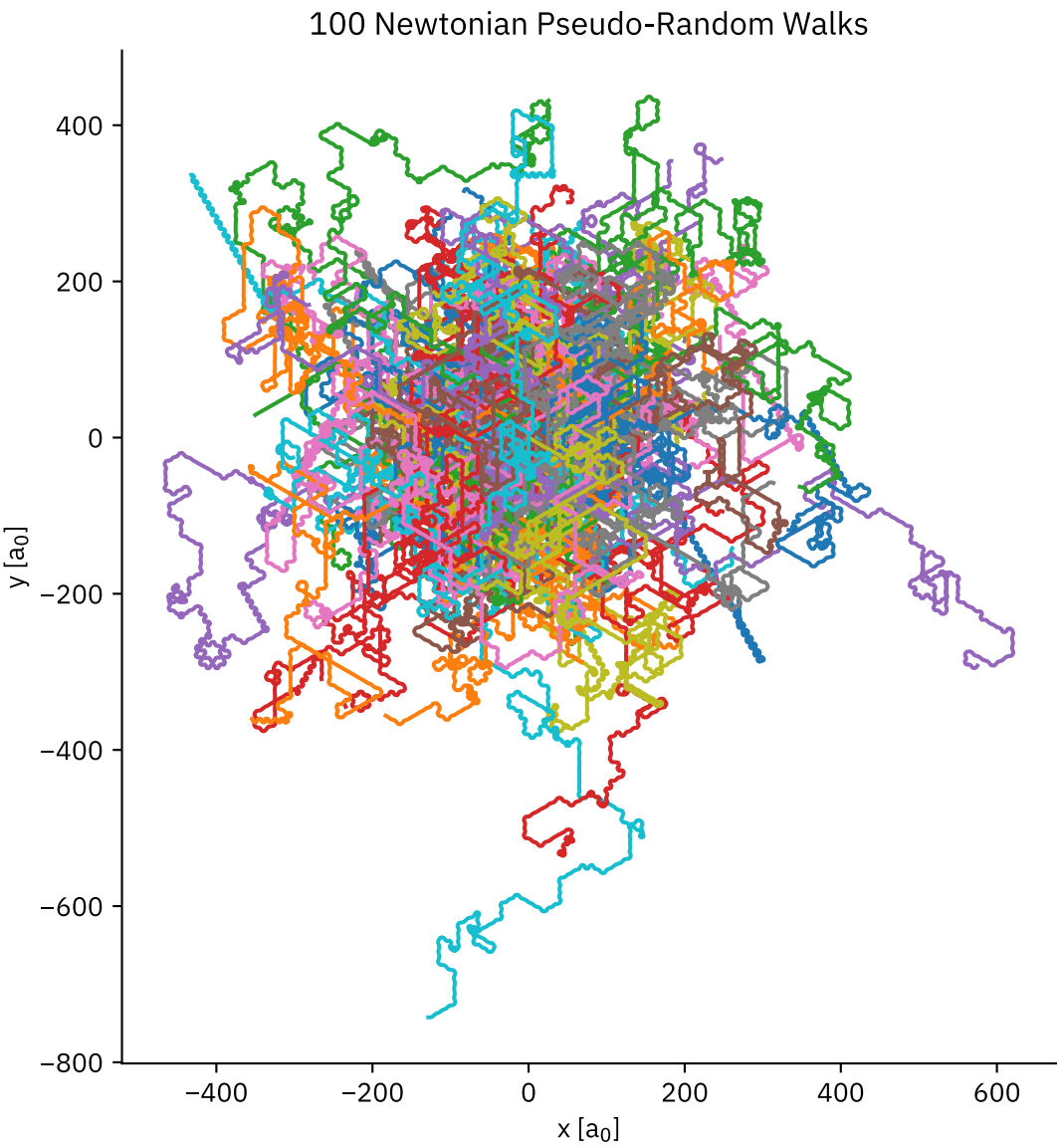
```
traj_1, result_1 = demo_stacie()
```

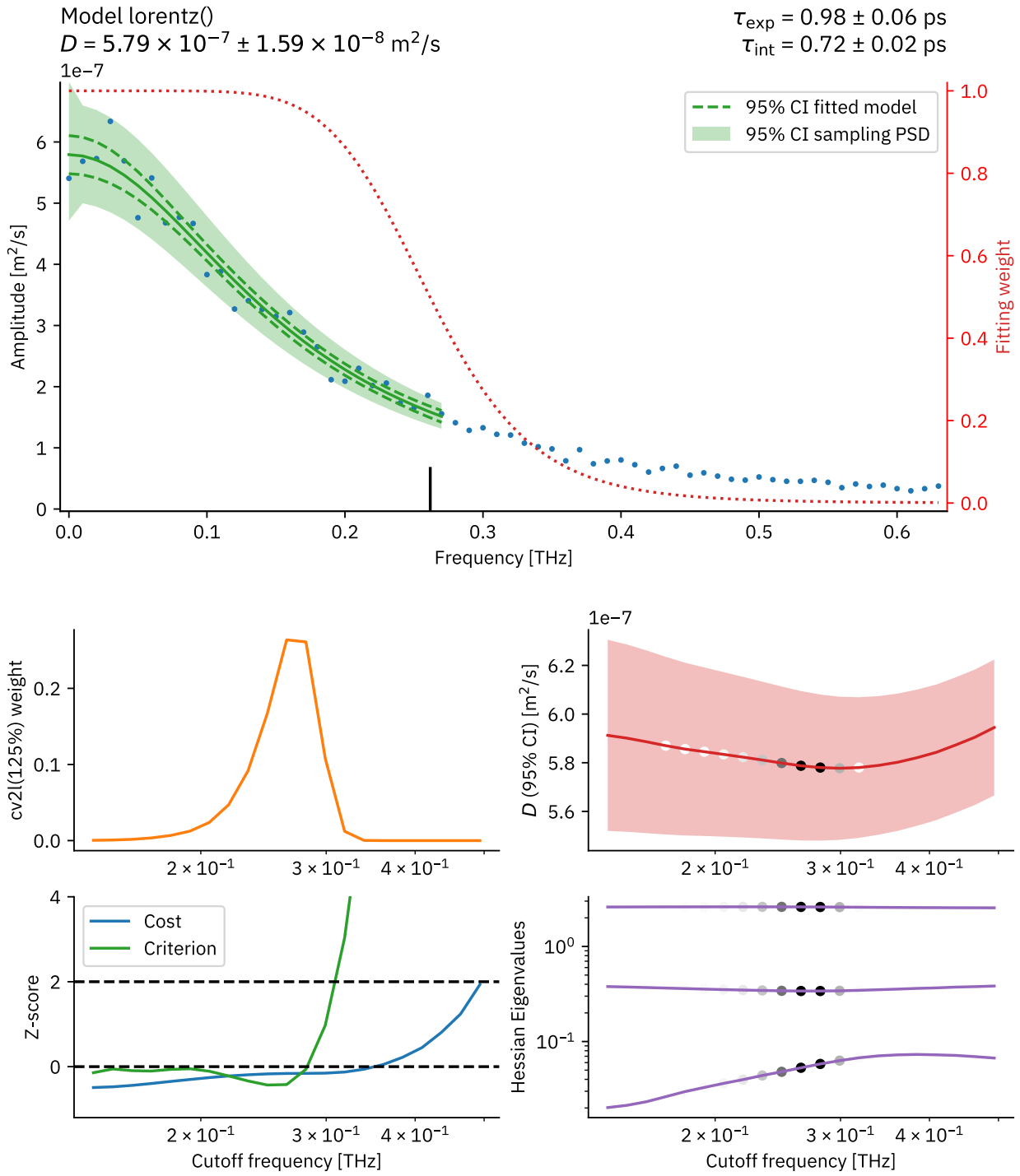
```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion  fcut [THz]
---------  ----------  ----------
```

(continues on next page)

(continued from previous page)

```
     15.0        40.4    1.41e-01
     15.9        39.8    1.50e-01
     16.9        39.0    1.60e-01
     18.0        38.3    1.70e-01
     19.1        37.6    1.81e-01
     20.3        37.0    1.93e-01
     21.6        36.3    2.06e-01
     23.0        35.7    2.19e-01
     24.4        35.0    2.33e-01
     25.9        34.4    2.48e-01
     27.6        33.9    2.64e-01
     29.3        34.0    2.81e-01
     31.2        34.8    2.99e-01
     33.2        37.0    3.18e-01
     35.3        40.9    3.39e-01
     37.5        46.7    3.61e-01
     39.9        56.4    3.84e-01
     42.4        75.9    4.09e-01
     45.1       102.9    4.35e-01
     48.0       133.8    4.63e-01
     51.1       161.2    4.93e-01
Cutoff criterion exceeds incumbent + margin: 33.9 + 100.0.

INPUT TIME SERIES
    Time step:                      0.01 ps
    Simulation time:                100.00 ps
    Maximum degrees of freedom:     400.0

MAIN RESULTS
    Autocorrelation integral:       5.79e-07 ± 1.59e-08 m$^2$/s
    Integrated correlation time:    0.72 ± 0.02 ps

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:     27.4 (ideally > 60)
    Regression cost Z-score:        -0.2 (ideally < 2)
    Cutoff criterion Z-score:       -0.1 (ideally < 2)

MODEL lorentz() | CUTOFF CRITERION cv2l(125%)
    Number of parameters:           3
    Average cutoff frequency:       2.62e-01 THz
    Exponential correlation time:   0.98 ± 0.06 ps

RECOMMENDED SIMULATION SETTINGS (EXPONENTIAL CORR. TIME)
    Block time:                     < 0.31 ± 0.03 ps
    Simulation time:                > 61.70 ± 0.45 ps
```


100 Newtonian Pseudo-Random Walks
400
200
0
−200
−400
−600
−800
y [a0]
−400
−200
0
200
400
600
x [a0]



The spectrum has several peaks related to oscillations of the particles around a local minimum. These peaks are irrelevant to the diffusion coefficient. The broad peak at zero frequency is used by STACIE to derive the diffusion coefficient. The obtained value is not directly comparable to experiment because the 2D lattice model for the surface is not based on an experimental case. However, the order of magnitude is comparable to the self-diffusion constants of pure liquids [BUNO22].

It is also interesting to compare the integrated and exponential autocorrelation time, as they are not the same in this case.

```
print(f"corrtime_exp = {result_1.corrtime_exp / PICOSECOND:.3f} ps")
print(f"corrtime_int = {result_1.corrtime_int / PICOSECOND:.3f} ps")
```

```
corrtime_exp = 0.982 ps
corrtime_int = 0.718 ps
```

The integrated autocorrelation time is smaller than the exponential one because the former is an average of all time scales of the particle velocities. This includes the slow diffusion and faster oscillations in local minima. In contrast, the exponential autocorrelation time only represents the slow diffusive motion.

Finally, it is well known that the velocity autocorrelation function of molecules in a liquid decays according to a power law [AW70]. One might wonder why the Lorentz model can be used here since it implies that diffusion can be described with an exponentially decaying autocorrelation function. The system in this notebook exhibits exponential decay because every particle only interacts with the surface, and not with each other, such that there are no collective modes with long memory effects.

5.4.4 Surface diffusion with block averages

This section repeats the same example, but now with block averages of velocities. *Block averages* are primarily useful for reducing storage requirements when saving trajectories to disk before processing them with STACIE. In this example, the block size is determined by the following guideline:

```
print(np.pi * result_1.corrtime_exp / (10 * TIMESTEP))
```

```
61.70265956349357
```

Let's use a block size of 60 to stay on the safe side.

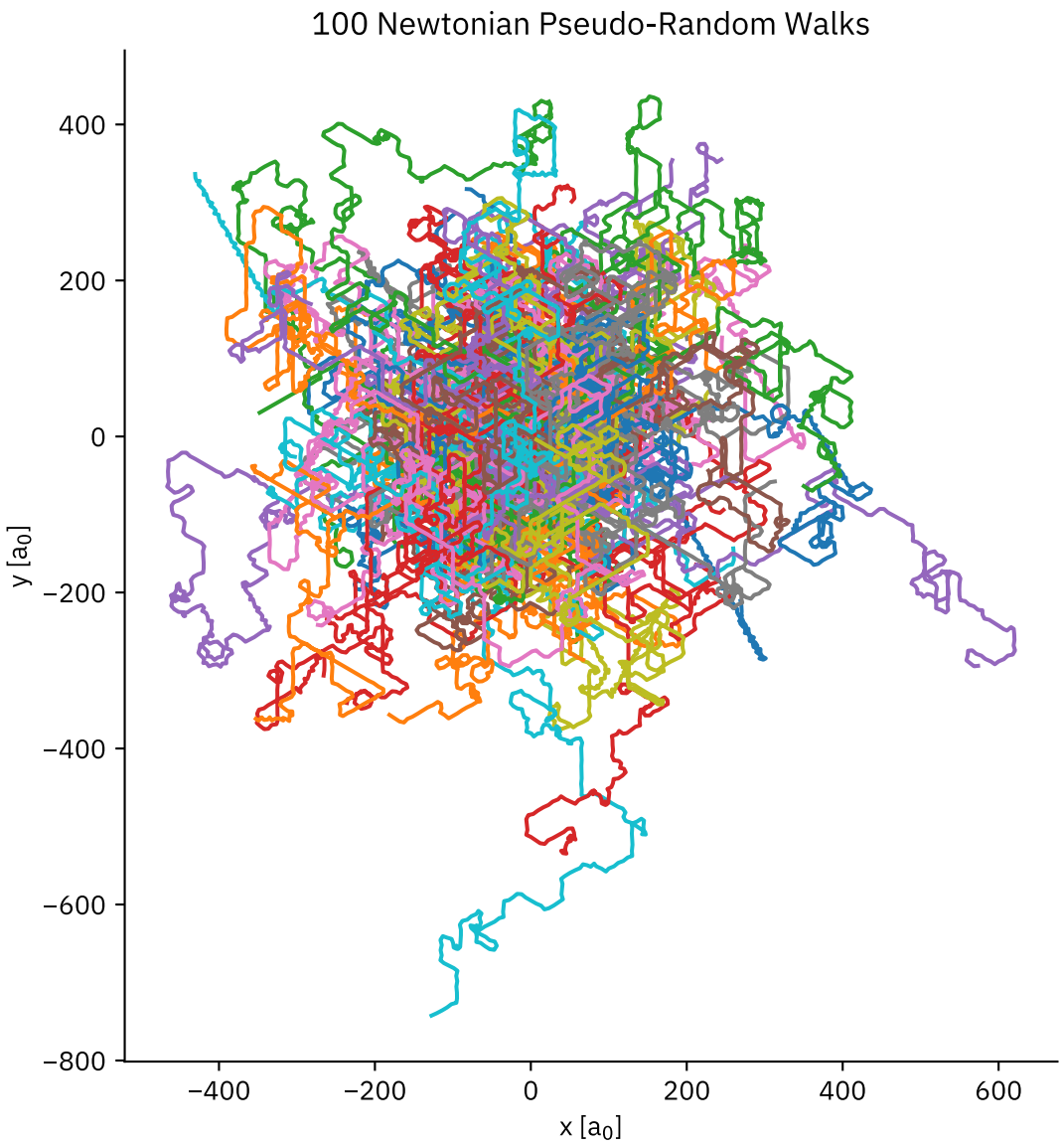
```
traj_60, result_60 = demo_stacie(60)
```

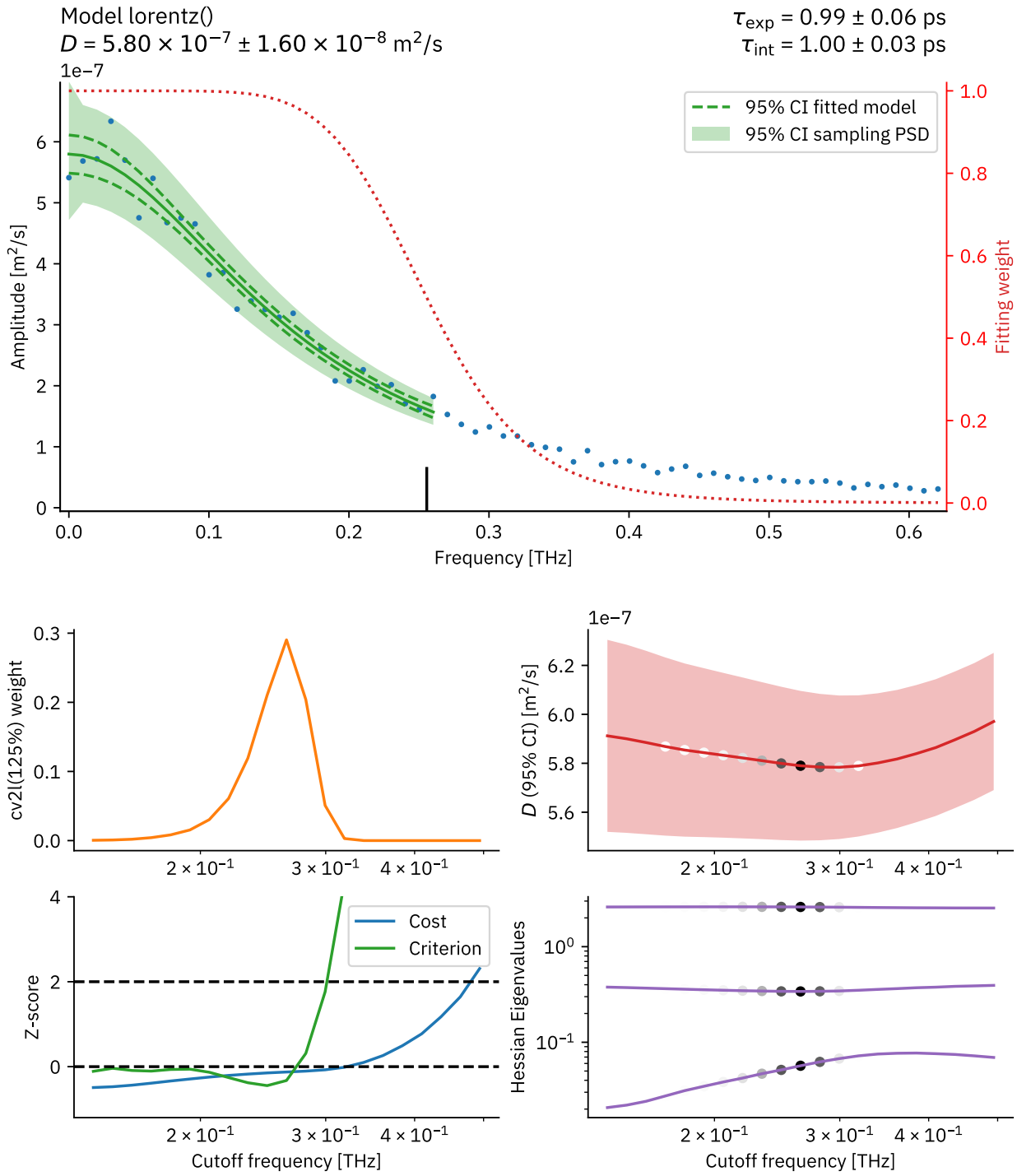
```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion  fcut [THz]
---------  ----------  ----------
     15.0        40.4    1.41e-01
     15.9        39.7    1.51e-01
     16.9        38.9    1.60e-01
     18.0        38.1    1.71e-01
     19.1        37.5    1.82e-01
     20.3        36.9    1.93e-01
     21.6        36.2    2.06e-01
     23.0        35.5    2.19e-01
     24.4        34.8    2.33e-01
     25.9        34.2    2.48e-01
     27.6        33.9    2.64e-01
     29.3        34.3    2.81e-01
     31.2        35.7    2.99e-01
     33.2        38.5    3.19e-01
     35.3        43.0    3.39e-01
     37.5        49.4    3.61e-01
     39.9        59.1    3.84e-01
     42.4        76.9    4.09e-01
```

(continues on next page)

(continued from previous page)

```
     45.1        97.5    4.36e-01
     48.0       118.6    4.64e-01
     51.1       135.8    4.94e-01
Cutoff criterion exceeds incumbent + margin: 33.9 + 100.0.

INPUT TIME SERIES
    Time step:                      0.30 ps
    Simulation time:                99.90 ps
    Maximum degrees of freedom:     400.0

MAIN RESULTS
    Autocorrelation integral:       5.80e-07 ± 1.60e-08 m$^2$/s
    Integrated correlation time:    1.00 ± 0.03 ps

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:     26.7 (ideally > 60)
    Regression cost Z-score:        -0.1 (ideally < 2)
    Cutoff criterion Z-score:       -0.1 (ideally < 2)

MODEL lorentz() | CUTOFF CRITERION cv2l(125%)
    Number of parameters:           3
    Average cutoff frequency:       2.56e-01 THz
    Exponential correlation time:   0.99 ± 0.06 ps

RECOMMENDED SIMULATION SETTINGS (EXPONENTIAL CORR. TIME)
    Block time:                     < 0.31 ± 0.03 ps
    Simulation time:                > 61.99 ± 0.45 ps
```


100 Newtonian Pseudo-Random Walks
400
200
0
−200
−400
−600
−800
y [a0]
−400
−200
0
200
400
600
x [a0]



As expected, there are no significant changes in the results.

It is again interesting to compare the integrated and exponential autocorrelation times.

```
print(f"corrtime_exp = {result_60.corrtime_exp / PICOSECOND:.3f} ps")
print(f"corrtime_int = {result_60.corrtime_int / PICOSECOND:.3f} ps")
```

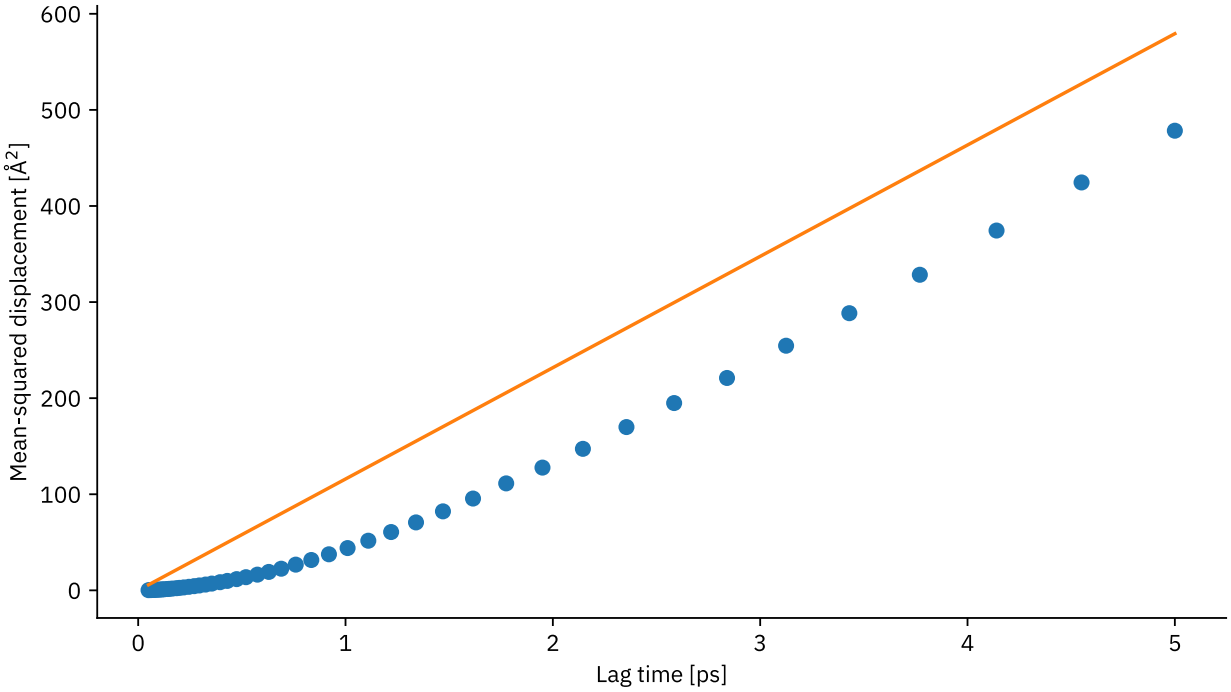
```
corrtime_exp = 0.987 ps
corrtime_int = 0.997 ps
```

The exponential autocorrelation time is unaffected by the block averages. However, the integrated autocorrelation time has increased and is now closer to the exponential value. Taking block averages removes the fastest oscillations, causing the integrated autocorrelation time to be dominated by slow diffusive motion.

5.4.5 Comparison to mean-squared displacement analysis

This section does not perform a full regression to derive the diffusion coefficient from the mean-squared displacement (MSD) data. Instead, it simply computes the MSDs from the trajectories, and plots them together with the expected slope from STACIE's analysis above, to confirm that STACIE's results are consistent with the MSD analysis. This avoids the pernicious choices required for a full regression analysis of the MSD data.

```
def plot_msd(traj, result, lags, time_step):
    """Plot mean-squared displacements from trajectory and STACIE result.

    Parameters
    ----------
    traj
        The trajectory object.
    result
        The STACIE result object.
    lags
        The integer lag times (in number of (block) time steps)
        for which to compute the MSDs.
    time_step
        The time step size of the trajectory, possibly accounting for the block size.
    """
    lag_times = lags * time_step
    natom = traj.coords.shape[0]
    msds = compute_msds(traj.coords.reshape(natom * 2, traj.nstep), lags)
    plt.close("msd")
    _, ax = plt.subplots(num="msd")
    ax.plot(
        lag_times / PICOSECOND,
        msds / ANGSTROM**2,
        "C0o",
        label="MSD from trajectories",
    )
    ax.plot(
        lag_times / PICOSECOND,
        2 * result.acint * lag_times / ANGSTROM**2,
        "C1-",
        label="Expected slope",
    )
    ax.set_xlabel("Lag time [ps]")
    ax.set_ylabel("Mean-squared displacement [Å$^2$]")


lags = np.unique(np.logspace(1, 3, 50).astype(int))
plot_msd(traj_1, result_1, lags, TIMESTEP)
```



As expected, the simple comparison confirms that STACIE's results are consistent with the MSD analysis. For sufficiently large lag times, the MSDs increase linearly with time, with a slope that corresponds to the diffusion coefficient derived with STACIE.

Note that STACIE only estimates the slope of a straight line fitted to the MSD curve. It does not provide information on the intercept.

5.4.6 Regression Tests

If you are experimenting with this notebook, you can ignore any exceptions below. The tests are only meant to pass for the notebook in its original form.

```
acint_unit = sc.value("atomic unit of time") / sc.value("atomic unit of length") ** 2
acint_1 = result_1.acint / acint_unit
if abs(acint_1 - 5.80e-7) > 5e-9:
    raise ValueError(f"Wrong acint (no block average): {acint_1:.2e}")
acint_60 = result_60.acint / acint_unit
if abs(acint_60 - 5.80e-7) > 5e-9:
    raise ValueError(f"Wrong acint (block size 60): {acint_60:.2e}")
```

The remaining notebooks process the output of external simulation codes. Input files for these simulations can be found in the Git source repository of STACIE. You can rerun these simulations to generate the required data or use the data files from the ZIP archive mentioned above.

5.5 Shear Viscosity of a Lennard-Jones Liquid Near the Triple Point (LAMMPS)

This example shows how to calculate viscosity of argon from pressure tensor data obtained from *LAMMPS MD* simulations. The required theoretical background is explained the *Shear Viscosity* section. The same simulations are also used for the *bulk viscosity* and *thermal conductivity* examples in the following two notebooks. The goal of the argon examples is to derive the three transport properties with a relative error smaller than those found in the literature.

All argon MD simulations use the Lennard-Jones potential with reduced Lennard-Jones units. For example, the reduced unit of viscosity is denoted as η^*, and the reduced unit of time as τ^*. The simulated system consists of 1372 argon atoms. The thermodynamic state $\rho = 0.8442\,\rho^*$ and $T = 0.722\,\mathrm{T}^*$ corresponds to a liquid phase near the triple point ($\rho = 0.0845\,\rho^*$ and $T = 0.69\,\mathrm{T}^*$). This liquid state is known to exhibit slow relaxation times, which complicates the convergence of transport properties and makes it a popular test case for computational methods.

The LAMMPS input files can be found in the directory `docs/data/lammps_lj3d` of STACIE's Git source repository. To obtain sufficient data for all three properties, we performed 100 independent runs, for which the *guesstimated relative error* is tabulated below. The *Lorentz model* is used to fit the spectrum, with degrees $S_{\mathrm{num}} = \{0, 2\}$ and $S_{\mathrm{den}} = \{2\}$, corresponding to $P = 3$ parameters.

Property	M	Guess rel. error
Shear viscosity	500	0.5 %
Bulk viscosity	100	1.2 %
Thermal conductivity	300	0.7 %

The (initial) settings for the production runs were determined as follows. In general, the integration time step in MD simulations roughly corresponds to one tenth of a period of the fastest oscillations in the system. At shorter time scales than 10 steps, the dynamics is most likely irrelevant for transport properties. Hence, in our first simulations, all data was recorded with block averages of 10 steps. As mentioned in the section on the *block averages*, at least $400P$ blocks are recommended. The initial production runs therefore consisted of 12000 MD steps. Note that these values are only coarse estimates. As explained below, the production runs were extended twice to improve the statistics.

Details of the MD simulations can be found the LAMMPS inputs `docs/data/lammps_lj3d/template-init.lammps` and `docs/data/lammps_lj3d/template-ext.lammps` in STACIE's Git repository. These input files are actually Jinja2 templates that are rendered with different random seeds (and restart files) for each run. The initial production simulations start from an FCC crystal structure, which is first melted for 5000 steps at an elevated temperature of $T = 1.5\,\mathrm{T}^*$ in the *NVT* ensemble. The system is then equilibrated at the desired temperature of $T = 0.722\,\mathrm{T}^*$ for 5000 additional steps. Starting from the equilibrated states, production runs were performed in the *NVE* ensemble. The velocities are not rescaled after the NVT equilibration, to ensure that the set of NVE runs as a whole is representative of the NVT ensemble. During the production phase, trajectory data is collected with block averages over 10 steps.

The LAMMPS input files contain commands to write output files that can be directly loaded using Python and NumPy without any additional converters or wrappers. The following output files from `docs/data/lammps_lj3d/sims/replica_????_part_??/` were used for the analysis:

- `info.yaml`: simulation settings that may be useful for post-processing.
- `nve_thermo.txt`: subsampled instantaneous temperature and related quantities
- `nve_pressure_blav.txt`: block-averaged (off)diagonal pressure tensor components
- `nve_heatflux_blav.txt`: block-averaged x, y, and z components of the heat flux vector, i.e. J^{h}_x, J^{h}_y, and J^{h}_z. Heat fluxes are used in the thermal conductivity example, not in this notebook.

Note

The results in this example were obtained using LAMMPS 29 Aug 2024 Update 3. Minor differences may arise when using a different version of LAMMPS, or even the same version compiled with a different compiler.

5.5.1 Library Imports and Configuration

```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from path import Path
from yaml import safe_load
from scipy.stats import chi2
from stacie import (
    UnitConfig,
    compute_spectrum,
    estimate_acint,
    LorentzModel,
    plot_fitted_spectrum,
    plot_extras,
)
from utils import plot_instantaneous_percentiles, plot_cumulative_temperature_histogram
```

```
mpl.rc_file("matplotlibrc")
%config InlineBackend.figure_formats = ["svg"]
```

```
# You normally do not need to change this path.
# It only needs to be overridden when building the documentation.
DATA_ROOT = Path(os.getenv("DATA_ROOT", "./")) / "lammps_lj3d/sims/"
```

5.5.2 Analysis of the Equilibration Runs

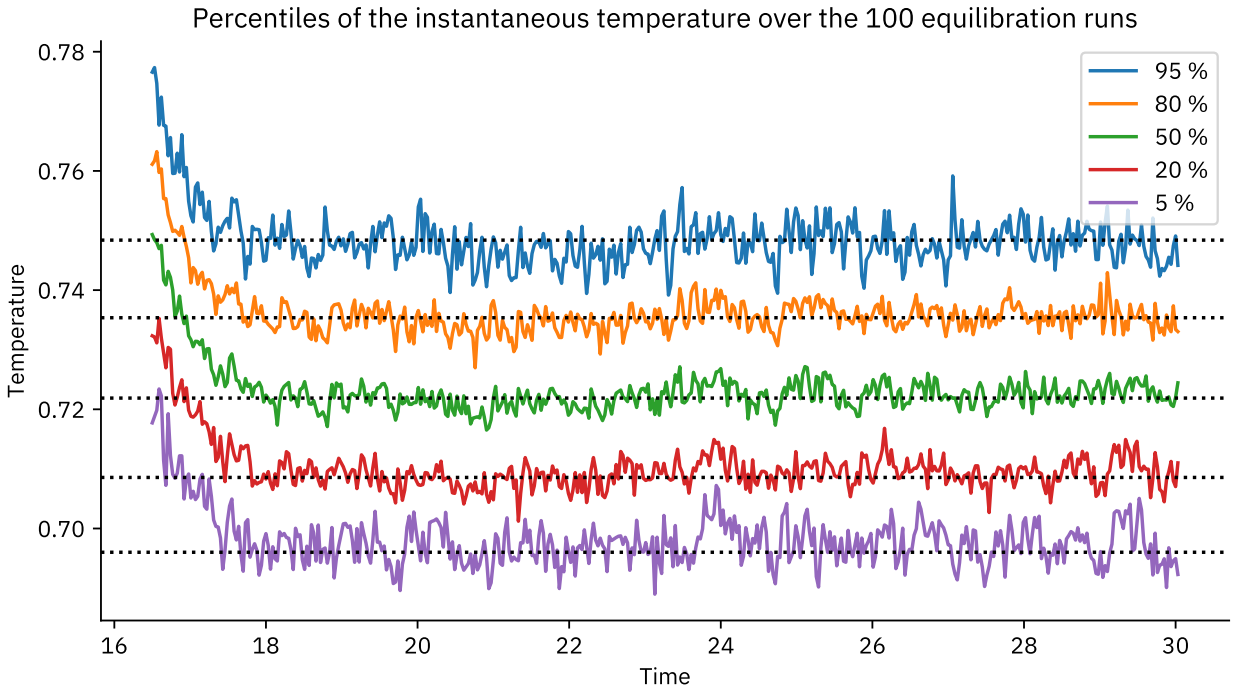
To ensure that the production runs start from a well-equilibrated state, we first analyze the equilibration runs. The following code cell plots percentiles of the instantaneous temperature as a function of time over all independent runs. For reference, the theoretical percentiles of the NVT ensemble are shown as horizontal dotted lines.

```
def plot_equilibration(ntraj: int = 100):
    """Plot percentiles of the instantaneous temperature."""
    # Load the configuration from the YAML file.
    with open(DATA_ROOT / "replica_0000_part_00/info.yaml") as fh:
        info = safe_load(fh)
    temp_d = info["temperature"]
    ndof = info["natom"] * 3 - 3

    # Load trajectory data.
    temps = []
    time = None
    for itraj in range(ntraj):
        equil_dir = DATA_ROOT / f"replica_{itraj:04d}_part_00/"
        data = np.loadtxt(equil_dir / "nvt_thermo.txt")
        temps.append(data[:, 1])
        if time is None:
            time = info["block_size"] * info["timestep"] * np.arange(len(data))
    temps = np.array(temps)

    # Select the last part (final temperature), discarding the melting phase.
    temps = temps[:, 550:]
```

(continues on next page)

(continued from previous page)

```
    time = time[550:]

    # Plot the instantaneous and desired temperature.
    plt.close("tempequil")
    _, ax = plt.subplots(num="tempequil")
    percents = [95, 80, 50, 20, 5]
    plot_instantaneous_percentiles(
        ax,
        time,
        temps,
        percents,
        expected=[chi2.ppf(percent / 100, ndof) * temp_d / ndof for percent in percents],
    )
    ax.set_title(
        "Percentiles of the instantaneous temperature over the 100 equilibration runs"
    )
    ax.set_ylabel("Temperature")
    ax.set_xlabel("Time")


plot_equilibration()
```



The plot shows that the equilibration runs were successful: They reach the correct average temperature **and** also exhibit the expected fluctuations. Note that we used a Langevin thermestat for equilibration. This is a robust local thermostat that quickly brings all degrees of freedom to the desired temperature. In comparison, a global Nosé-Hoover-chain (NHC) thermostat would still show large oscillations in the temperature, even after 5000 steps. Taking the last state from an NHC run generally results in biased initial conditions for the NVE runs. (You can see the difference by modifying the LAMMPS input files, rerunning them and then rerunning this notebook.)

5.5.3 Analysis of the Initial Production Simulations

The following code cell defines analysis functions:

- `get_indep_paniso` transforms the pressure tensor components into five independent anisotropic contributions, as explained in the *Shear Viscosity* theory section.
- `estimate_viscosity` calculates the viscosity and plots the results. It also prints recommendations for data reduction (block averaging) and simulation time, as explained in the following two sections of the documentation:
  - *Integrated and Exponential Autocorrelation Time*
  - *Reducing Storage Requirements with Block Averages*

  These will be used to determine whether our initial simulation settings are appropriate.

```
def get_indep_paniso(pcomps):
    return np.array(
        [
            (pcomps[0] - 0.5 * pcomps[1] - 0.5 * pcomps[2]) / np.sqrt(3),
            0.5 * pcomps[1] - 0.5 * pcomps[2],
            pcomps[3],
            pcomps[4],
            pcomps[5],
        ]
    )


def estimate_viscosity(name, pcomps, av_temperature, volume, timestep, verbose=True):
    # Create the spectrum of the pressure fluctuations.
    # Note that the Boltzmann constant is 1 in reduced LJ units.
    uc = UnitConfig(
        acint_fmt=".3f",
        acint_symbol="η",
        acint_unit_str="η*",
        freq_unit_str="1/τ*",
        time_fmt=".3f",
        time_unit_str="τ*",
    )
    spectrum = compute_spectrum(
        pcomps,
        prefactors=volume / av_temperature,
        timestep=timestep,
    )

    # Estimate the viscosity from the spectrum.
    result = estimate_acint(spectrum, LorentzModel(), verbose=verbose, uc=uc)

    if verbose:
        # Plot some basic analysis figures.
        plt.close(f"{name}_spectrum")
        _, ax = plt.subplots(num=f"{name}_spectrum")
        plot_fitted_spectrum(ax, uc, result)
        plt.close(f"{name}_extras")
        _, axs = plt.subplots(2, 2, num=f"{name}_extras")
        plot_extras(axs, uc, result)
```

(continues on next page)

(continued from previous page)

```
    # Return the viscosity
    return result
```

The next cell performs the analysis of the initial simulations. It prints the recommended block size and the simulation time for the production runs, and then generates two figures:

- The spectrum of the off-diagonal pressure fluctuations, and the model fitted to the spectrum.
- Additional intermediate results.

```
def analyze_production(npart: int, ntraj: int = 100, select: int | None = None):
    """
    Perform the analysis of the production runs.

    Parameters
    ----------
    npart
        Number of parts in the production runs.
        For the initial production runs, this is 1.
    ntraj
        Number of trajectories in the production runs.
    select
        If `None`, all anisotropic contributions are selected.
        If not `None`, only select the given anisotropic contribution
        for the viscosity estimate. Must be one of `0`, `1`, `2`, `3`, `4`, `None`.

    Returns
    -------
    result
        The result from STACIE's `estimate_acint` function,
    """
    # Load the configuration from the YAML file.
    with open(DATA_ROOT / "replica_0000_part_00/info.yaml") as fh:
        info = safe_load(fh)

    # Load trajectory data.
    thermos = []
    pcomps_full = []
    for itraj in range(ntraj):
        thermos.append([])
        pcomps_full.append([])
        for ipart in range(npart):
            prod_dir = DATA_ROOT / f"replica_{itraj:04d}_part_{ipart:02d}/"
            thermos[-1].append(np.loadtxt(prod_dir / "nve_thermo.txt"))
            pcomps_full[-1].append(np.loadtxt(prod_dir / "nve_pressure_blav.txt"))
    thermos = [np.concatenate(parts).T for parts in thermos]
    pcomps_full = [np.concatenate(parts).T for parts in pcomps_full]
    av_temperature = np.mean([thermo[1] for thermo in thermos])

    # Compute the viscosity.
    pcomps_aniso = np.concatenate([get_indep_paniso(p[1:]) for p in pcomps_full])
    if select is not None:
        if select < 0 or select > 4:
            raise ValueError(f"Invalid selection {select}, must be in [0, 4]")
        pcomps_aniso = pcomps_aniso[select::5]
```

(continues on next page)

(continued from previous page)

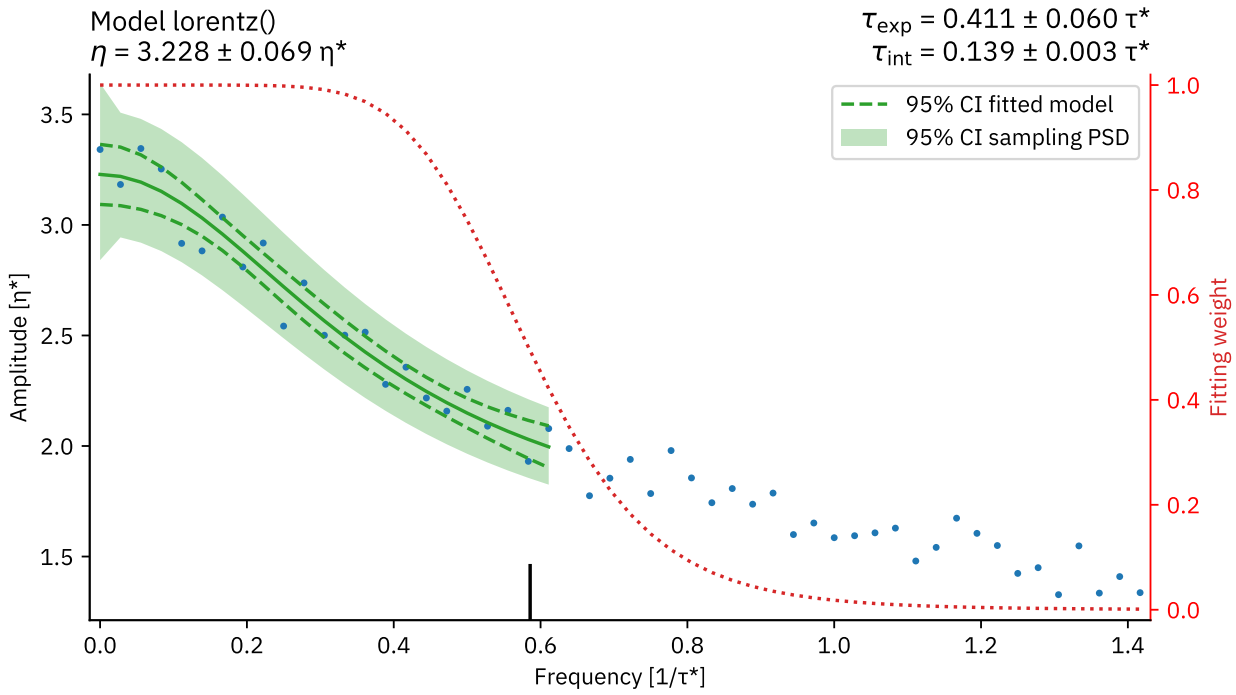
```
    return estimate_viscosity(
        f"part{npart}",
        pcomps_aniso,
        av_temperature,
        info["volume"],
        info["timestep"] * info["block_size"],
        verbose=select is None,
    )


eta_production_init = analyze_production(1).acint
```

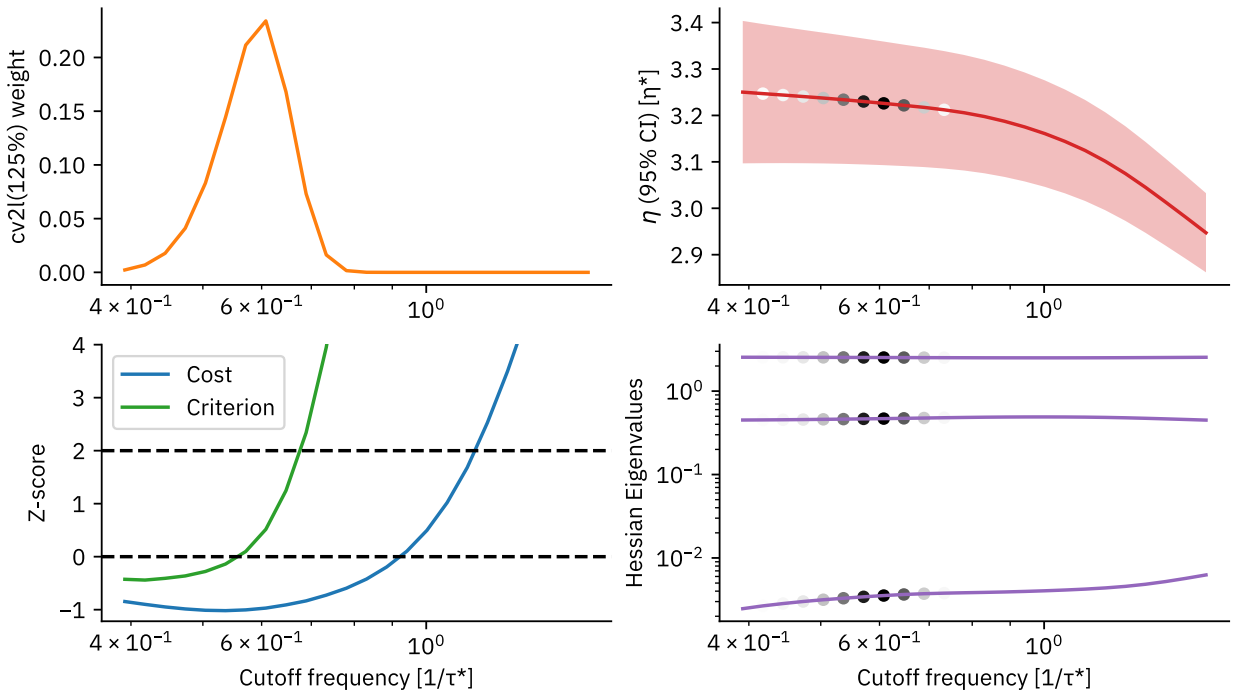
```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion  fcut [1/τ*]
---------  ----------  ----------
     15.0        12.5    3.93e-01
     15.9        11.4    4.18e-01
     16.9        10.4    4.45e-01
     18.0         9.6    4.73e-01
     19.1         8.9    5.04e-01
     20.3         8.3    5.36e-01
     21.6         7.9    5.71e-01
     23.0         7.8    6.08e-01
     24.4         8.2    6.47e-01
     25.9         9.0    6.89e-01
     27.6        10.5    7.33e-01
     29.3        12.8    7.81e-01
     31.2        16.1    8.31e-01
     33.2        20.5    8.85e-01
     35.3        26.3    9.42e-01
     37.5        33.2    1.00e+00
     39.9        41.1    1.07e+00
     42.4        49.5    1.14e+00
     45.1        58.0    1.21e+00
     48.0        67.0    1.29e+00
     51.1        76.9    1.37e+00
     54.4        88.4    1.46e+00
     57.8       102.9    1.55e+00
     61.5       121.4    1.65e+00
Cutoff criterion exceeds incumbent + margin: 7.8 + 100.0.

INPUT TIME SERIES
    Time step:                      0.030 τ*
    Simulation time:                36.000 τ*
    Maximum degrees of freedom:     1000.0

MAIN RESULTS
    Autocorrelation integral:       3.228 ± 0.069 η*
    Integrated correlation time:    0.139 ± 0.003 τ*

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:     22.1 (ideally > 60)
    Regression cost Z-score:        -1.0 (ideally < 2)
```

(continues on next page)

(continued from previous page)

```
    Cutoff criterion Z-score:      0.5 (ideally < 2)

MODEL lorentz() | CUTOFF CRITERION cv2l(125%)
    Number of parameters:          3
    Average cutoff frequency:      5.86e-01 1/τ*
    Exponential correlation time:  0.411 ± 0.060 τ*

RECOMMENDED SIMULATION SETTINGS (EXPONENTIAL CORR. TIME)
    Block time:                    < 0.129 ± 0.034 τ*
    Simulation time:               > 25.821 ± 0.477 τ*
```





Several things can be observed in the analysis of the initial production runs:

- The recommendations based on the exponential correlation time were met by the initial simulation settings.
  - The recommended simulation time is 25.8 τ^*, which about 8600 steps. The initial production runs (12000 steps) were therefore sufficient.
  - The recommended block time is 0.129 τ^*, which corresponds to about 40 steps. The block size used in the initial production runs (10 steps) was sufficiently small.
- The relative error of the viscosity estimate is about 2%, which is larger than the guesstimated value 0.5%. This is fine and somewhat expected, since this guess is known to be crude.
- The Lorentz model used to fit the spectrum was a fair choice, but for higher frequencies, the sampling PSD clearly decays faster than the fitted model. For the case of viscosity, there is (to the best of our knowledge) no solid theoretical argument to support the exponential decay of the ACF of the pressure tensor. It just seems to be a reasonable choice for this case.
- The effective number of points fitted to the spectrum is 22.1, which is low for a 3 parameter model. For high-quality production simulations, it would be good to triple the simulation length, as to multiply the resolution of the frequency grid by 3. This is hopefully sufficient to reach 60 effective points.

As can be seen in the comparison to literature results below, the results for the initial production runs were already quite good. However, for the sake of demonstration, the production runs were extended by an additional 24000 steps each, to triple the simulation time. This revealed that the effective number of points fitted to the spectrum increase to 61, which is a sublinear increase, just enough to reach the target of 60. For the sake of demonstration, we decided to extend the production runs by another 64000 steps, which resulted in a total simulation time of 300 τ^* per run.

The difficulty of increasing the effective number of fitted points can be understood as follows. The Lorentz model is not capable of fitting the spectrum to higher frequencies. By including more data points, the limitations of the approximating model also become clearer, and the cutoff criterion will detect some underfitting (and thus risk for bias) at lower cutoffs.

5.5.4 Analysis of the Production Simulations

Here we just repeat the analysis, but now with extended production runs.

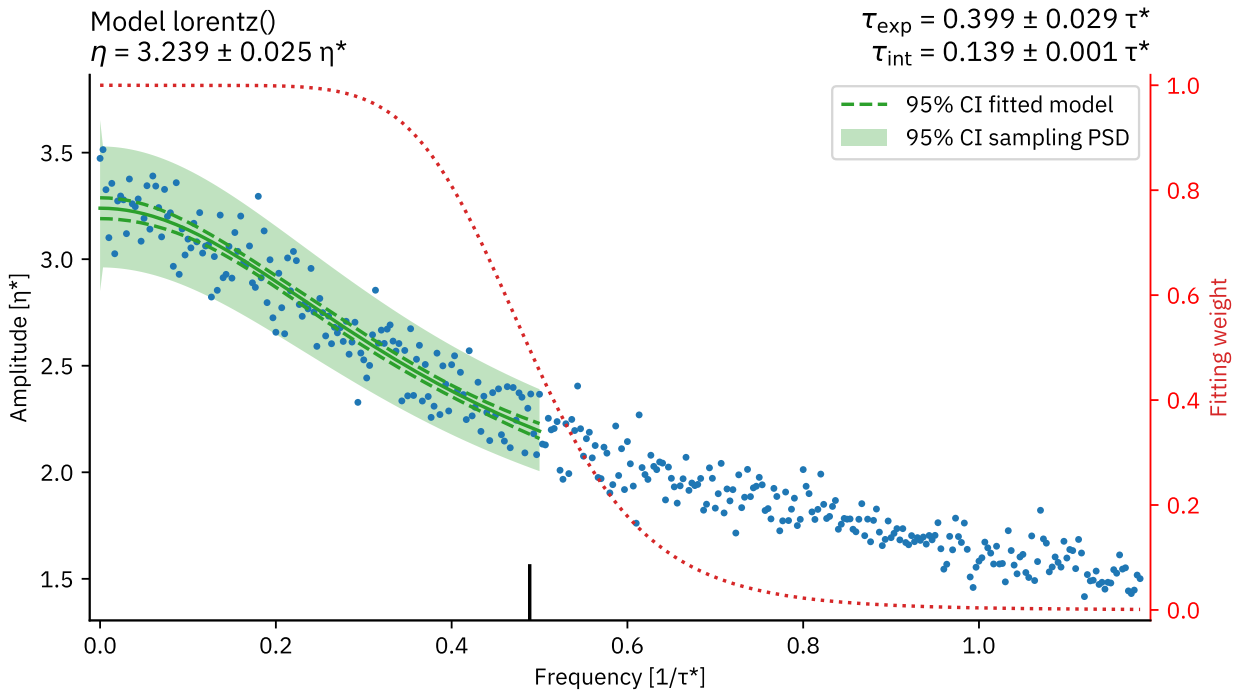
```
eta_production_ext = analyze_production(3).acint
```

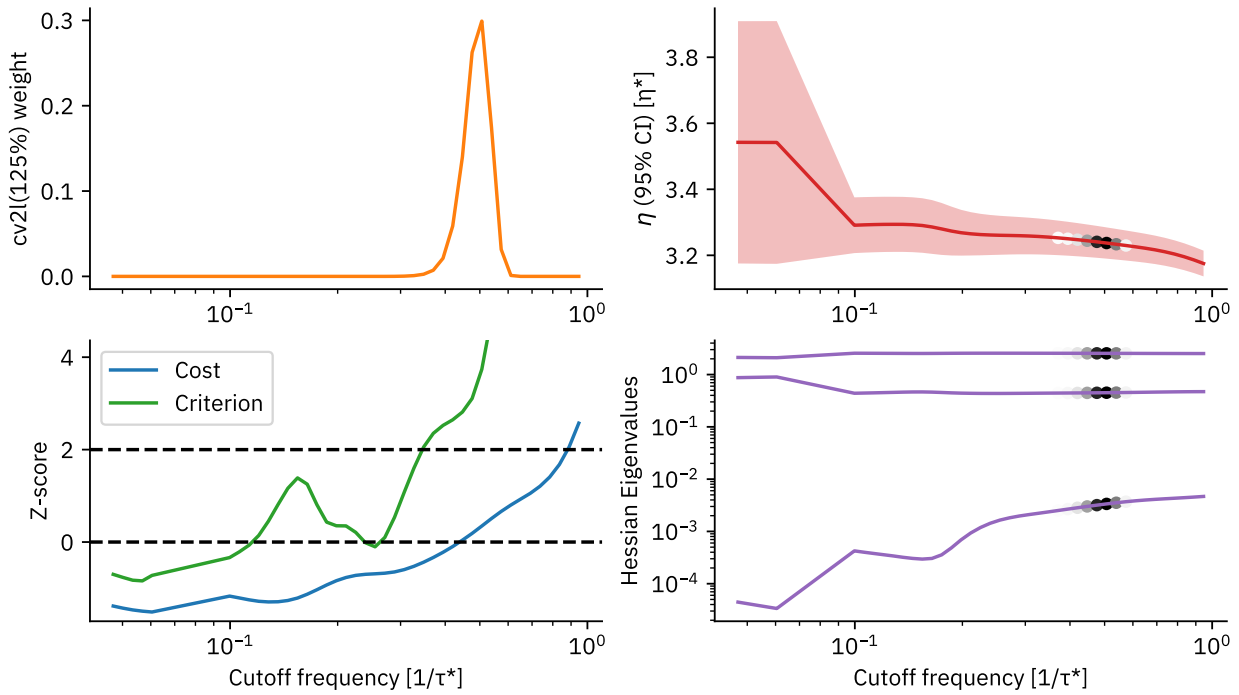
```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion  fcut [1/τ*]
---------  ----------  ----------
     15.0        41.2    4.71e-02
     15.9        41.4    5.01e-02
     16.9        41.5    5.34e-02
     18.0        41.7    5.68e-02
     19.1        42.1    6.05e-02
     20.3         inf    6.44e-02  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     21.6         inf    6.85e-02  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     23.0         inf    7.30e-02  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     24.4         inf    7.77e-02  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     25.9         inf    8.27e-02  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     27.6         inf    8.80e-02  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     29.3         inf    9.37e-02  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     31.2        99.4    9.97e-02
     33.2        81.4    1.06e-01
     35.3        69.9    1.13e-01
     37.5        61.6    1.20e-01
     39.9        56.0    1.28e-01
     42.4        51.7    1.36e-01
     45.1        48.9    1.45e-01
     48.0        47.4    1.54e-01
     51.1        46.8    1.64e-01
     54.4        45.8    1.75e-01
     57.8        43.6    1.86e-01
     61.5        40.2    1.98e-01
     65.5        36.0    2.11e-01
     69.7        31.7    2.25e-01
     74.1        27.5    2.39e-01
     78.9        24.0    2.55e-01
     83.9        21.3    2.71e-01
     89.3        19.4    2.89e-01
     95.0        17.9    3.07e-01
    101.1        16.8    3.27e-01
    107.6        15.8    3.48e-01
    114.5        14.7    3.70e-01
    121.9        13.6    3.94e-01
    129.7        12.6    4.20e-01
    138.0        11.7    4.47e-01
    146.9        11.1    4.76e-01
    156.3        11.0    5.06e-01
    166.4        11.5    5.39e-01
    177.1        13.2    5.74e-01
    188.5        16.6    6.11e-01
    200.6        22.3    6.50e-01
    213.5        30.8    6.92e-01
    227.2        42.5    7.37e-01
    241.9        57.6    7.84e-01
```

(continues on next page)

(continued from previous page)

```
    257.4        75.9    8.35e-01
    274.0        97.8    8.89e-01
    291.6       124.0    9.46e-01
Cutoff criterion exceeds incumbent + margin: 11.0 + 100.0.

INPUT TIME SERIES
    Time step:                      0.030 τ*
    Simulation time:                300.000 τ*
    Maximum degrees of freedom:     1000.0

MAIN RESULTS
    Autocorrelation integral:       3.239 ± 0.025 η*
    Integrated correlation time:    0.139 ± 0.001 τ*

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:     150.9 (ideally > 60)
    Regression cost Z-score:        0.3 (ideally < 2)
    Cutoff criterion Z-score:       3.6 (ideally < 2)

MODEL lorentz() | CUTOFF CRITERION cv2l(125%)
    Number of parameters:           3
    Average cutoff frequency:       4.89e-01 1/τ*
    Exponential correlation time:   0.399 ± 0.029 τ*

RECOMMENDED SIMULATION SETTINGS (EXPONENTIAL CORR. TIME)
    Block time:                     < 0.125 ± 0.016 τ*
    Simulation time:                > 25.047 ± 0.228 τ*
```





Some remarks about the final results:

- The effective number of points has increased to 150.9, which is a fine number of data points for a model with $P = 3$ parameters.
- For higher frequency cutoffs, both Z-scores increase, showing that the autocorrelation function only decays exponentially in the limit of large lag times. This is expected, since at sufficiently short time scales, the pressure tensor fluctuations are smooth functions, i.e. not featuring the cusp of a purely exponential ACF.

5.5.5 Comparison to Literature Results

Comprehensive literature surveys on computational estimates of the shear viscosity of a Lennard-Jones fluid can be found in [MLK04a] and [VSG07a]. These papers also present new results, which are included in the table below. Since the simulation settings ($r^*_{\text{cut}} = 2.5$, $N = 1372$, $T^* = 0.722$ and $\rho^* = 0.8442$) are identical to those used in this notebook, the reported values should be directly comparable.

Method	Simulation time [τ*]	Shear viscosity [η*]	Reference
EMD NVE (STACIE)	3600	3.228 ± 0.069	(here) initial
EMD NVE (STACIE)	10800	3.208 ± 0.041	(here) extension 1
EMD NVE (STACIE)	30000	3.239 ± 0.025	(here) extension 2
EMD NVE (Helfand-Einstein)	75000	3.277 ± 0.098	[MLK04a]
EMD NVE (Helfand-moment)	600000	3.268 ± 0.055	[VSG07a]

This comparison confirms that STACIE can reproduce a well-known viscosity result, in line with literature results. To achieve a state-of-the-art statistical *uncertainty*, it requires far less simulation time. Even our longest production runs are still less than half as long as in the cited papers, and we achieve a much smaller uncertainties.

To be fair, the simulation time only accounts for production runs. Our setup also includes a sig-

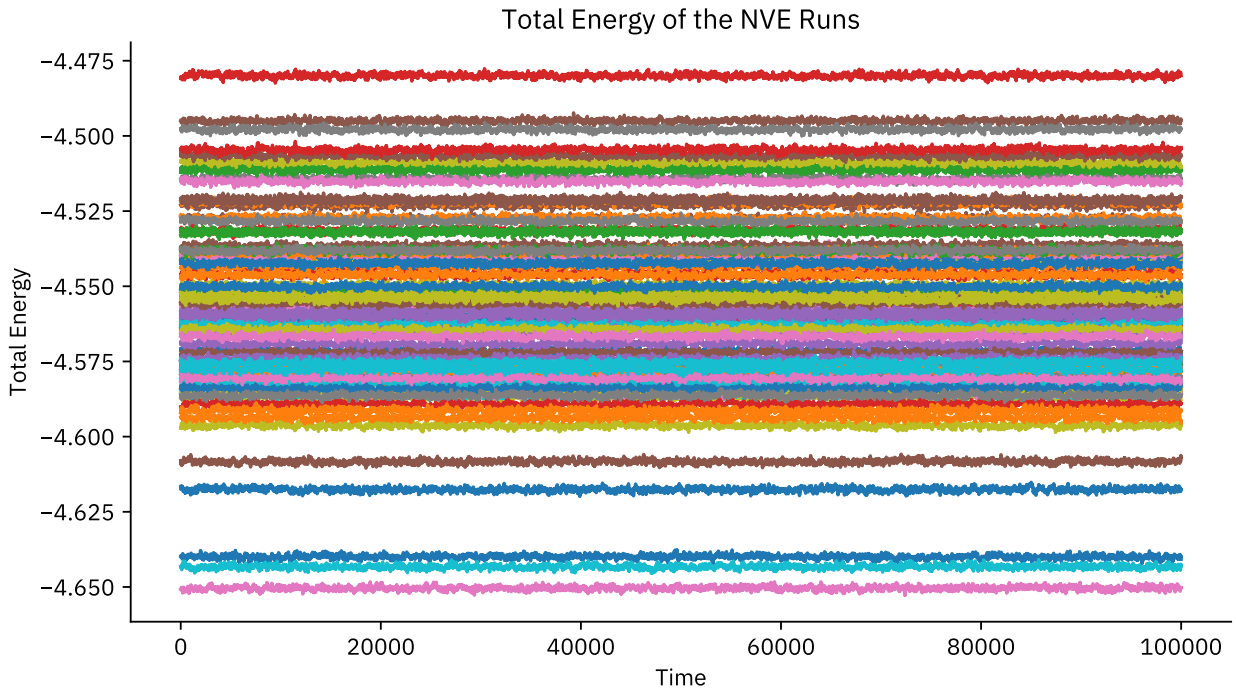
nificant amount of equilibration runs (3000 τ^* in total) to ensure that different production runs are uncorrelated. Even when these additional runs are included, the overall simulation time remains significantly lower than in the cited papers.

5.5.6 Validation of the Production Runs

To further establish that our NVE runs together represent the NVT ensemble, the following two cells perform additional validation checks.

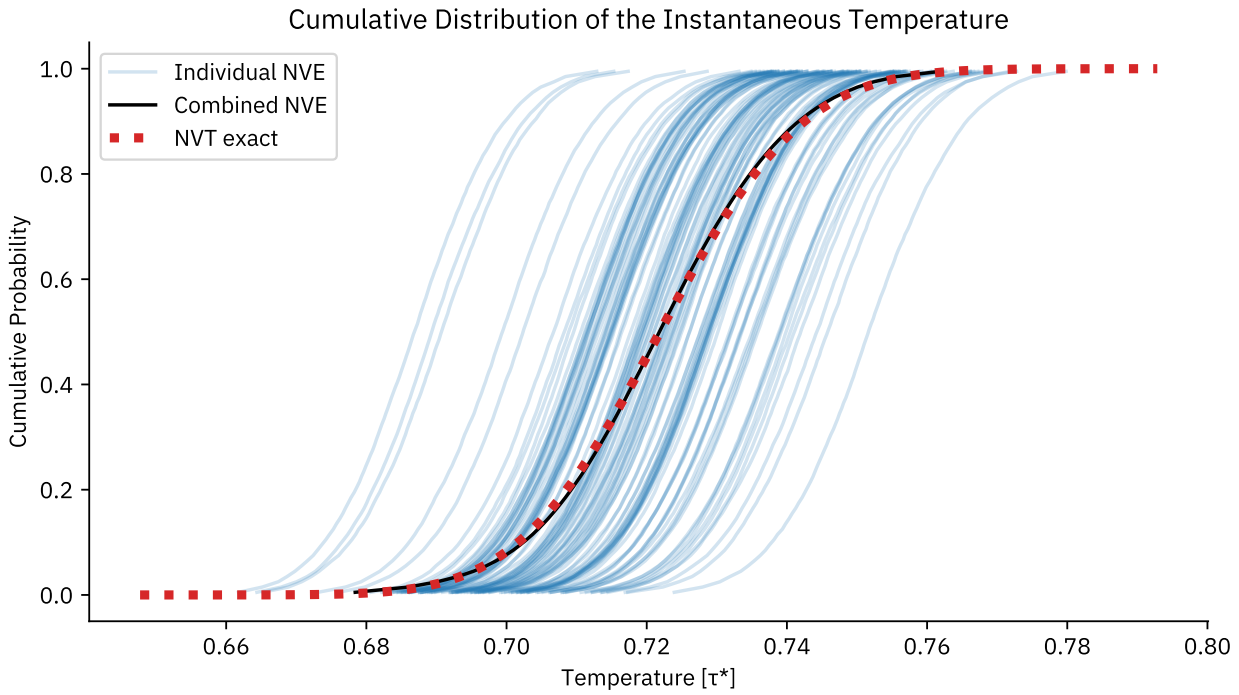
- A plot of the conserved quantity of the separate NVE runs, to detect any drift.
- The distribution of the instantaneous temperature, which should match the desired NVT distribution. For each individual NVE run and for the combined NVE runs, cumulative distributions are plotted. The function also plots the expected cumulative distribution of the NVT ensemble.

```
def plot_total_energy():
    # Load trajectory data.
    time = None
    energies = []
    for itraj in range(100):
        time_traj = []
        energies_traj = []
        for ipart in range(3):
            prod_dir = DATA_ROOT / f"replica_{itraj:04d}_part_{ipart:02d}/"
            data = np.loadtxt(prod_dir / "nve_thermo.txt")
            if time is None:
                time_traj.append(data[:, 0])
            energies_traj.append(data[:, 2:])
        if time is None:
            time = np.concatenate(time_traj)
        energies.append(energies_traj)
    energies = [np.concatenate(energy).T for energy in energies]

    # Plot the total energy of the NVE runs.
    plt.close("energyprod")
    _, ax = plt.subplots(num="energyprod")
    for kes, pes in energies:
        ax.plot(time, kes + pes)
    ax.set_xlabel("Time")
    ax.set_ylabel("Total Energy")
    ax.set_title("Total Energy of the NVE Runs")


plot_total_energy()
```



There is no noticeable drift in the total energy of the NVE runs. Apart from the usual (and acceptable) numerical noise, the total energy is conserved perfectly.

```
def validate_temperature():
    """Plot cumulative distributions of the instantaneous temperature."""
    # Load the configuration from the YAML file.
    with open(DATA_ROOT / "replica_0000_part_00/info.yaml") as fh:
        info = safe_load(fh)
    temp_d = info["temperature"]
    ndof = info["natom"] * 3 - 3

    # Load trajectory data.
    temps = []
    for itraj in range(100):
        temps.append([])
        for ipart in range(3):
            prod_dir = DATA_ROOT / f"replica_{itraj:04d}_part_{ipart:02d}/"
            temps[-1].append(np.loadtxt(prod_dir / "nve_thermo.txt")[:, 1])
    temps = [np.concatenate(temp).T for temp in temps]

    # Plot the instantaneous and desired temperature distribution.
    plt.close("tempprod")
    _, ax = plt.subplots(num="tempprod")
    plot_cumulative_temperature_histogram(ax, temps, temp_d, ndof, "τ*")


validate_temperature()
```



This plot offers detailed insight into NVE versus NVT temperature distributions:

- In the NVE ensemble, the temperature distribution is relatively narrow. Hence, using a single NVE run would not be representative of the temperature variance of the NVT ensemble.
- Some of the individual NVE runs have significantly lower ore higher temperatures than the average. If the transport property of interest has a nonlinear dependence on the temperature, this effect will lead to a shift in the estimated transport property, compared to using a single NVE run.

In the limit of macroscopic system sizes ($N \to \infty$), the NVE ensemble converges to the NVT ensemble. However, in simulations, one operates at finite system sizes, well below the thermodynamic limit.

5.5.7 Validation of the Independence of the Anistropic Contributions

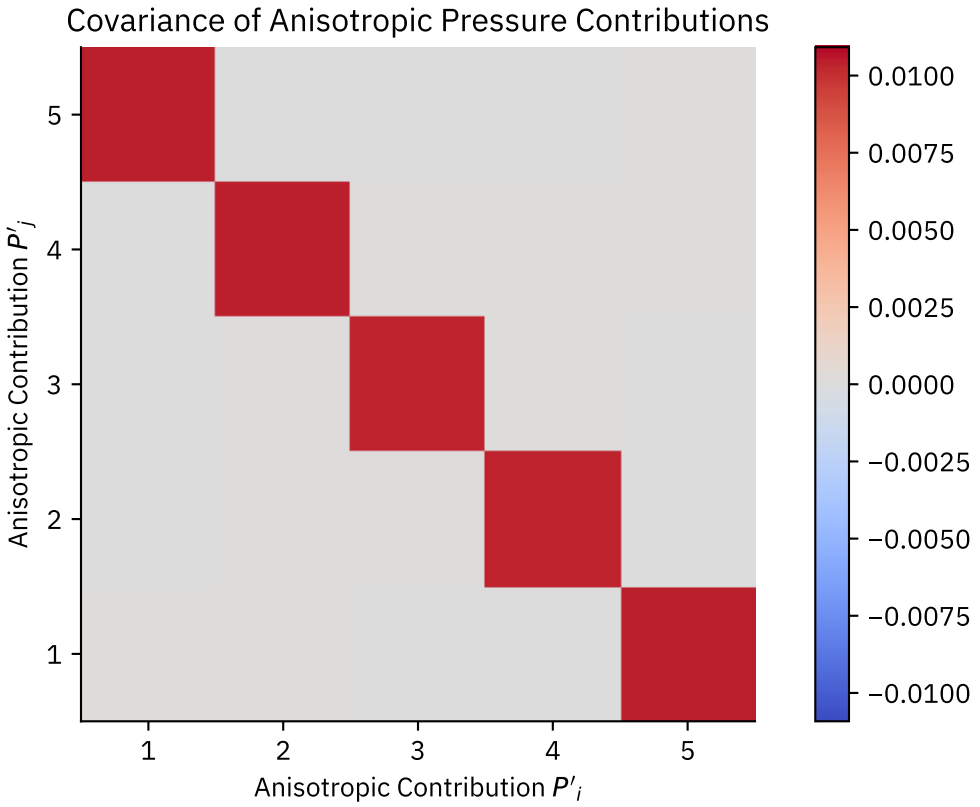
Here we validate numerically that the *five independent anisotropic contributions* to the pressure tensor are indeed statistically independent. The covariance matrix of the anisotropic contributions is computed and the off-diagonal elements are plotted.

```
def validate_independence(ntraj: int = 100):
    """Validate the independence of the anisotropic contributions."""
    # Load trajectory data.
    pcomps_full = []
    for itraj in range(ntraj):
        pcomps_full.append([])
        for ipart in range(3):
            prod_dir = DATA_ROOT / f"replica_{itraj:04d}_part_{ipart:02d}/"
            pcomps_full[-1].append(np.loadtxt(prod_dir / "nve_pressure_blav.txt")[:, 1:])
    pcomps_aniso = [get_indep_paniso(np.concatenate(parts).T) for parts in pcomps_full]

    # Compute the average of the covariance matrix over all NVE trajectories.
    cov = np.mean([np.cov(p, ddof=0) for p in pcomps_aniso], axis=0)
```

(continues on next page)

(continued from previous page)

```
    scale = abs(cov).max() * 1.05

    # Plot the covariance matrix.
    plt.close("covariance")
    _, ax = plt.subplots(num="covariance")
    im = ax.imshow(
        cov, cmap="coolwarm", vmin=-scale, vmax=scale, extent=[0.5, 5.5, 0.5, 5.5]
    )
    ax.set_title("Covariance of Anisotropic Pressure Contributions")
    ax.set_xlabel("Anisotropic Contribution $P'_i$")
    ax.set_ylabel("Anisotropic Contribution $P'_j$")
    plt.colorbar(im, ax=ax)


validate_independence()
```



The plot confirms that there is (at least visually) no sign of any statistical correlation between the anisotropic contributions. Note that one may perform more rigorous statistical tests to validate the independence of the five contributions. Here, we keep it simple for the sake of an intuitive demonstration.

5.5.8 Validation of the consistency of the Anisotropic Contributions

The following code cell shows that the five independent anisotropic contributions result in the same shear viscosity estimate, within the predicted uncertainties.

```
def validate_consistency():
    for i in range(5):
        result = analyze_production(3, select=i)
        eta = result.acint
        eta_std = result.acint_std
        print(f"Anisotropic contribution {i + 1}: η = {eta:.3f} ± {eta_std:.3f} η*")


validate_consistency()
```

```
Anisotropic contribution 1: η = 3.275 ± 0.059 η*
Anisotropic contribution 2: η = 3.194 ± 0.048 η*
Anisotropic contribution 3: η = 3.256 ± 0.055 η*
Anisotropic contribution 4: η = 3.183 ± 0.048 η*
Anisotropic contribution 5: η = 3.257 ± 0.055 η*
```

Note that one may perform more rigorous statistical tests to validate the consistency of the results. Here, we keep it simple for the sake of an intuitive demonstration.

5.5.9 Regression Tests

If you are experimenting with this notebook, you can ignore any exceptions below. The tests are only meant to pass for the notebook in its original form.

```
if abs(eta_production_init - 3.236) > 0.1:
    raise ValueError(f"wrong viscosity (production): {eta_production_init:.3e}")
if abs(eta_production_ext - 3.257) > 0.1:
    raise ValueError(f"wrong viscosity (production): {eta_production_ext:.3e}")
```

5.6 Bulk Viscosity of a Lennard-Jones Liquid Near the Triple Point (LAMMPS)

This example demonstrates how to compute the bulk viscosity of a Lennard-Jones liquid near its triple point using LAMMPS. It uses the same production runs and conventions as in the *Shear viscosity example*. The required theoretical background is explained the section *Bulk Viscosity*. In essence, it is computed in the same way as the shear viscosity, except that the isotropic pressure fluctuations are used as input.

> **Note**
>
> The results in this example were obtained using LAMMPS 29 Aug 2024 Update 3. Minor differences may arise when using a different version of LAMMPS, or even the same version compiled with a different compiler.

5.6.1 Library Imports and Matplotlib Configuration

```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
```

(continues on next page)

(continued from previous page)

```
from path import Path
from yaml import safe_load
from stacie import (
    UnitConfig,
    compute_spectrum,
    estimate_acint,
    LorentzModel,
    plot_fitted_spectrum,
    plot_extras,
)
```

```
mpl.rc_file("matplotlibrc")
%config InlineBackend.figure_formats = ["svg"]
```

```
# You normally do not need to change this path.
# It only needs to be overridden when building the documentation.
DATA_ROOT = Path(os.getenv("DATA_ROOT", "./")) / "lammps_lj3d/sims/"
```

5.6.2 Analysis of the Production Simulations

The following code cells define analysis functions used below.

- `get_piso`: Computes the isotropic pressure from the diagonal components of the time-dependent pressure tensor (P_{xx}, P_{yy}, and P_{zz}), as explained in the *bulk viscosity* theory section.
- `estimate_bulk_viscosity`: Computes the bulk viscosity, visualizes the results, and provides recommendations for data reduction (block averaging) and simulation time, as explained in the following two sections of the documentation:
  - *Integrated and Exponential Autocorrelation Time*
  - *Reducing Storage Requirements with Block Averages*

```
def estimate_bulk_viscosity(name, p_iso, av_temperature, volume, timestep):
    # Compute spectrum of the isotropic pressure fluctuations.
    # Note that the Boltzmann constant is 1 in reduced LJ units.
    uc = UnitConfig(
        acint_fmt=".3f",
        acint_symbol="η_b",
        acint_unit_str="η*",
        freq_unit_str="1/τ*",
        time_fmt=".3f",
        time_unit_str="τ*",
    )
    spectrum = compute_spectrum(
        p_iso,
        prefactors=volume / av_temperature,
        timestep=timestep,
        include_zero_freq=False,
    )

    # Estimate the bulk viscosity from the spectrum.
    result = estimate_acint(spectrum, LorentzModel(), verbose=True, uc=uc)
```

(continues on next page)

(continued from previous page)

```
    # Plot some basic analysis figures.
    plt.close(f"{name}_spectrum")
    _, ax = plt.subplots(num=f"{name}_spectrum")
    plot_fitted_spectrum(ax, uc, result)
    plt.close(f"{name}_extras")
    _, axs = plt.subplots(2, 2, num=f"{name}_extras")
    plot_extras(axs, uc, result)

    # Return the bulk viscosity
    return result.acint
```

Note

When computing bulk viscosity, the `include_zero_freq` argument in the `compute_spectrum` function must be set to `False`, as the average pressure is nonzero. This ensures the DC component is excluded from the spectrum. See the *bulk viscosity* theory section for more details.

```
def demo_production(npart: int = 3, ntraj: int = 100):
    """
    Perform the analysis of the production runs.

    Parameters
    ----------
    npart
        Number of parts in the production runs.
        For the initial production runs, this is 1.
    ntraj
        Number of trajectories in the production runs.

    Returns
    -------
    eta_bulk
        The estimated bulk viscosity.
    """
    # Load the configuration from the YAML file.
    with open(DATA_ROOT / "replica_0000_part_00/info.yaml") as fh:
        info = safe_load(fh)

    # Load trajectory data.
    thermos = []
    p_isos = []
    for itraj in range(ntraj):
        thermos.append([])
        p_isos.append([])
        for ipart in range(npart):
            prod_dir = DATA_ROOT / f"replica_{itraj:04d}_part_{ipart:02d}/"
            thermos[-1].append(np.loadtxt(prod_dir / "nve_thermo.txt"))
            # The average over columns 2, 3 and 4 of parts corresponds
            # to the time-dependent isotropic pressure.
            p_comps = np.loadtxt(prod_dir / "nve_pressure_blav.txt")
            p_isos[-1].append(p_comps[:, 1:4].mean(axis=1))
```

(continues on next page)

(continued from previous page)

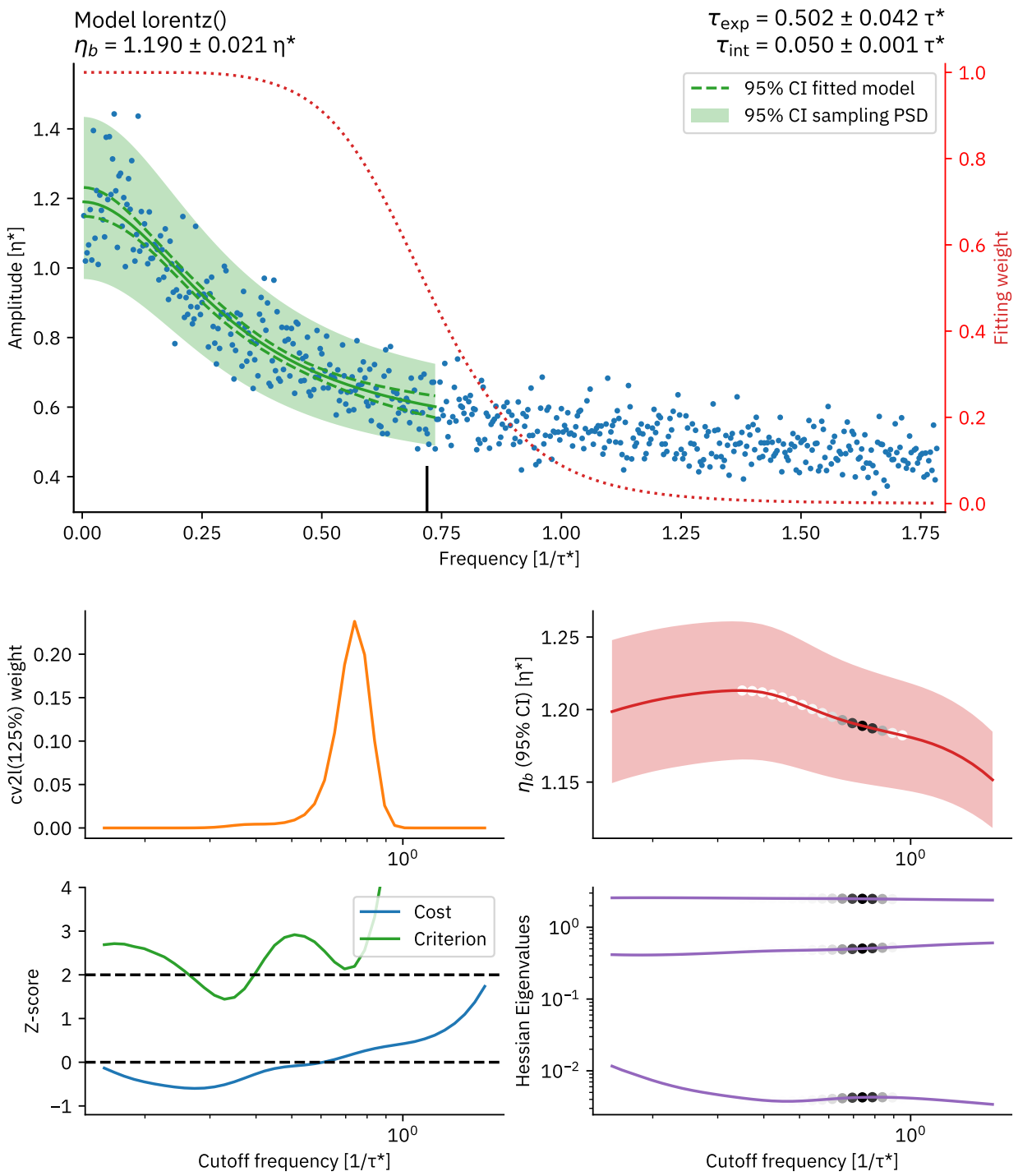
```
    thermos = [np.concatenate(parts).T for parts in thermos]
    p_iso = [np.concatenate(parts) for parts in p_isos]

    # Compute the average temperature
    av_temperature = np.mean([thermo[1] for thermo in thermos])

    # Compute the bulk viscosity
    return estimate_bulk_viscosity(
        f"part{npart}",
        p_iso,
        av_temperature,
        info["volume"],
        info["timestep"] * info["block_size"],
    )


eta_bulk_production = demo_production(3)
```

```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion  fcut [1/τ*]
---------  ----------  ----------
     15.0         inf    5.03e-02  (No correlation time estimate available.)
     16.0         inf    5.36e-02  (No correlation time estimate available.)
     17.1         inf    5.71e-02  (No correlation time estimate available.)
     18.2         inf    6.07e-02  (No correlation time estimate available.)
     19.4         inf    6.46e-02  (No correlation time estimate available.)
     20.7         inf    6.88e-02  (No correlation time estimate available.)
     22.0         inf    7.33e-02  (No correlation time estimate available.)
     23.5         inf    7.80e-02  (No correlation time estimate available.)
     25.0         inf    8.30e-02  (No correlation time estimate available.)
     26.7         inf    8.84e-02  (No correlation time estimate available.)
     28.4         inf    9.41e-02  (No correlation time estimate available.)
     30.3         inf    1.00e-01  (No correlation time estimate available.)
     32.3         inf    1.07e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-1.73755917e-03,  4.17952194e-01,  2.58378537e+00]))
     34.4         inf    1.13e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-1.33790325e-03,  4.17271199e-01,  2.58406670e+00]))
     36.7         inf    1.21e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-2.33922935e-04,  4.16524612e-01,  2.58370931e+00]))
     39.1         inf    1.29e-01  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     41.6         inf    1.37e-01  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     44.3         inf    1.46e-01  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     47.2        96.1    1.55e-01
     50.3        48.0    1.65e-01
     53.6        34.2    1.76e-01
     57.1        27.2    1.87e-01
     60.8        22.9    1.99e-01
     64.7        20.0    2.12e-01
     68.9        17.8    2.26e-01
     73.4        16.0    2.40e-01
     78.2        14.5    2.56e-01
     83.3        13.2    2.72e-01
     88.7        12.0    2.90e-01
```

(continues on next page)

(continued from previous page)

```
     94.4        11.0    3.08e-01
    100.5        10.3    3.28e-01
    107.1         9.7    3.49e-01
    114.0         9.4    3.72e-01
    121.4         9.3    3.96e-01
    129.2         9.3    4.22e-01
    137.6         9.2    4.49e-01
    146.5         9.0    4.78e-01
    156.0         8.6    5.09e-01
    166.1         8.1    5.41e-01
    176.8         7.5    5.76e-01
    188.3         6.8    6.13e-01
    200.4         6.1    6.53e-01
    213.4         5.6    6.95e-01
    227.2         5.3    7.40e-01
    241.9         5.5    7.88e-01
    257.5         6.2    8.38e-01
    274.2         7.6    8.92e-01
    291.9         9.7    9.50e-01
    310.7        12.9    1.01e+00
    330.8        17.2    1.08e+00
    352.2        22.9    1.15e+00
    374.9        30.3    1.22e+00
    399.1        39.8    1.30e+00
    424.9        51.8    1.38e+00
    452.3        66.6    1.47e+00
    481.5        84.6    1.57e+00
    512.6       106.9    1.67e+00
Cutoff criterion exceeds incumbent + margin: 5.3 + 100.0.

INPUT TIME SERIES
    Time step:                      0.030 τ*
    Simulation time:                300.000 τ*
    Maximum degrees of freedom:     200.0

MAIN RESULTS
    Autocorrelation integral:       1.190 ± 0.021 η*
    Integrated correlation time:    0.050 ± 0.001 τ*

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:     220.9 (ideally > 60)
    Regression cost Z-score:        0.2 (ideally < 2)
    Cutoff criterion Z-score:       2.5 (ideally < 2)

MODEL lorentz() | CUTOFF CRITERION cv2l(125%)
    Number of parameters:           3
    Average cutoff frequency:       7.20e-01 1/τ*
    Exponential correlation time:   0.502 ± 0.042 τ*

RECOMMENDED SIMULATION SETTINGS (EXPONENTIAL CORR. TIME)
    Block time:                     < 0.158 ± 0.024 τ*
    Simulation time:                > 31.517 ± 0.336 τ*
```



The cutoff criterion Z-score is relatively high, around 2. This suggests that the fits on the two halves deviate more from each other than what would be expected from the *uncertainty* of the spectrum. There are multiple potential explanations for this observation:

- One potential explanation is that the isotropic pressure fluctuations are not perfectly Gaussian. This is expected for a Lennard-Jones fluid, as expansion of the system will result in slightly lower restoring forces than compression. Such a slightly non-Gaussian distribution of the pressure fluctuations can result in a distribution of spectral data that deviates from the Gamma distribution employed by STACIE.
- Another potential cause is that there is not yet sufficient data to fix the cutoff frequency. This

can be addressed by generating more trajectory data, which will make it easier to determine the suitable range of cutoff frequencies. We have not further expanded the production runs in this example, to keep the computational cost low. Furthermore, as shown in the comparison below, we already obtained a good agreement with the literature results and relatively small uncertainties.

5.6.3 Comparison to Literature Results

Computational estimates of the bulk viscosity of a Lennard-Jones fluid can be found in [MLK04b]. Since the simulation settings ($r_{\text{cut}}^* = 2.5$, $N = 1372$, $T^* = 0.722$ and $\rho^* = 0.8442$) are identical to those used in this notebook, the reported values should be directly comparable.

Method	Simulation time [τ^*]	Bulk viscosity [$\eta_b{}^*$]	Reference
EMD NVE (STACIE)	3600	1.182 ± 0.071	(here) initial
EMD NVE (STACIE)	10800	1.159 ± 0.031	(here) extension 1
EMD NVE (STACIE)	30000	1.190 ± 0.021	(here) extension 2
EMD NVE (Helfand-Einstein)	300000	1.186 ± 0.084	[MLK04b]

This comparison demonstrates that STACIE accurately reproduces bulk viscosity results while achieving lower statistical uncertainty with significantly less data than existing methods.

Note that the results for only the initial NVE production run are not included because the sanity checks indicated that the data was not sufficient.

5.6.4 Regression Tests

If you are experimenting with this notebook, you can ignore any exceptions below. The tests are only meant to pass for the notebook in its original form.

```
if abs(eta_bulk_production - 1.195) > 0.1:
    raise ValueError(f"wrong viscosity (production): {eta_bulk_production:.3e}")
```

5.7 Thermal Conductivity of a Lennard-Jones Liquid Near the Triple Point (LAMMPS)

This example shows how to derive the thermal conductivity using heat flux data from a LAMMPS simulation. It uses the same production runs and conventions as in the *Shear viscosity example*. The required theoretical background is explained the section *Thermal Conductivity*.

Warning

A Lennard-Jones system only exhibits pairwise interactions, for which the LAMMPS command `compute/heat flux` produces valid results. For systems with three- or higher-body interactions, one cannot simply use the same command. Consult the theory section on *thermal conductivity* for more background.

Note

The results in this example were obtained using LAMMPS 29 Aug 2024 Update 3. Minor differences may arise when using a different version of LAMMPS, or even the same version compiled with a different compiler.

5.7.1 Library Imports and Configuration

```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from path import Path
from yaml import safe_load
from stacie import (
    UnitConfig,
    compute_spectrum,
    estimate_acint,
    LorentzModel,
    plot_fitted_spectrum,
    plot_extras,
)
```

```
mpl.rc_file("matplotlibrc")
%config InlineBackend.figure_formats = ["svg"]
```

```
# You normally do not need to change this path.
# It only needs to be overridden when building the documentation.
DATA_ROOT = Path(os.getenv("DATA_ROOT", "./")) / "lammps_lj3d/sims/"
```

5.7.2 Analysis of the Production Simulations

The function `estimate_thermal_conductivity` implements the analysis, assuming the data have been read from the LAMMPS outputs and are passed as function arguments.

```
def estimate_thermal_conductivity(name, jcomps, av_temperature, volume, timestep):
    # Create the spectrum of the heat flux fluctuations.
    # Note that the Boltzmann constant is 1 in reduced LJ units.
    uc = UnitConfig(
        acint_fmt=".3f",
        acint_symbol="κ",
        acint_unit_str="κ*",
        freq_unit_str="1/τ*",
        time_fmt=".3f",
        time_unit_str="τ*",
    )
    spectrum = compute_spectrum(
        jcomps,
        prefactors=1 / (volume * av_temperature**2),
        timestep=timestep,
    )

    # Estimate the thermal conductivity from the spectrum.
    result = estimate_acint(spectrum, LorentzModel(), verbose=True, uc=uc)
```

(continues on next page)

(continued from previous page)

```
    # Plot some basic analysis figures.
    plt.close(f"{name}_spectrum")
    _, ax = plt.subplots(num=f"{name}_spectrum")
    plot_fitted_spectrum(ax, uc, result)
    plt.close(f"{name}_extras")
    _, axs = plt.subplots(2, 2, num=f"{name}_extras")
    plot_extras(axs, uc, result)

    # Return the thermal conductivity
    return result.acint
```

The following cell implements the analysis of the production simulations.

```
def demo_production(npart: int = 3, ntraj: int = 100):
    """
    Perform the analysis of the production runs.

    Parameters
    ----------
    npart
        Number of parts in the production runs.
        For the initial production runs, this is 1.
    ntraj
        Number of trajectories in the production runs.

    Returns
    -------
    kappa
        The estimated thermal conductivity.
    """
    # Load the configuration from the YAML file.
    with open(DATA_ROOT / "replica_0000_part_00/info.yaml") as fh:
        info = safe_load(fh)

    # Load trajectory data, without hardcoding the number of runs and parts.
    thermos = []
    heatfluxes = []
    for itraj in range(ntraj):
        thermos.append([])
        heatfluxes.append([])
        for ipart in range(npart):
            prod_dir = DATA_ROOT / f"replica_{itraj:04d}_part_{ipart:02d}/"
            thermos[-1].append(np.loadtxt(prod_dir / "nve_thermo.txt"))
            heatfluxes[-1].append(np.loadtxt(prod_dir / "nve_heatflux_blav.txt"))
    thermos = [np.concatenate(parts).T for parts in thermos]
    heatfluxes = [np.concatenate(parts).T for parts in heatfluxes]

    # Compute the average temperature.
    av_temperature = np.mean([thermo[1].mean() for thermo in thermos])

    # Compute the thermal conductivity.
    # Note that the last three columns are not used in the analysis.
    # According to the LAMMPS documentation, the last three columns
```

(continues on next page)

(continued from previous page)

```
    # only contain the convective contribution to the heat flux.
    # See https://docs.lammps.org/compute_heat_flux.html
    jcomps = np.concatenate([heatflux[1:4] for heatflux in heatfluxes])
    return estimate_thermal_conductivity(
        f"part{npart}",
        jcomps,
        av_temperature,
        info["volume"],
        info["timestep"] * info["block_size"],
    )


kappa_production = demo_production(3)
```

```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion  fcut [1/τ*]
---------  ----------  ----------
     15.0         inf    4.71e-02  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     15.9         inf    5.01e-02  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     16.9       130.9    5.34e-02
     18.0       124.3    5.68e-02
     19.1         inf    6.05e-02  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-6.75412020e-06,  4.43842256e-01,  2.55616450e+00]))
     20.3         inf    6.44e-02  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-3.50407301e-06,  4.43068635e-01,  2.55693487e+00]))
     21.6         inf    6.85e-02  (No correlation time estimate available.)
     23.0         inf    7.30e-02  (No correlation time estimate available.)
     24.4         inf    7.77e-02  (No correlation time estimate available.)
     25.9         inf    8.27e-02  (No correlation time estimate available.)
     27.6         inf    8.80e-02  (No correlation time estimate available.)
     29.3         inf    9.37e-02  (No correlation time estimate available.)
     31.2         inf    9.97e-02  (No correlation time estimate available.)
     33.2         inf    1.06e-01  (No correlation time estimate available.)
     35.3         inf    1.13e-01  (No correlation time estimate available.)
     37.5         inf    1.20e-01  (No correlation time estimate available.)
     39.9         inf    1.28e-01  (No correlation time estimate available.)
     42.4         inf    1.36e-01  (No correlation time estimate available.)
     45.1         inf    1.45e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-1.00541799e-04,  4.38053694e-01,  2.56204685e+00]))
     48.0         inf    1.54e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-1.17919538e-04,  4.37106550e-01,  2.56301137e+00]))
     51.1         inf    1.64e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-9.50267075e-05,  4.36440454e-01,  2.56365457e+00]))
     54.4         inf    1.75e-01  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     57.8         inf    1.86e-01  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     61.5         inf    1.98e-01  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     65.5         inf    2.11e-01  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     69.7         inf    2.25e-01  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     74.1         inf    2.39e-01  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     78.9         inf    2.55e-01  (rel.err. tau_exp > 1.0e+02 x rel.err. ac integral)
     83.9        93.2    2.71e-01
     89.3        80.6    2.89e-01
     95.0        72.2    3.07e-01
```

(continues on next page)

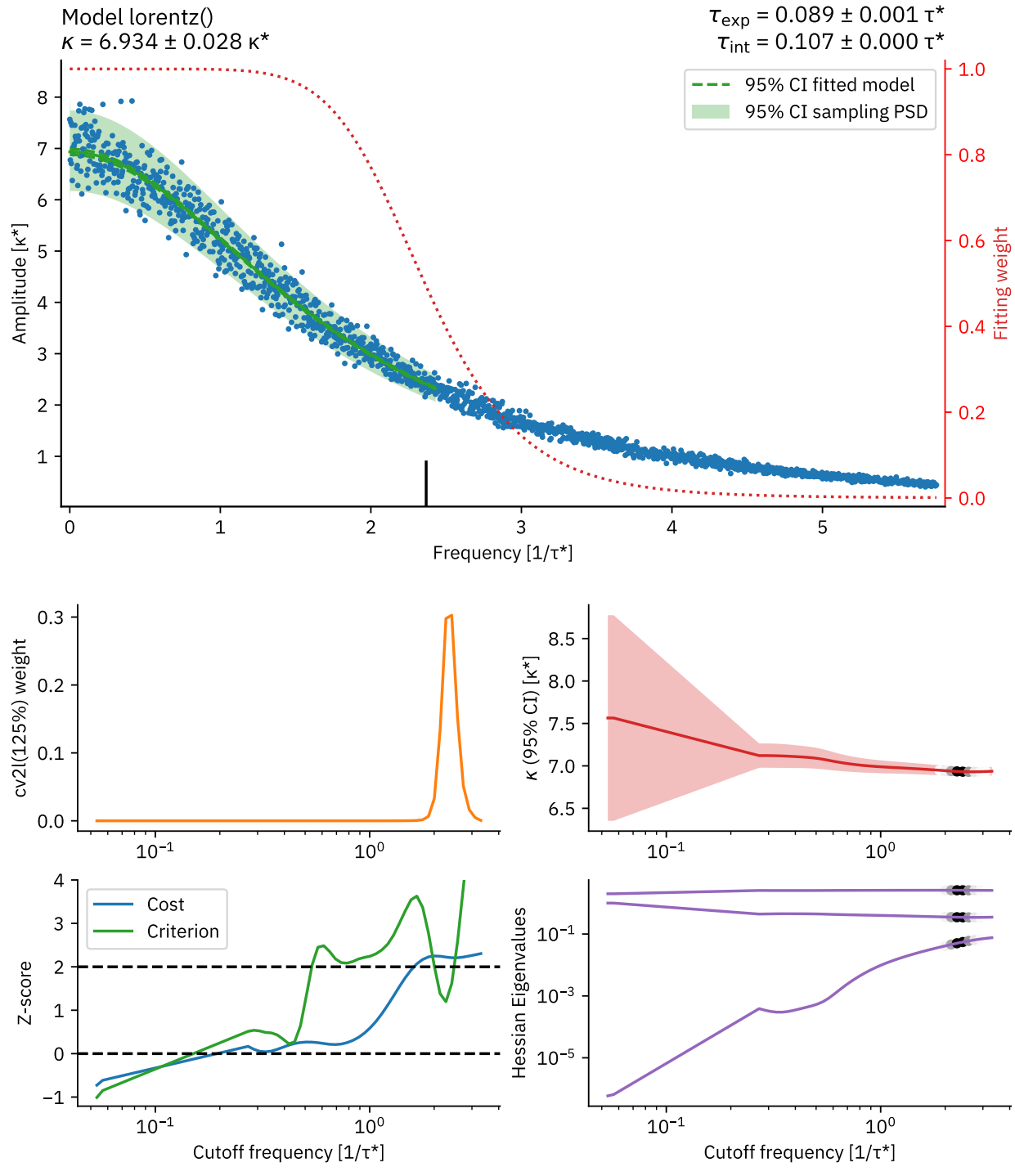
(continued from previous page)

```
    101.1        66.4    3.27e-01
    107.6        61.5    3.48e-01
    114.5        57.6    3.70e-01
    121.9        53.4    3.94e-01
    129.7        49.5    4.20e-01
    138.0        45.8    4.47e-01
    146.9        43.2    4.76e-01
    156.3        41.8    5.06e-01
    166.4        40.7    5.39e-01
    177.1        39.4    5.74e-01
    188.5        36.9    6.11e-01
    200.6        33.5    6.50e-01
    213.5        29.8    6.92e-01
    227.2        26.0    7.37e-01
    241.9        22.3    7.84e-01
    257.4        18.8    8.35e-01
    274.0        15.7    8.89e-01
    291.6        12.8    9.46e-01
    310.4        10.3    1.01e+00
    330.4         8.1    1.07e+00
    351.7         6.3    1.14e+00
    374.3         4.8    1.21e+00
    398.4         3.6    1.29e+00
    424.1         2.7    1.38e+00
    451.4         1.9    1.47e+00
    480.5         1.2    1.56e+00
    511.5         0.4    1.66e+00
    544.4        -0.8    1.77e+00
    579.5        -2.2    1.88e+00
    616.9        -3.8    2.00e+00
    656.6        -5.2    2.13e+00
    698.9        -6.0    2.27e+00
    744.0        -6.0    2.42e+00
    791.9        -5.4    2.57e+00
    843.0        -4.3    2.74e+00
    897.3        -3.1    2.91e+00
    955.1        -1.9    3.10e+00
   1016.7         0.2    3.30e+00
Reached the maximum number of effective points (1000).

INPUT TIME SERIES
    Time step:                     0.030 τ*
    Simulation time:               300.000 τ*
    Maximum degrees of freedom:    600.0

MAIN RESULTS
    Autocorrelation integral:      6.934 ± 0.028 κ*
    Integrated correlation time:   0.107 ± 0.000 τ*

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:    729.1 (ideally > 60)
    Regression cost Z-score:       2.2 (ideally < 2)
    Cutoff criterion Z-score:      1.8 (ideally < 2)
```

(continues on next page)

(continued from previous page)

```
MODEL lorentz() | CUTOFF CRITERION cv2l(125%)
    Number of parameters:          3
    Average cutoff frequency:      2.37e+00 1/τ*
    Exponential correlation time:  0.089 ± 0.001 τ*

RECOMMENDED SIMULATION SETTINGS (EXPONENTIAL CORR. TIME)
    Block time:                    < 0.028 ± 0.001 τ*
    Simulation time:               > 5.611 ± 0.007 τ*
```



The exponential correlation time of the heat flux tensor fluctuations is about four times shorter than that of the pressure tensor fluctuations. This means that the thermal conductivity is a bit easier to compute than the viscosity. Note that the selected block size is still compatible with this shorter time scale.

Similarly to the bulk viscosity, the Z-scores are clearly positive. This could be for the same reasons as in the bulk viscosity example. In addition, the block size of 0.03 τ^* is slightly larger than the recommended 0.028 τ^*, meaning that the spectrum might be perturbed by (very) small aliasing effects that could distort the fit.

5.7.3 Comparison to Literature Results

A detailed literature survey of computational estimates of the thermal conductivity of a Lennard-Jones fluid can be found in [VSG07b]. Viscardi also computes new estimates, one of which is included in the table below. This value can be directly comparable to the current notebook, because the settings are identical ($r^*_{\text{cut}} = 2.5$, $N = 1372$, $T^* = 0.722$ and $\rho^* = 0.8442$).

Method	Simulation time [τ^*]	Thermal conductivity [κ^*]	Reference
EMD NVE (STACIE)	3600	6.852 ± 0.079	(here) initial
EMD NVE (STACIE)	10800	6.968 ± 0.045	(here) extension 1
EMD NVE (STACIE)	30000	6.934 ± 0.028	(here) extension 2
EMD NVE (Helfand-moment)	600000	6.946 ± 0.12	[VSG07b]

This small comparison confirms that STACIE can reproduce a well-known thermal conductivity result, with small error bars, while using much less trajectory data than existing methods.

5.7.4 Regression Tests

If you are experimenting with this notebook, you can ignore any exceptions below. The tests are only meant to pass for the notebook in its original form.

```
if abs(kappa_production - 6.953) > 0.2:
    raise ValueError(f"wrong thermal conductivity (production): {kappa_production:.3e}")
```

5.8 Ionic Electrical Conductivity of Molten Sodium Chloride at 1100 K (OpenMM)

> **Warning**
>
> This example notebook is work in progress. There are still some issues with the MD results obtained with OpenMM, which are discussed in the notebook.

This notebook shows how to post-process trajectories from OpenMM simulations to calculate the ionic electrical conductivity. The OpenMM trajectories are converted to NPZ files within the Jupyter Notebooks of the simulation, making the approach here easily adaptable to other codes

or physical systems. All OpenMM simulation notebooks can be found in the directory `docs/data/openmm_salt` in STACIE's source repository. The required theoretical background is explained the *Ionic Electrical Conductivity* section.

The MD simulations are performed using the Born-Huggins-Mayer-Tosi-Fumi potential, which is a popular choice for molten salts. [TF64] This potential does not use mixing rules and it is not natively implemented in OpenMM, but it can be incorporated using the `CustomNonbondedForce` and some creativity, see `docs/data/openmm_salt/bhmft.py` in the Git repository. The molten salt was simulated with a 3D periodic box of 1728 ions (864 Na^+ and 864 Cl^-). The time step in all simulations was 5 fs.

Following the *Recommendations for MD Simulations*, an initial block size of 10 steps (50 fs) was used. Because there is little prior knowledge on the structure of the spectrum, the exponential polynomial model (ExpPoly) with degrees $S = \{0, 1\}$ was used initially, i.e. with $P = 2$ parameters. As explained in the section on *block averages*, $400P$ blocks were collected in the initial production runs, amounting to 8000 steps (40 ps) of simulation time.

In total 100 NVE production runs were performed. For each run, the system was first equilibrated in the NVT and later NPT ensemble. According to the section *How to Prepare Sufficient Inputs for STACIE?*, 100 runs should be sufficient to obtain a relative error on the ionic conductivity of about 1%:

$$\epsilon_{\text{rel}} \approx \frac{1}{\sqrt{20PM}} \approx 0.0091$$

where P is the number of parameters in the model and $M = 100 \times 3$ is the number of independent input sequences. (100 trajectories, 3 Cartesian components of the charge current per trajectory)

> **Note**
>
> The results in this example were obtained using OpenMM 8.2.0. Minor differences may arise when using a different version of OpenMM, or even the same version compiled with a different compiler.

5.8.1 Library Imports and Configuration

```
import os
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
from path import Path
import scipy.constants as sc
from scipy.stats import chi2
from stacie import (
    ExpPolyModel,
    PadeModel,
    UnitConfig,
    compute_spectrum,
    estimate_acint,
    plot_fitted_spectrum,
    plot_extras,
)
from utils import plot_instantaneous_percentiles, plot_cumulative_temperature_histogram
```

```
mpl.rc_file("matplotlibrc")
%config InlineBackend.figure_formats = ["svg"]
```

```
# You normally do not need to change this path.
# It only needs to be overridden when building the documentation.
DATA_ROOT = Path(os.getenv("DATA_ROOT", "./")) / "openmm_salt/output/"
```

5.8.2 Analysis of the NpT Equilibration Runs

To validate that the equilibration runs have reached to proper temperature distribution, the following cell implements a plot of the percentiles (over the 100 trajectories) of a thermodynamic quantity (temperature or volume).

```
def plot_openmm_percentiles(
    ensemble: str,
    field: str,
    unitstr: str,
    unit: float = 1,
    npart: int = 1,
    ntraj: int = 100,
    expected: None = None,
    ymin: float | None = None,
    ymax: float | None = None,
):
    """Plot the temperature of the NpT equilibration runs."""
    time = None
    natom = None
    sequences = []
    time = None
    for itraj in range(ntraj):
        row = []
        if itraj == 0:
            time = []
        for ipart in range(npart):
            path_npz = DATA_ROOT / f"sim{itraj:04d}_part{ipart:02d}_{ensemble}_traj.npz"
            if not path_npz.exists():
                print(f"File {path_npz} not found, skipping.")
                row = None
                break
            data = np.load(path_npz)
            natom = len(data["atnums"])
            if itraj == 0:
                time.append(data["time"])
            row.append(data[field])
        if row is None:
            continue
        if itraj == 0:
            time = np.concatenate(time)
        row = np.concatenate(row)
        sequences.append(row)
    sequences = np.array(sequences)

    percents = np.array([95, 80, 50, 20, 5])
    if field == "temperature":
```

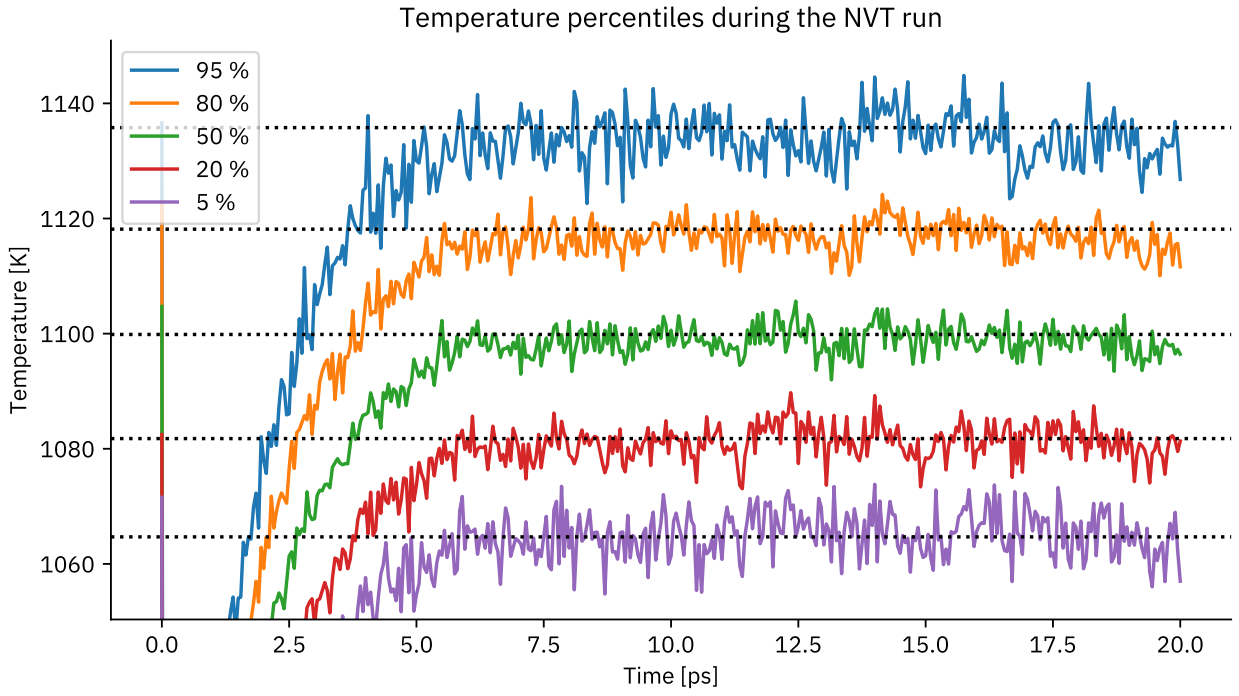
(continues on next page)

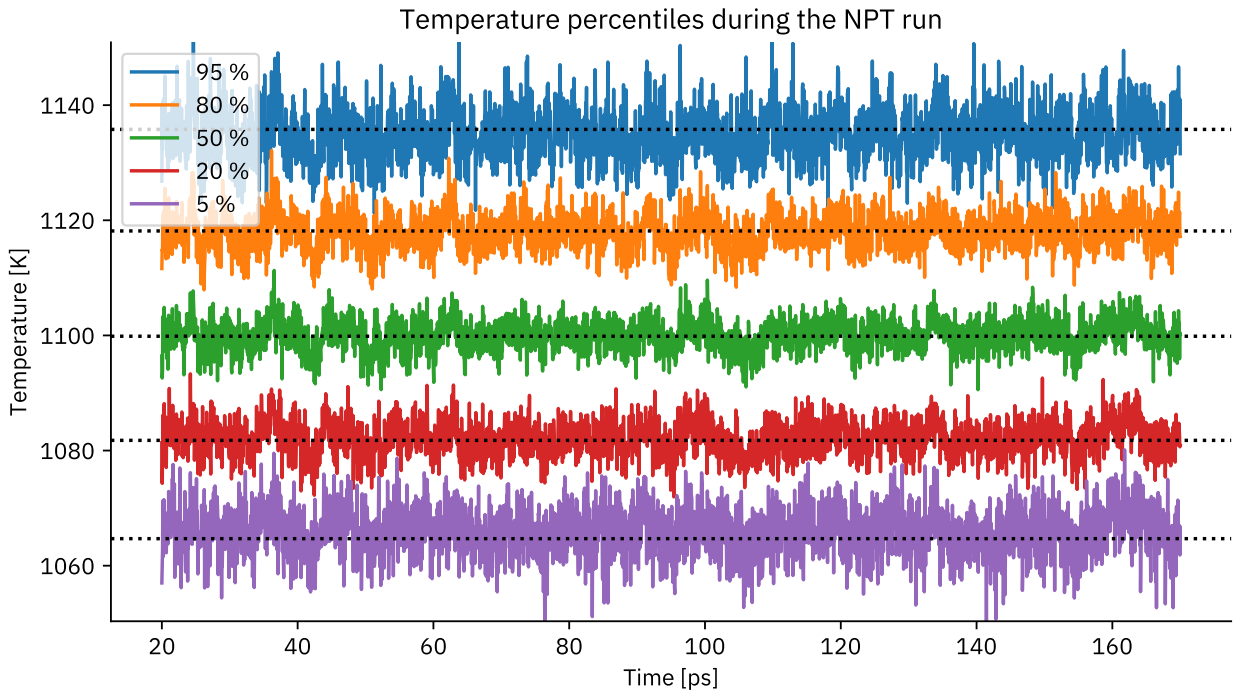
(continued from previous page)

```
        temp_d = 1100
        ndof = 3 * natom - 3
        expected = chi2.ppf(percents / 100, ndof) * temp_d / ndof
        ymin = chi2.ppf(0.01, ndof) * temp_d / ndof
        ymax = chi2.ppf(0.99, ndof) * temp_d / ndof
    else:
        expected = None
    time_unit = 1e-12
    num = f"percentiles_{field}_{ensemble}"
    plt.close(num)
    _, ax = plt.subplots(num=num)
    plot_instantaneous_percentiles(
        ax,
        time / time_unit,
        sequences / unit,
        percents,
        None if expected is None else expected / unit,
        ymin,
        ymax,
    )
    ax.set_title(f"{field.title()} percentiles during the {ensemble.upper()} run")
    ax.set_xlabel("Time [ps]")
    ax.set_ylabel(f"{field.title()} [{unitstr}]")
```

The following cell plots the temperature percentiles for the NVT and NPT equilibration runs.

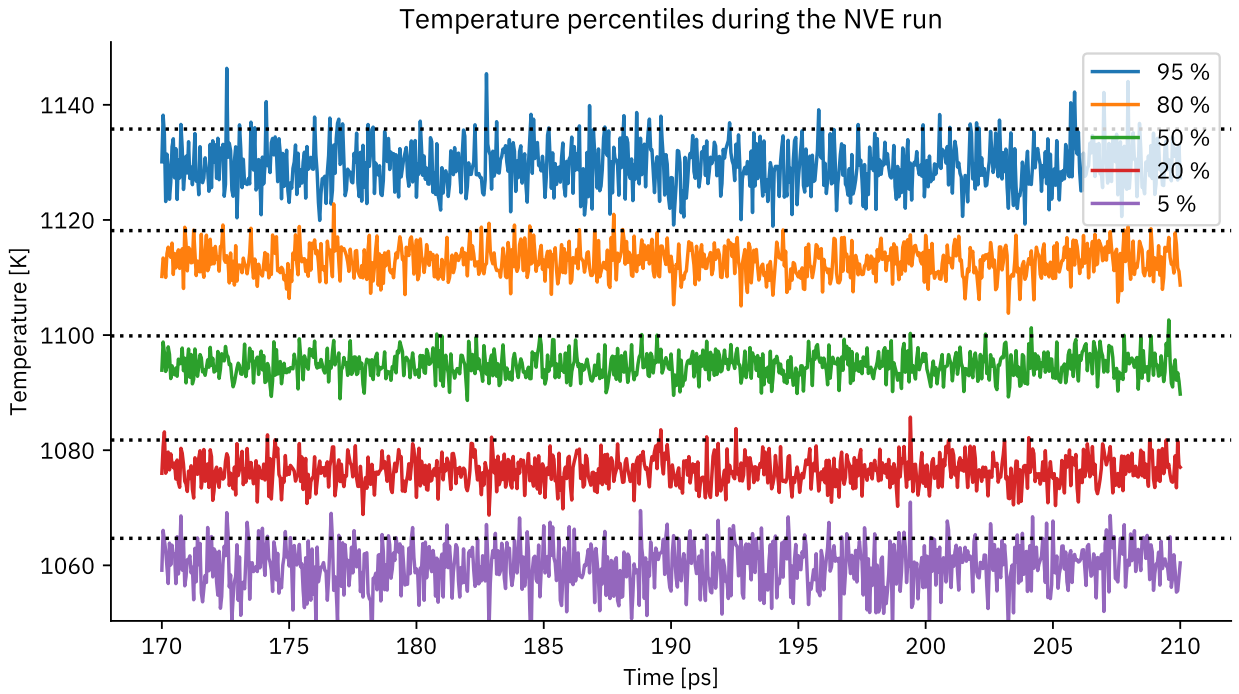
```
plot_openmm_percentiles("nvt", "temperature", "K")
plot_openmm_percentiles("npt", "temperature", "K")
```





The percentiles look good for the equilibration runs: they quickly reach their theoretical values (black dotted lines) and then fluctuate around them. The following cell plots the temperature percentiles for the initial NVE production runs.

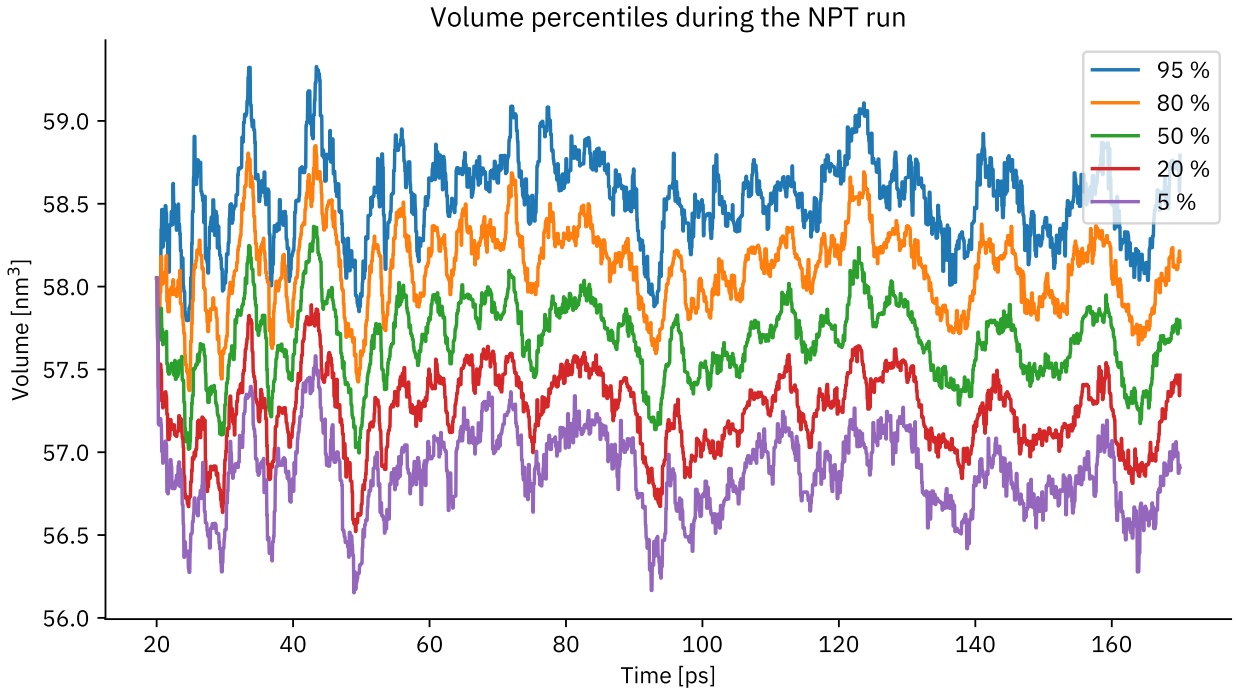
```
plot_openmm_percentiles("nve", "temperature", "K")
```



This is clearly not the correct temperature distribution! There is a known problem with restart files in the NVE ensemble in OpenMM. Due to a bug, it tends to lower the temperature of the system. More details on this issue can be found here: https://github.com/openmm/openmm/issues/4948

The following cell plots the percentiles of the volume (over the 100 trajectories), which we can only use to validate that the volume distribution converges. However, we cannot trivially compare these percentiles to an expected distribution.

```
plot_openmm_percentiles("npt", "volume", "nm$^3$", unit=1e-27, expected="last")
```



This plot is not completely satisfactory either, as it suggests that the volume fluctuations of the 100 runs exhibit synchronized fluctuations, while they should be independent. (They use different random seeds for their MC Barostat.)

5.8.3 Reusable Code for the Analysis of the Production Runs

The `analyze` function takes a few parameters to apply the same analysis with STACIE to different inputs (initial and extended production runs). After the analysis, it generates screen output and figures as discussed in the *minimal example*.

```
BOLTZMANN_CONSTANT = sc.value("Boltzmann constant")  # J/K


def analyze(model, npart: int = 1, ntraj: int = 100) -> float:
    """Analyze MD trajectories to compute the ionic conductivity.

    Parameters
    ----------
    model
        The model fitted to the spectrum.
    npart
        The number of parts in the simulation to load.
        The default value of 1 corresponds to only loading the initial production runs.

    Returns
    -------
```

(continues on next page)

(continued from previous page)

```
    acint
        The estimated ionic conductivity, mainly used for regression testing.
    """
    # Get the time step from the first NPZ file.
    time = np.load(DATA_ROOT / "sim0000_part00_nve_traj.npz")["time"]
    timestep = time[1] - time[0]

    def iter_sequences():
        """A generator that only loads one MD trajectory at a time in memory."""
        for itraj in range(ntraj):
            paths_npz = [
                DATA_ROOT / f"sim{itraj:04d}_part{ipart:02d}_nve_traj.npz"
                for ipart in range(npart)
            ]
            if not all(path_npz.exists() for path_npz in paths_npz):
                print(f"Some of {paths_npz} not found, skipping.")
                continue
            dipole = []
            for path_npz in paths_npz:
                data = np.load(path_npz)
                dipole.append(data["dipole"])
            dipole = np.concatenate(dipole, axis=1)
            data = np.load(paths_npz[0])
            prefactor = 1.0 / (
                data["volume"][0] * data["temperature"].mean() * BOLTZMANN_CONSTANT
            )
            # The finite difference is equivalent to a block-averaged charge current.
            current = np.diff(dipole, axis=1) / timestep
            yield prefactor, current

    # Configure units for output
    uc = UnitConfig(
        acint_symbol=r"\sigma",
        acint_unit_str=r"S/m",
        acint_fmt=".1f",
        time_unit=1e-15,
        time_unit_str="fs",
        time_fmt=".3f",
        freq_unit=1e12,
        freq_unit_str="THz",
    )

    # Perform the analysis with STACIE
    spectrum = compute_spectrum(
        iter_sequences(),
        timestep=timestep,
        prefactors=None,
        include_zero_freq=False,
    )
    result = estimate_acint(spectrum, model, verbose=True, uc=uc)

    # Plot some basic analysis figures.
    prefix = "conductivity"
    plt.close(f"{prefix}_spectrum")
```

(continues on next page)

(continued from previous page)

```
    _, ax = plt.subplots(num=f"{prefix}_fitted")
    plot_fitted_spectrum(ax, uc, result)
    plt.close(f"{prefix}_extras")
    _, axs = plt.subplots(2, 2, num=f"{prefix}_extras")
    plot_extras(axs, uc, result)

    # Return the ionic conductivity.
    return result.acint
```

5.8.4 Analysis of the Initial Production Simulation

The following cell computes the ionic conductivity of the molten salt at 1100 K, from the initial production runs (8000 steps each).

```
conductivity_1_01 = analyze(ExpPolyModel([0, 1]))
```

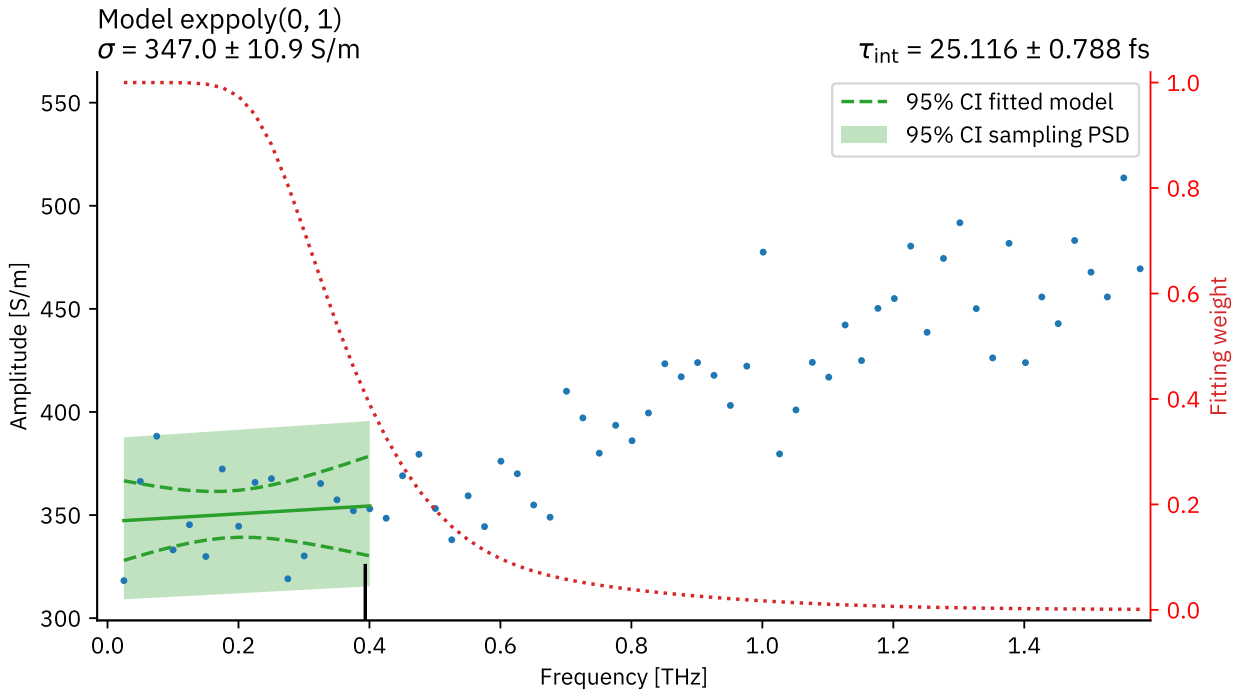
```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion  fcut [THz]
---------  ----------  ----------
     10.0       -30.5    2.56e-01
     10.7       -30.6    2.73e-01
     11.4       -30.7    2.90e-01
     12.2       -30.7    3.09e-01
     13.0       -30.7    3.29e-01
     13.8       -30.7    3.50e-01
     14.8       -30.6    3.73e-01
     15.8       -30.6    3.97e-01
     16.8       -30.5    4.22e-01
     17.9       -30.4    4.50e-01
     19.1       -30.2    4.79e-01
     20.4       -29.9    5.09e-01
     21.7       -29.4    5.42e-01
     23.2       -28.8    5.77e-01
     24.7       -28.2    6.14e-01
     26.3       -27.7    6.54e-01
     28.0       -27.5    6.96e-01
     29.9       -27.5    7.41e-01
     31.8       -27.8    7.89e-01
     33.9       -28.1    8.40e-01
     36.1       -28.3    8.94e-01
     38.5       -28.3    9.52e-01
     41.0       -28.0    1.01e+00
     43.7       -27.5    1.08e+00
     46.5       -26.8    1.15e+00
     49.6       -26.1    1.22e+00
     52.8       -25.3    1.30e+00
     56.2       -24.4    1.38e+00
     59.9       -23.3    1.47e+00
     63.8       -21.9    1.57e+00
     67.9       -19.9    1.67e+00
     72.4       -17.3    1.78e+00
     77.1       -13.6    1.89e+00
     82.1        -8.3    2.01e+00
```

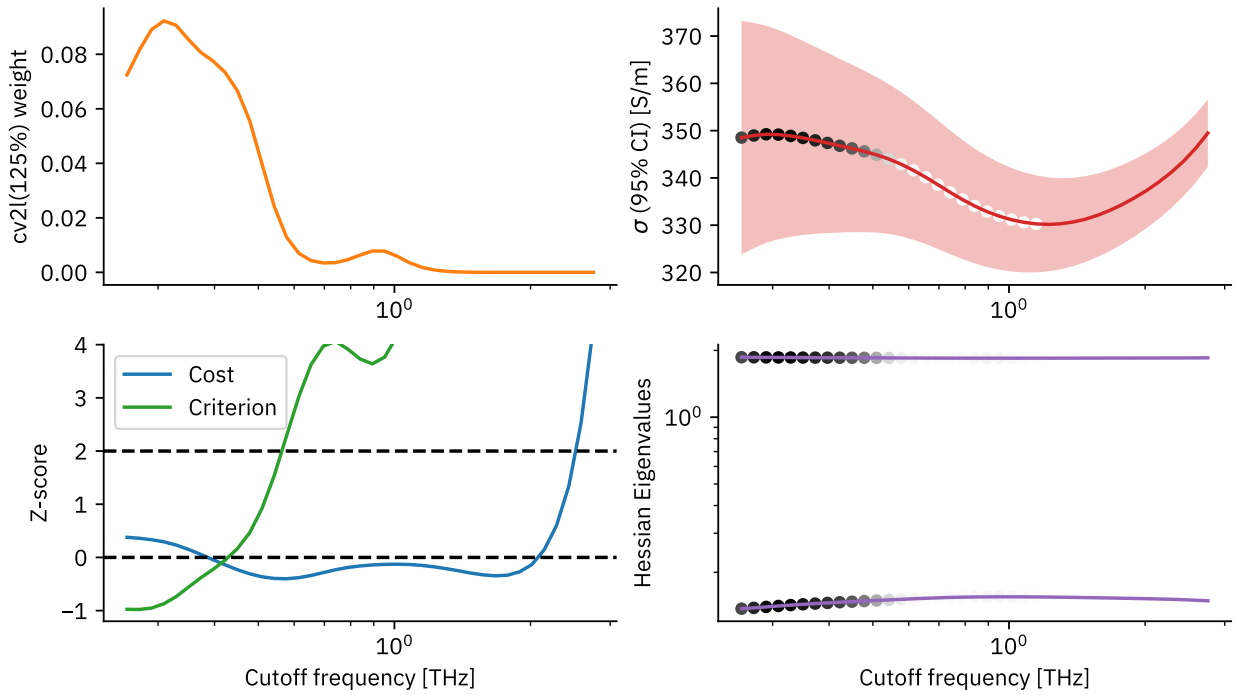
(continues on next page)

(continued from previous page)

```
     87.4        -0.6    2.14e+00
     93.1        10.8    2.28e+00
     99.1        27.5    2.43e+00
    105.5        52.4    2.59e+00
    112.3        89.3    2.75e+00
Cutoff criterion exceeds incumbent + margin: -30.7 + 100.0.

INPUT TIME SERIES
    Time step:                      50.000 fs
    Simulation time:                39950.000 fs
    Maximum degrees of freedom:     600.0

MAIN RESULTS
    Autocorrelation integral:       347.0 ± 10.9 S/m
    Integrated correlation time:    25.116 ± 0.788 fs

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:     15.6 (ideally > 40)
    Regression cost Z-score:        0.1 (ideally < 2)
    Cutoff criterion Z-score:       -0.1 (ideally < 2)

MODEL exppoly(0, 1) | CUTOFF CRITERION cv2l(125%)
    Number of parameters:           2
    Average cutoff frequency:       3.94e-01 THz
```

```
/home/toon/univ/molmod/stacie/src/stacie/model.py:313: RuntimeWarning: overflow encountered␣
↪in exp
  amplitudes_model = np.exp(np.dot(design_matrix, pars))
```





The analysis of the initial production runs shows that the trajectories are not yet sufficient for a reliable interpretation of the autocorrelation integrals:

- Only 15 effective points are used for the fitting.
- The relative error is 3.1%, while higher than the coarse estimate of 0.9%, it is of the right order of magnitude.

The extra plots reveal another reason for extending the MD simulations. The cutoff weight is significant at the lowest cutoff frequency, suggesting that a finer grid with lower frequencies could reveal new details. Hence, we extended the production runs by 8000 additional steps to refine the frequency grid, of which the results are discussed in the following subsection.

5.8.5 Analysis of the Extended Production Simulation (8000 + 8000 steps)

We simply call the same `analyze()` function, but now with `npart=2`, which loads the initial production runs and their first extension.

```
conductivity_2_01 = analyze(ExpPolyModel([0, 1]), npart=2)
```

```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion  fcut [THz]
---------  ----------  ----------
     10.0       -28.0    1.28e-01
     10.7       -27.7    1.36e-01
     11.4       -27.4    1.45e-01
     12.2       -27.3    1.54e-01
     13.0       -27.5    1.64e-01
     13.8       -27.9    1.75e-01
     14.8       -28.5    1.86e-01
     15.8       -29.2    1.98e-01
     16.8       -29.8    2.11e-01
     17.9       -30.3    2.25e-01
```

(continues on next page)

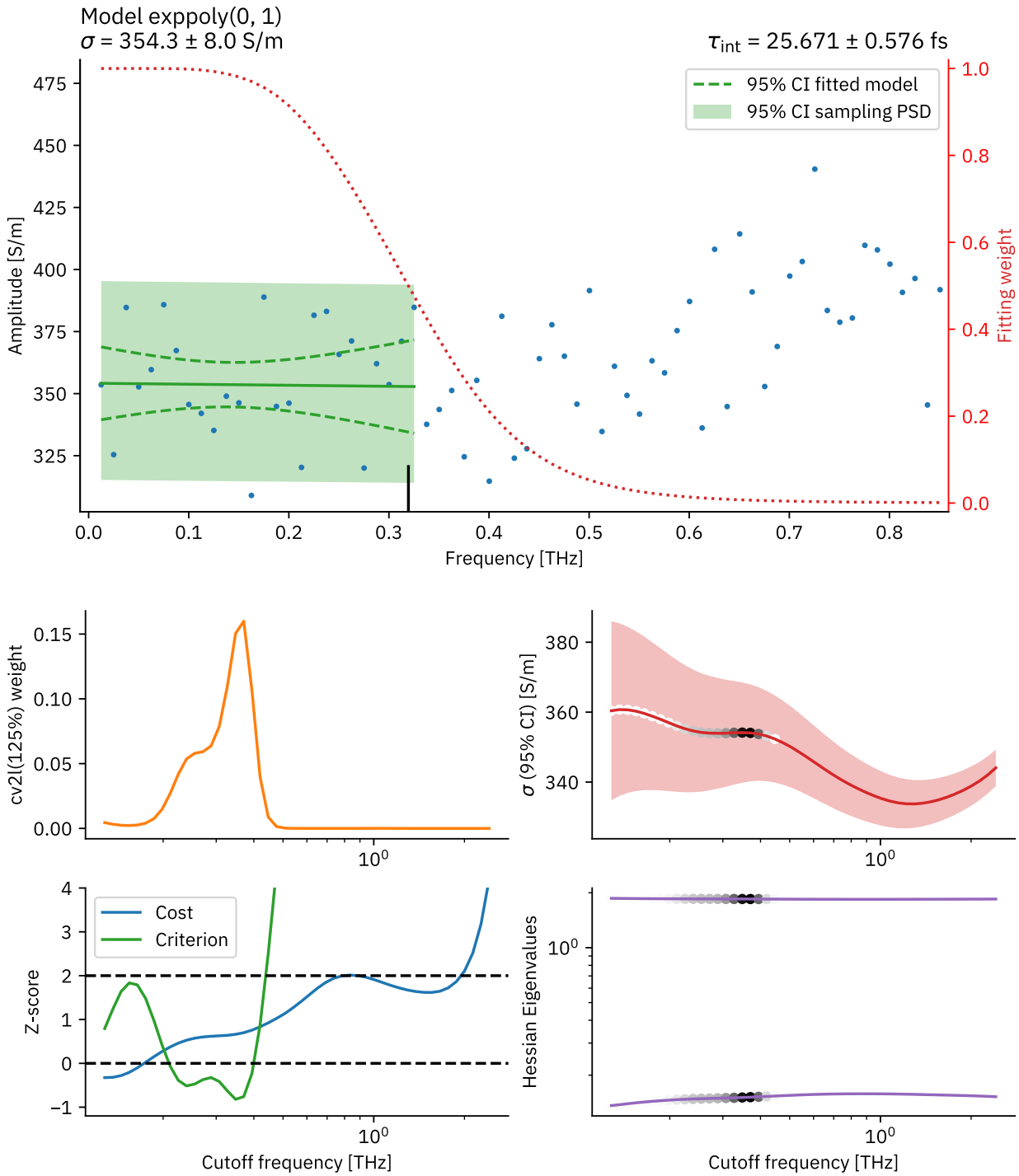
(continued from previous page)

```
     19.1       −30.5    2.39e−01
     20.4       −30.6    2.55e−01
     21.7       −30.6    2.71e−01
     23.2       −30.7    2.88e−01
     24.7       −30.9    3.07e−01
     26.3       −31.2    3.27e−01
     28.0       −31.6    3.48e−01
     29.9       −31.6    3.70e−01
     31.8       −31.2    3.94e−01
     33.9       −30.2    4.20e−01
     36.1       −28.7    4.47e−01
     38.5       −26.9    4.76e−01
     41.0       −25.1    5.06e−01
     43.7       −23.6    5.39e−01
     46.5       −22.7    5.74e−01
     49.6       −22.3    6.11e−01
     52.8       −22.3    6.50e−01
     56.2       −22.4    6.92e−01
     59.9       −22.6    7.36e−01
     63.8       −22.8    7.84e−01
     67.9       −22.9    8.35e−01
     72.4       −23.1    8.88e−01
     77.1       −23.3    9.46e−01
     82.1       −23.5    1.01e+00
     87.4       −23.6    1.07e+00
     93.1       −23.4    1.14e+00
     99.1       −22.9    1.21e+00
    105.5       −22.0    1.29e+00
    112.3       −20.6    1.38e+00
    119.6       −18.8    1.46e+00
    127.4       −16.7    1.56e+00
    135.6       −13.9    1.66e+00
    144.4       −10.0    1.77e+00
    153.7        −4.1    1.88e+00
    163.7         5.1    2.00e+00
    174.3        19.1    2.13e+00
    185.6        40.9    2.27e+00
    197.6        74.1    2.41e+00
Cutoff criterion exceeds incumbent + margin: −31.6 + 100.0.

INPUT TIME SERIES
    Time step:                     50.000 fs
    Simulation time:               79950.000 fs
    Maximum degrees of freedom:    600.0

MAIN RESULTS
    Autocorrelation integral:      354.3 ± 8.0 S/m
    Integrated correlation time:   25.671 ± 0.576 fs

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:    25.7 (ideally > 40)
    Regression cost Z-score:       0.6 (ideally < 2)
    Cutoff criterion Z-score:      −0.4 (ideally < 2)
```

(continues on next page)

(continued from previous page)

```
MODEL exppoly(0, 1) | CUTOFF CRITERION cv2l(125%)
    Number of parameters:          2
    Average cutoff frequency:      3.19e-01 THz
```



The extended analysis shows that the results are starting to converge. However, the number of fitted points is still only 26, which is relatively low. To get more robust results, we extended the simulations once more. We added 184000 more steps, resulting in a simulation time of 1 ns for each of the 100 trajectories.

5.8.6 Analysis of the Extended Production Simulation (8000 + 8000 + 184000 steps)

We simply call the same `analyze()` function, but now with `npart=3`, which loads the initial production runs and their first and second extensions.

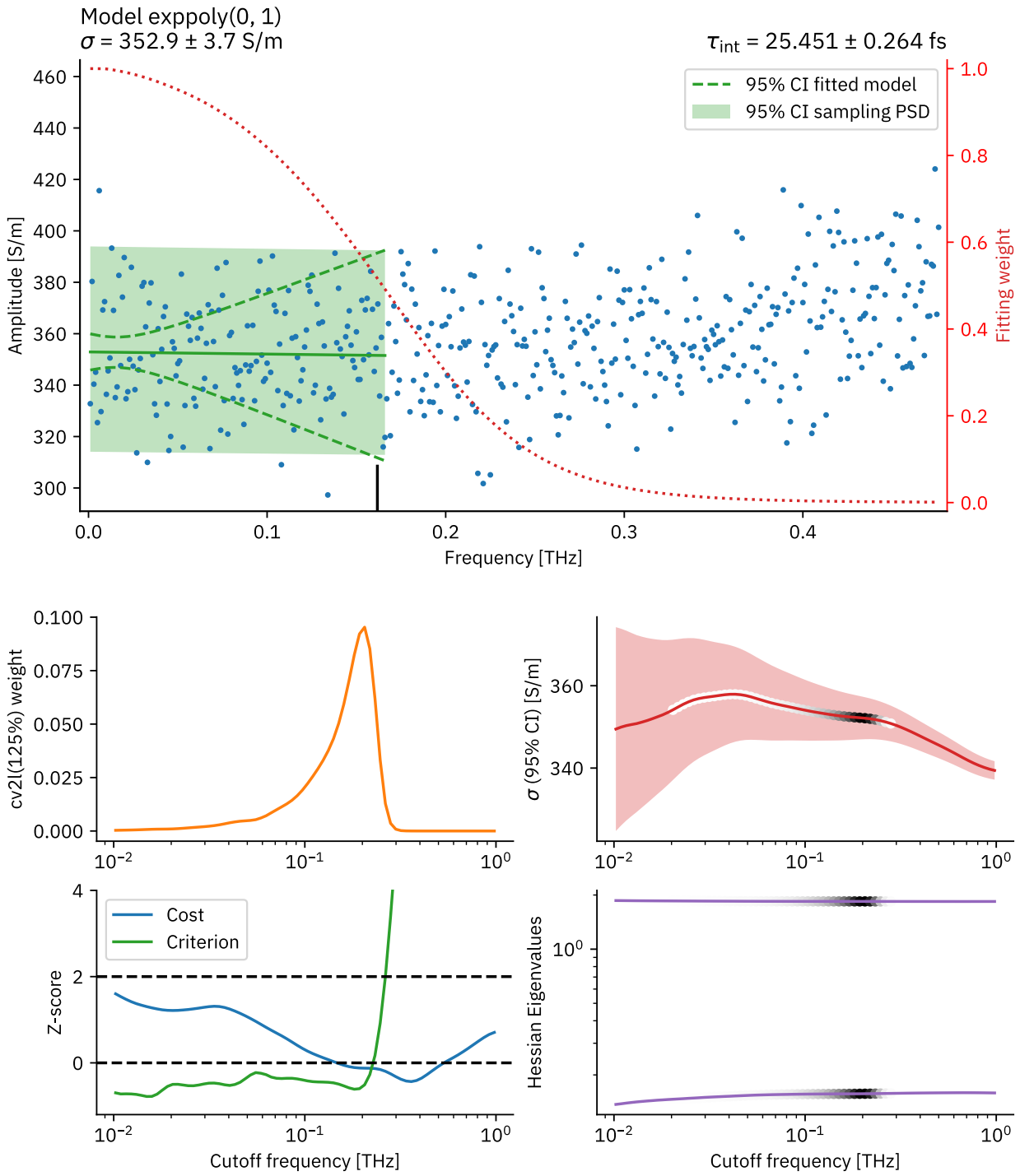
```
conductivity_3_01 = analyze(ExpPolyModel([0, 1]), npart=3)
```

```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion  fcut [THz]
---------  ----------  ----------
     10.0       -27.0    1.02e-02
     10.7       -27.1    1.09e-02
     11.4       -27.3    1.16e-02
     12.2       -27.4    1.23e-02
     13.0       -27.5    1.31e-02
     13.8       -27.6    1.40e-02
     14.8       -27.8    1.49e-02
     15.8       -27.9    1.58e-02
     16.8       -28.0    1.69e-02
     17.9       -28.0    1.80e-02
     19.1       -28.0    1.91e-02
     20.4       -28.1    2.04e-02
     21.7       -28.2    2.17e-02
     23.2       -28.4    2.31e-02
     24.7       -28.5    2.45e-02
     26.3       -28.6    2.61e-02
     28.0       -28.7    2.78e-02
     29.9       -28.8    2.96e-02
     31.8       -28.9    3.15e-02
     33.9       -29.0    3.36e-02
     36.1       -29.2    3.57e-02
     38.5       -29.3    3.80e-02
     41.0       -29.5    4.05e-02
     43.7       -29.6    4.31e-02
     46.5       -29.6    4.59e-02
     49.6       -29.7    4.88e-02
     52.8       -29.7    5.20e-02
     56.2       -29.8    5.53e-02
     59.9       -29.9    5.89e-02
     63.8       -30.1    6.27e-02
     67.9       -30.3    6.67e-02
     72.4       -30.4    7.10e-02
     77.1       -30.5    7.56e-02
     82.1       -30.7    8.05e-02
     87.4       -30.8    8.57e-02
     93.1       -30.9    9.12e-02
     99.1       -31.1    9.71e-02
    105.5       -31.2    1.03e-01
    112.3       -31.4    1.10e-01
    119.6       -31.5    1.17e-01
    127.4       -31.6    1.25e-01
    135.6       -31.7    1.33e-01
    144.4       -31.9    1.41e-01
    153.7       -32.0    1.50e-01
```

(continues on next page)

(continued from previous page)

```
    163.7       -32.2    1.60e-01
    174.3       -32.4    1.70e-01
    185.6       -32.5    1.81e-01
    197.6       -32.6    1.93e-01
    210.3       -32.7    2.06e-01
    223.9       -32.6    2.19e-01
    238.4       -32.2    2.33e-01
    253.8       -31.6    2.48e-01
    270.2       -30.7    2.64e-01
    287.7       -29.4    2.81e-01
    306.3       -27.9    2.99e-01
    326.0       -26.4    3.18e-01
    347.1       -25.0    3.39e-01
    369.5       -23.7    3.61e-01
    393.4       -22.5    3.84e-01
    418.8       -21.3    4.09e-01
    445.8       -20.0    4.35e-01
    474.6       -18.5    4.63e-01
    505.3       -16.8    4.93e-01
    537.9       -15.1    5.25e-01
    572.6       -13.5    5.59e-01
    609.6       -12.2    5.95e-01
    648.9       -11.3    6.33e-01
    690.8       -11.0    6.74e-01
    735.4       -11.0    7.17e-01
    782.8       -11.2    7.64e-01
    833.3       -11.5    8.13e-01
    887.1       -11.7    8.65e-01
    944.4       -11.6    9.21e-01
   1005.3       -11.0    9.81e-01
Reached the maximum number of effective points (1000).

INPUT TIME SERIES
    Time step:                      50.000 fs
    Simulation time:                999950.000 fs
    Maximum degrees of freedom:     600.0

MAIN RESULTS
    Autocorrelation integral:       352.9 ± 3.7 S/m
    Integrated correlation time:    25.451 ± 0.264 fs

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:     165.4 (ideally > 40)
    Regression cost Z-score:        0.1 (ideally < 2)
    Cutoff criterion Z-score:       -0.3 (ideally < 2)

MODEL exppoly(0, 1) | CUTOFF CRITERION cv2l(125%)
    Number of parameters:           2
    Average cutoff frequency:       1.62e-01 THz
```



The analysis of the full extended production runs leads to a modest improvement. However, the utility of the first-order term of the model is questionable, given that the slope is nearly zero and could go either way according to the confidence intervals of the model (green dashed curves). Hence, we first test a constant (white noise) model to the first part of the spectrum:

```
conductivity_3_0 = analyze(ExpPolyModel([0]), npart=3)
```

```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion  fcut [THz]
```

(continues on next page)

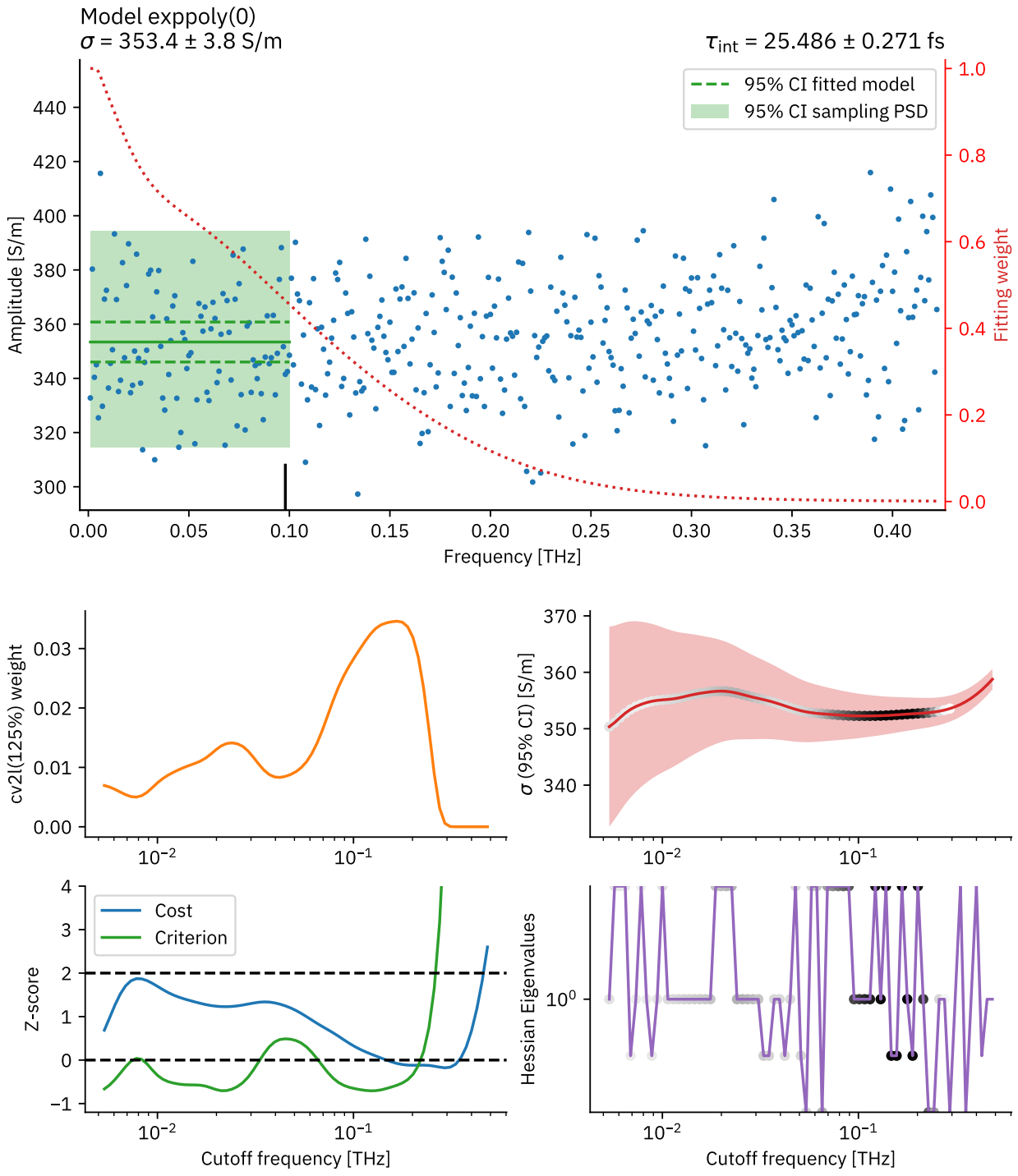
(continued from previous page)

```
---------  ----------  ----------
      5.0        -2.5    5.36e-03
      5.4        -2.4    5.71e-03
      5.7        -2.4    6.07e-03
      6.1        -2.3    6.47e-03
      6.6        -2.2    6.88e-03
      7.0        -2.1    7.33e-03
      7.5        -2.1    7.80e-03
      8.0        -2.2    8.30e-03
      8.6        -2.3    8.84e-03
      9.1        -2.4    9.41e-03
      9.8        -2.5    1.00e-02
     10.4        -2.6    1.07e-02
     11.1        -2.7    1.13e-02
     11.9        -2.7    1.21e-02
     12.7        -2.8    1.29e-02
     13.5        -2.8    1.37e-02
     14.4        -2.9    1.46e-02
     15.4        -2.9    1.55e-02
     16.4        -2.9    1.65e-02
     17.5        -3.0    1.76e-02
     18.7        -3.0    1.87e-02
     19.9        -3.1    1.99e-02
     21.2        -3.1    2.12e-02
     22.6        -3.2    2.26e-02
     24.1        -3.2    2.40e-02
     25.7        -3.2    2.56e-02
     27.4        -3.1    2.72e-02
     29.2        -3.1    2.90e-02
     31.1        -3.0    3.08e-02
     33.2        -2.9    3.28e-02
     35.4        -2.8    3.50e-02
     37.7        -2.7    3.72e-02
     40.1        -2.6    3.96e-02
     42.7        -2.6    4.22e-02
     45.5        -2.7    4.49e-02
     48.5        -2.7    4.78e-02
     51.7        -2.7    5.09e-02
     55.0        -2.8    5.41e-02
     58.6        -2.9    5.76e-02
     62.4        -3.0    6.13e-02
     66.5        -3.2    6.53e-02
     70.8        -3.3    6.95e-02
     75.4        -3.4    7.40e-02
     80.3        -3.6    7.88e-02
     85.5        -3.7    8.38e-02
     91.1        -3.8    8.93e-02
     97.0        -3.8    9.50e-02
    103.2        -3.9    1.01e-01
    109.9        -3.9    1.08e-01
    117.1        -4.0    1.15e-01
    124.6        -4.0    1.22e-01
    132.7        -4.0    1.30e-01
    141.3        -4.0    1.38e-01
```

(continues on next page)

(continued from previous page)

```
    150.4        -4.1    1.47e-01
    160.2        -4.1    1.57e-01
    170.5        -4.1    1.67e-01
    181.6        -4.1    1.77e-01
    193.3        -4.0    1.89e-01
    205.8        -4.0    2.01e-01
    219.1        -3.9    2.14e-01
    233.3        -3.7    2.28e-01
    248.4        -3.3    2.43e-01
    264.4        -2.7    2.58e-01
    281.5        -1.7    2.75e-01
    299.7         0.0    2.93e-01
    319.1         2.8    3.12e-01
    339.7         7.0    3.32e-01
    361.6        13.2    3.53e-01
    385.0        22.0    3.76e-01
    409.8        34.1    4.00e-01
    436.3        50.2    4.26e-01
    464.4        71.6    4.53e-01
    494.4        99.8    4.82e-01
Cutoff criterion exceeds incumbent + margin: -4.1 + 100.0.

INPUT TIME SERIES
    Time step:                      50.000 fs
    Simulation time:                999950.000 fs
    Maximum degrees of freedom:     600.0

MAIN RESULTS
    Autocorrelation integral:       353.4 ± 3.8 S/m
    Integrated correlation time:    25.486 ± 0.271 fs

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:     100.0 (ideally > 20)
    Regression cost Z-score:        0.6 (ideally < 2)
    Cutoff criterion Z-score:       -0.4 (ideally < 2)

MODEL exppoly(0) | CUTOFF CRITERION cv2l(125%)
    Number of parameters:           1
    Average cutoff frequency:       9.80e-02 THz
```



Another model to consider is the Pade model, not because we expect the ACF to decay exponentially, but because it features well-behaved high-frequency limits, which can facilitate the regression.

```
conductivity_3_p = analyze(PadeModel([0, 2], [2]), npart=3)
```

```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion  fcut [THz]
---------  ----------  ----------
```

(continues on next page)

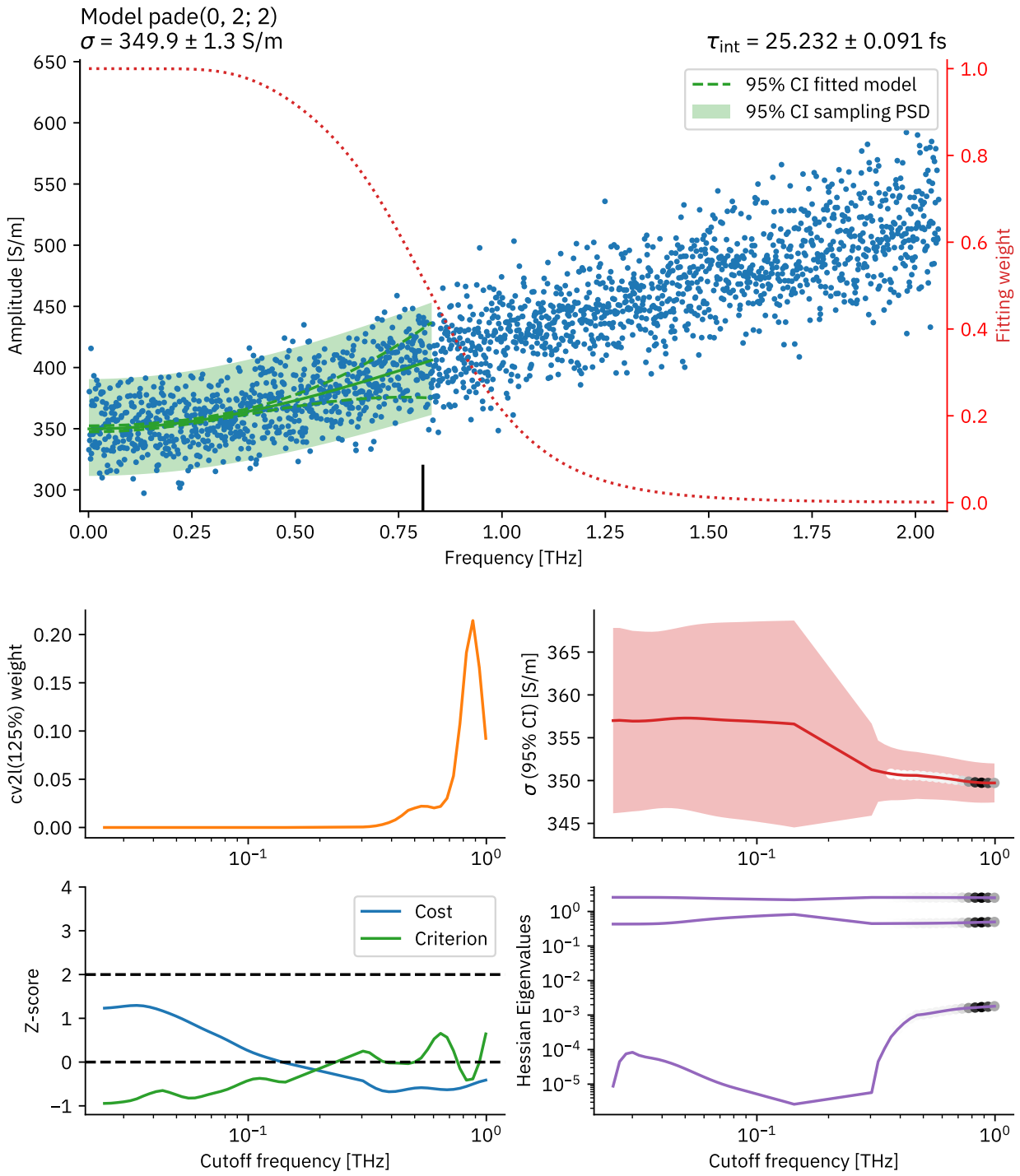
(continued from previous page)

```
    15.0         inf    1.51e-02  (cv2l: Linear dependencies in basis. evals=array([1.
↪63749975e-06, 1.02811096e-03, 2.99897025e+00]))
    16.0         inf    1.61e-02  (cv2l: Linear dependencies in basis. evals=array([1.
↪14118976e-06, 7.93346219e-04, 2.99920551e+00]))
    17.1         inf    1.71e-02  (cv2l: Linear dependencies in basis. evals=array([7.
↪86750561e-07, 6.14028456e-04, 2.99938518e+00]))
    18.2         inf    1.82e-02  (cv2l: Linear dependencies in basis. evals=array([5.
↪42867412e-07, 4.75275041e-04, 2.99952418e+00]))
    19.4         inf    1.94e-02  (cv2l: Linear dependencies in basis. evals=array([3.
↪76196721e-07, 3.69469427e-04, 2.99963015e+00]))
    20.7         inf    2.06e-02  (cv2l: Linear dependencies in basis. evals=array([2.
↪59666420e-07, 2.89088199e-04, 2.99971065e+00]))
    22.0         inf    2.20e-02  (cv2l: Linear dependencies in basis. evals=array([1.
↪80689157e-07, 2.27594112e-04, 2.99977223e+00]))
    23.5         inf    2.34e-02  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-2.74279924e-05,  4.30590654e-01,  2.56943677e+00]))
    25.0       -89.1    2.49e-02
    26.7       -89.7    2.65e-02
    28.4       -90.2    2.82e-02
    30.3       -90.3    3.00e-02
    32.3       -90.0    3.20e-02
    34.4       -89.9    3.40e-02
    36.7       -89.8    3.62e-02
    39.1       -89.7    3.86e-02
    41.6       -89.4    4.11e-02
    44.3       -89.2    4.37e-02
    47.2       -89.0    4.65e-02
    50.3       -88.7    4.95e-02
    53.6       -88.4    5.27e-02
    57.1       -88.0    5.61e-02
    60.8       -87.6    5.97e-02
    64.7       -87.1    6.36e-02
    68.9       -86.7    6.77e-02
    73.4       -86.3    7.21e-02
    78.2       -85.9    7.67e-02
    83.3       -85.5    8.17e-02
    88.7       -85.1    8.69e-02
    94.4       -84.7    9.25e-02
   100.5       -84.2    9.85e-02
   107.1       -83.7    1.05e-01
   114.0       -83.3    1.12e-01
   121.4       -82.9    1.19e-01
   129.2       -82.5    1.26e-01
   137.6       -82.1    1.35e-01
   146.5       -81.6    1.43e-01
   156.0         inf    1.53e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-1.54359370e-06,  4.41366618e-01,  2.55863493e+00]))
   166.1         inf    1.62e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-1.49142797e-06,  4.41582231e-01,  2.55841926e+00]))
   176.8         inf    1.73e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-2.91330612e-06,  4.41639799e-01,  2.55836311e+00]))
   188.3         inf    1.84e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
↪evals=array([-8.02534064e-06,  4.41998992e-01,  2.55800903e+00]))
   200.4         inf    1.96e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
```

(continues on next page)

(continued from previous page)

```
 ↪evals=array([-1.49369521e-05,  4.42334093e-01,  2.55768084e+00]))
    213.4         inf    2.09e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
 ↪evals=array([-2.27159654e-05,  4.42747757e-01,  2.55727496e+00]))
    227.2         inf    2.22e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
 ↪evals=array([-2.87488075e-05,  4.43190437e-01,  2.55683831e+00]))
    241.9         inf    2.36e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
 ↪evals=array([-4.04863273e-05,  4.43778776e-01,  2.55626171e+00]))
    257.5         inf    2.52e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
 ↪evals=array([-4.33777791e-05,  4.44331986e-01,  2.55571139e+00]))
    274.2         inf    2.68e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
 ↪evals=array([-4.06989167e-05,  4.44918800e-01,  2.55512190e+00]))
    291.9         inf    2.85e-01  (opt: Hessian matrix has non-positive eigenvalues:␣
 ↪evals=array([-2.40531661e-05,  4.45570259e-01,  2.55445379e+00]))
    310.7      -102.2    3.03e-01
    330.8      -102.7    3.23e-01
    352.2      -103.3    3.44e-01
    374.9      -103.9    3.66e-01
    399.1      -104.4    3.90e-01
    424.9      -104.9    4.15e-01
    452.3      -105.3    4.41e-01
    481.5      -105.7    4.70e-01
    512.6      -105.8    5.00e-01
    545.7      -105.9    5.32e-01
    580.9      -105.9    5.67e-01
    618.4      -105.8    6.03e-01
    658.4      -105.9    6.42e-01
    700.9      -106.2    6.84e-01
    746.1      -106.8    7.28e-01
    794.2      -107.5    7.75e-01
    845.5      -108.0    8.25e-01
    900.1      -108.2    8.78e-01
    958.1      -107.9    9.35e-01
   1020.0      -107.4    9.95e-01
Reached the maximum number of effective points (1000).

INPUT TIME SERIES
    Time step:                     50.000 fs
    Simulation time:               999950.000 fs
    Maximum degrees of freedom:    600.0

MAIN RESULTS
    Autocorrelation integral:      349.9 ± 1.3 S/m
    Integrated correlation time:   25.232 ± 0.091 fs

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:    829.1 (ideally > 60)
    Regression cost Z-score:       -0.5 (ideally < 2)
    Cutoff criterion Z-score:      -0.1 (ideally < 2)

MODEL pade(0, 2; 2) | CUTOFF CRITERION cv2l(125%)
    Number of parameters:          3
    Average cutoff frequency:      8.09e-01 THz
```



This is indeed a successful regression, with 829 effective points for a three-parameter model. The relative error estimate on the final result is 0.37%.

5.8.7 Density

To enable a proper comparison with the experimental and other simulation results, we also need to estimate the density of the system. This is done by averaging the density over the NpT trajectories from the production runs.

```
def estimate_density(ntraj: int = 100):
    densities = []
    molar_vols = []
    masses = {11: 22.990, 17: 35.45}  # g/mol
    avogadro = 6.02214076e23  # 1/mol
    for itraj in range(ntraj):
        path_npz = DATA_ROOT / f"sim{itraj:04d}_part00_npt_traj.npz"
        if not path_npz.exists():
            print(f"File {path_npz} not found, skipping.")
            continue
        data = np.load(path_npz)
        mass = sum(masses[atnum] for atnum in data["atnums"]) / avogadro
        volume = data["volume"] * 10**6  # from m³ to cm³
        densities.append(mass / volume)
        molar_vols.append(2 * avogadro * volume / len(data["atnums"]) / 2)
    density = np.mean(densities)
    print(f"Mass density: {density:.3f} ± {np.std(densities):.3f} g/cm³")
    print(f"Molar volume: {np.mean(molar_vols):.4f} ± {np.std(molar_vols):.4f} cm³/mol")
    return density


density = estimate_density()
```

```
Mass density: 1.454 ± 0.014 g/cm³
Molar volume: 20.1042 ± 0.1977 cm³/mol
```

5.8.8 Comparison to Literature Results

Transport properties for this system are challenging to compute accurately. Consequently, simulation results from the literature may exhibit some variation. While the results should be broadly comparable to some extent, deviations may arise due to the differences in post-processing techniques, and the absence of reported error bars in some studies. Furthermore, in [WSLY14] smaller simulation cells were used (512 ions instead of 1728), which may also contribute to discrepancies.

In the table below, we included some more results obtained with STACIE than those discussed above. We also computed the conductivity with the Pade model for all cases, which was a better choice in retrospect.

Ensemble	Simulated time [ns]	Density [g/cm3]	Conductivity [S/m]	Reference
NpT+NVE	4	1.454 ± 0.014	347 ± 10.9	init expoly(0,1)
NpT+NVE	8	1.454 ± 0.014	354 ± 8.0	ext1 expoly(0,1)
NpT+NVE	100	1.454 ± 0.014	353 ± 3.7	ext2 expoly(0,1)
NpT+NVE	100	1.454 ± 0.014	353 ± 3.8	ext2 expoly(0)
NpT+NVE	4	1.454 ± 0.014	343 ± 5.4	init pade(0, 2; 2)
NpT+NVE	8	1.454 ± 0.014	346 ± 3.7	ext1 pade(0, 2; 2)
NpT+NVE	100	1.454 ± 0.014	349 ± 1.3	ext2 pade(0, 2; 2)
NpT+NVT	6	1.456	348 ± 7	[WDZ+20]
NpT+NVT	> 5	1.444	≈ 310	[WSLY14]
Experiment	N.A.	1.542 ± 0.006	366 ± 3	[JDL+68] [BH61]

The comparison shows that the results obtained with STACIE align reasonably well with the literature. In terms of statistical efficiency, STACIE achieves comparable or smaller error bars for

about the same simulation time. The deviation from experiment is attributed to the approximations in the NaCl potential. [WDZ+20]

Finally, this example also shows why transport properties can be difficult to compute. As more data is collected, a more detailed spectrum is obtained. Simple models can struggle to explain the increasing amount of information. When extending the total simulation time from 8 ns to 100 ns, the effective number of points in the fit does not grow accordingly. As a result, the uncertainties decrease rather slowly with increasing simulation time.

5.8.9 Technical Details of the Analysis of the Literature Data

References for the experimental data:

- Density [JDL+68]
- Ionic conductivity [JDL+68]

The following cell converts a molar ionic conductivity from the literature back to a conductivity.

```
def convert_molar_conductivity():
    """Convert a specific conductance to a conductivity."""
    # Parameters taken from Wang 2020 (https://doi.org/10.1063/5.0023225)
    # and immediately converted to SI units
    molar_conductivity = 140 * 1e-4  # S m²/mol
    molar_conductivity_std = 3 * 1e-4  # S m²/mol
    density = 1.456 * 1e3  # kg/m³
    molar_mass = (22.990 + 35.45) * 1e-3  # kg/mol
    molar_volume = molar_mass / density  # m³/mol
    conductivity = molar_conductivity / molar_volume
    conductivity_std = molar_conductivity_std / molar_volume
    print("Conductivity [S/m]", conductivity)
    print("Conductivity std [S/m]", conductivity_std)


convert_molar_conductivity()
```

```
Conductivity [S/m] 348.8021902806297
Conductivity std [S/m] 7.474332648870638
```

5.8.10 Validation of the Production Runs

To further establish that our NVE runs together represent the NpT ensemble, the following two cells perform additional validation checks.

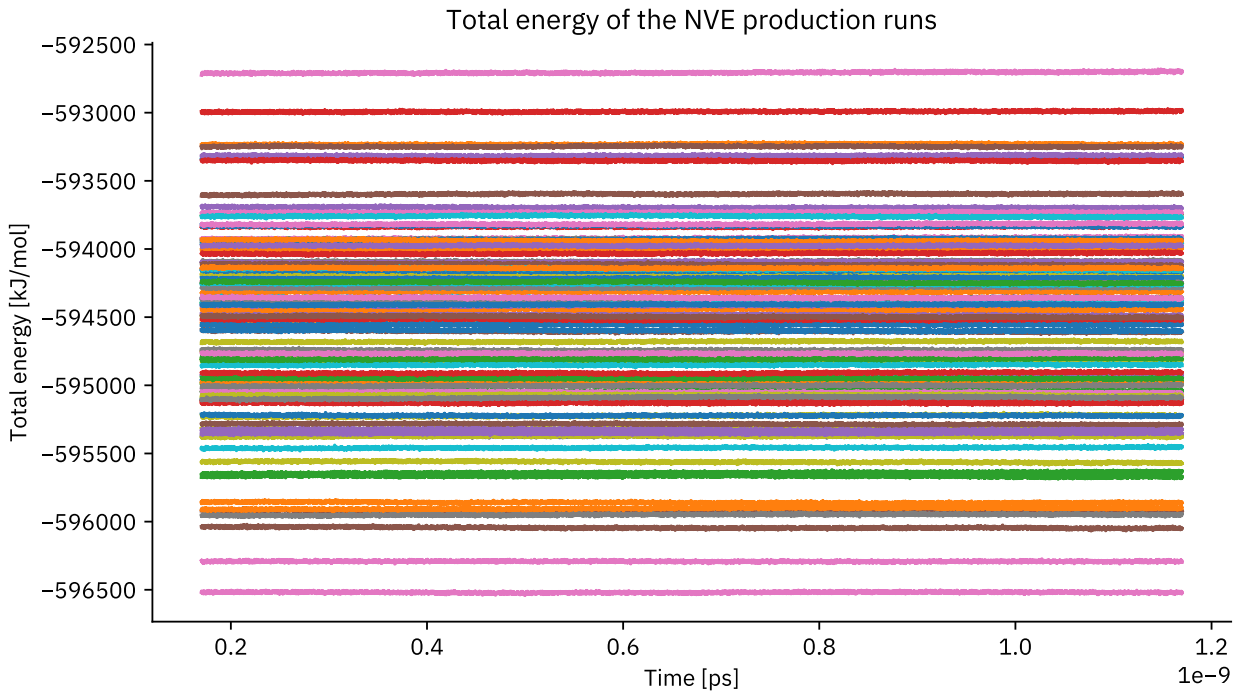
- A plot of the conserved quantity of the separate NVE runs, to detect any drift.
- The distribution of the instantaneous temperature, which should match the desired NpT distribution. For each individual NVE run and for the combined NVE runs, cumulative distributions are plotted. The function also plots the expected cumulative distribution of the NpT ensemble.

```
def plot_total_energy(npart: int = 3, ntraj: int = 100):
    time = None
    energies = []
    for itraj in range(ntraj):
        if itraj == 0:
            time = []
```

(continues on next page)

(continued from previous page)

```
        energies_traj = []
        for ipart in range(npart):
            path_npz = DATA_ROOT / f"sim{itraj:04d}_part{ipart:02d}_nve_traj.npz"
            if not path_npz.exists():
                print(f"File {path_npz} not found, skipping.")
                continue
            data = np.load(path_npz)
            if itraj == 0:
                time.append(data["time"])
            energies_traj.append(data["total_energy"])
        if itraj == 0:
            time = np.concatenate(time)
        energies.append(np.concatenate(energies_traj))

    num = "total_energy"
    plt.close(num)
    _, ax = plt.subplots(num=num)
    for energies_traj in energies:
        plt.plot(time, energies_traj)
    plt.title("Total energy of the NVE production runs")
    plt.xlabel("Time [ps]")
    plt.ylabel("Total energy [kJ/mol]")


plot_total_energy()
```



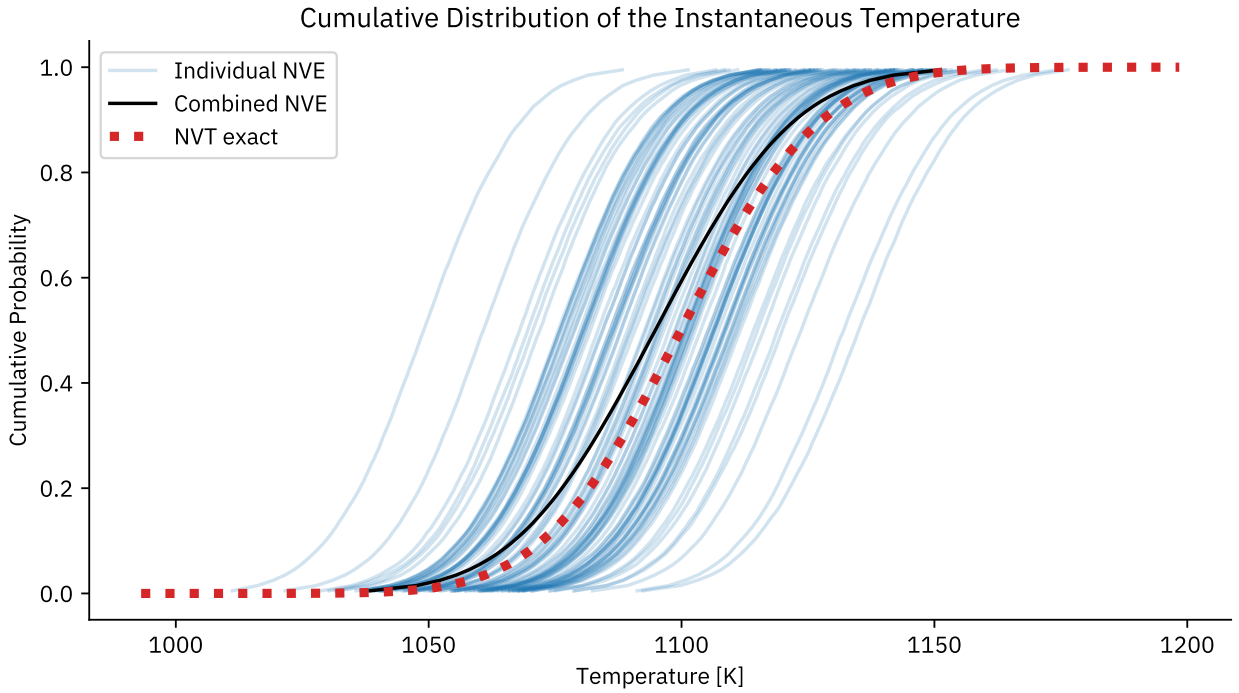
There is no noticeable drift in the total energy of the NVE runs. Apart from the usual (and acceptable) numerical noise, the total energy is conserved perfectly.

```
def plot_temperature_production(npart: int = 3, ntraj: int = 100):
    """Plot cumulative distributions of the instantaneous temperature."""
```

(continues on next page)

(continued from previous page)

```
    # Load the temperature data from the NVE production runs.
    natom = None
    temps = []
    for itraj in range(ntraj):
        temps.append([])
        for ipart in range(npart):
            path_npz = DATA_ROOT / f"sim{itraj:04d}_part{ipart:02d}_nve_traj.npz"
            if not path_npz.exists():
                print(f"File {path_npz} not found, skipping.")
                continue
            data = np.load(path_npz)
            natom = len(data["atnums"])
            temps[-1].append(data["temperature"])
    temps = np.array([np.concatenate(t) for t in temps])

    # Plot the instantaneous and desired temperature distribution.
    plt.close("tempprod")
    _, ax = plt.subplots(num="tempprod")
    ndof = 3 * natom - 3
    temp_d = 1100
    plot_cumulative_temperature_histogram(ax, temps, temp_d, ndof, "K")


plot_temperature_production()
```



Alas, as mentioned above, there is still a small mismatch between the obtained and expected NVT temperature distributions. This notebook will be updated after OpenMM issue #4948 has been resolved.

5.8.11 Regression Tests

If you are experimenting with this notebook, you can ignore any exceptions below. The tests are only meant to pass for the notebook in its original form.

```
if abs(conductivity_1_01 - 347) > 10:
    raise ValueError(f"wrong conductivity (production): {conductivity_1_01:.0f}")
if abs(conductivity_2_01 - 354) > 8:
    raise ValueError(f"wrong conductivity (production): {conductivity_2_01:.0f}")
if abs(conductivity_3_01 - 353) > 7:
    raise ValueError(f"wrong conductivity (production): {conductivity_3_01:.0f}")
if abs(conductivity_3_0 - 353) > 5:
    raise ValueError(f"wrong conductivity (production): {conductivity_3_0:.0f}")
if abs(conductivity_3_p - 349) > 3:
    raise ValueError(f"wrong conductivity (production): {conductivity_3_p:.0f}")
if abs(density - 1.449) > 0.02:
    raise ValueError(f"wrong density (production): {density:.3f}")
```

Some notebooks also use helper functions from the *utils.py* module.

5.9 Utility Module for Plots Reused in Multiple Examples.

```
import matplotlib.pyplot as plt
import numpy as np
from numpy.typing import ArrayLike, NDArray
from scipy.stats import chi2


__all__ = (
    "plot_instantaneous_percentiles",
    "plot_cumulative_temperature_histogram",
    "compute_msds",
)


def compute_msds(series: NDArray[float], lags: NDArray[int]) -> NDArray[float]:
    """Compute the mean squared displacements of one or more time series.

    Parameters
    ----------
    series
        The time series to compute the MSDs for, with shape `(*prefix, nstep)`.
        It is assumed that the last index corresponds to time.
    lags
        The lag times to consider (in number of time steps), with shape `(nlag,)`.

    Returns
    -------
    NDArray[float]
        The mean squared displacements for the specified lag times,
        with shape `(nlag,)`.
    """
    series = np.asarray(series)
    lags = np.asarray(lags)
```

(continues on next page)

(continued from previous page)

```
    if lags.ndim != 1:
        raise ValueError("lags must be a 1D array")
    msds = np.zeros(lags.shape)
    for i, lag in enumerate(lags):
        diffs = np.diff(series[..., ::lag], axis=-1)
        msds[i] = np.mean(diffs**2)
    return msds


def plot_instantaneous_percentiles(
    ax: plt.Axes,
    time: NDArray[float],
    data: NDArray[float],
    percents: ArrayLike,
    expected: ArrayLike | None = None,
    ymin: float | None = None,
    ymax: float | None = None,
):
    """Plot time-dependent percentiles of a data set.

    Parameters
    ----------
    ax
        The axes to plot on.
    time
        The time points corresponding to the data.
    data
        The data to plot. It should be a 2D array with shape (nsample, nstep).
    percents
        The percentages for which to plot the percentiles.
    expected
        The expected values to plot as horizontal lines.
    ymin
        Y-axis lower limit.
    ymax
        Y-axis upper limit.
    """
    for percent, percentile in zip(percents, np.percentile(data, percents, axis=0)):
        ax.plot(time, percentile, label=f"{percent} %")
    if expected is not None:
        for value in expected:
            ax.axhline(value, color="black", linestyle=":")
    ax.set_ylim(bottom=ymin, top=ymax)
    ax.set_title("Percentiles during the equilibration run")
    ax.legend()


def plot_cumulative_temperature_histogram(
    ax: plt.Axes,
    temps: NDArray[float],
    temp_d: float,
    ndof: int,
    temp_unit_str: str,
    nbin: int = 100,
```

(continues on next page)

(continued from previous page)

```
):
    """Plot a cumulative histogram of the temperature.

    Parameters
    ----------
    ax
        The axes to plot on.
    temps
        The temperature data to plot.
        This is expected to be a 2D array with shape (ntraj, nstep).
        Cumulative histograms of individual trajectories will be plotted,
        together with the combined and theoretical cumulative histogram.
    temp_d
        The desired temperature for the theoretical cumulative histogram.
    ndof
        The number of degrees of freedom for the system.
    temp_unit_str
        A string representing the unit of temperature.
    nbin
        The number of bins for the histogram.
    """
    label = "Individual NVE"
    quantiles = (np.arange(nbin) + 0.5) / nbin
    for temp in temps:
        temp.sort()
        ax.plot(
            np.quantile(temp, quantiles),
            quantiles,
            alpha=0.2,
            color="C0",
            label=label,
        )
        label = "__nolegend__"
    ax.plot(
        np.quantile(temps, quantiles),
        quantiles,
        color="black",
        label="Combined NVE",
    )
    temp_axis = np.linspace(np.min(temps), np.max(temps), 100)
    ax.plot(
        temp_axis,
        chi2.cdf(temp_axis * ndof / temp_d, ndof),
        color="C3",
        ls=":",
        lw=4,
        label="NVT exact",
    )
    ax.legend()
    ax.set_title("Cumulative Distribution of the Instantaneous Temperature")
    ax.set_xlabel(f"Temperature [{temp_unit_str}]")
    ax.set_ylabel("Cumulative Probability")
```

To illustrate the applicability of STACIE outside the field of molecular simulations, we also provide

an example analyzing cloud cover data:

5.10 Correlation Time Analysis of Cloud Cover Data

This example is inspired by the work of P. A. Jones [Jon92] on the analysis of temporal (and spatial) correlations in cloud cover data. Jones observed an exponential decay of the autocorrelation function, with half-life times ranging from 5 to 40 hours.

Here, we analyze a time series of cloud cover data obtained from the Open-Meteo platform. The data correspond to hourly cloud cover observations (in percentage of the sky covered by clouds) in Ghent, Belgium, from January 1, 2010 to January 1, 2020.

5.10.1 Library Imports and Matplotlib Configuration

```
import os
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from path import Path
from stacie import (
    compute_spectrum,
    estimate_acint,
    LorentzModel,
    UnitConfig,
    plot_extras,
    plot_fitted_spectrum,
)
from stacie.utils import split
```

```
mpl.rc_file("matplotlibrc")
%config InlineBackend.figure_formats = ["svg"]
```

5.10.2 Cloud Cover Data

In this example, cloud cover is expressed as the fraction of the sky covered by clouds, ranging from 0 (clear sky) to 1 (completely overcast).

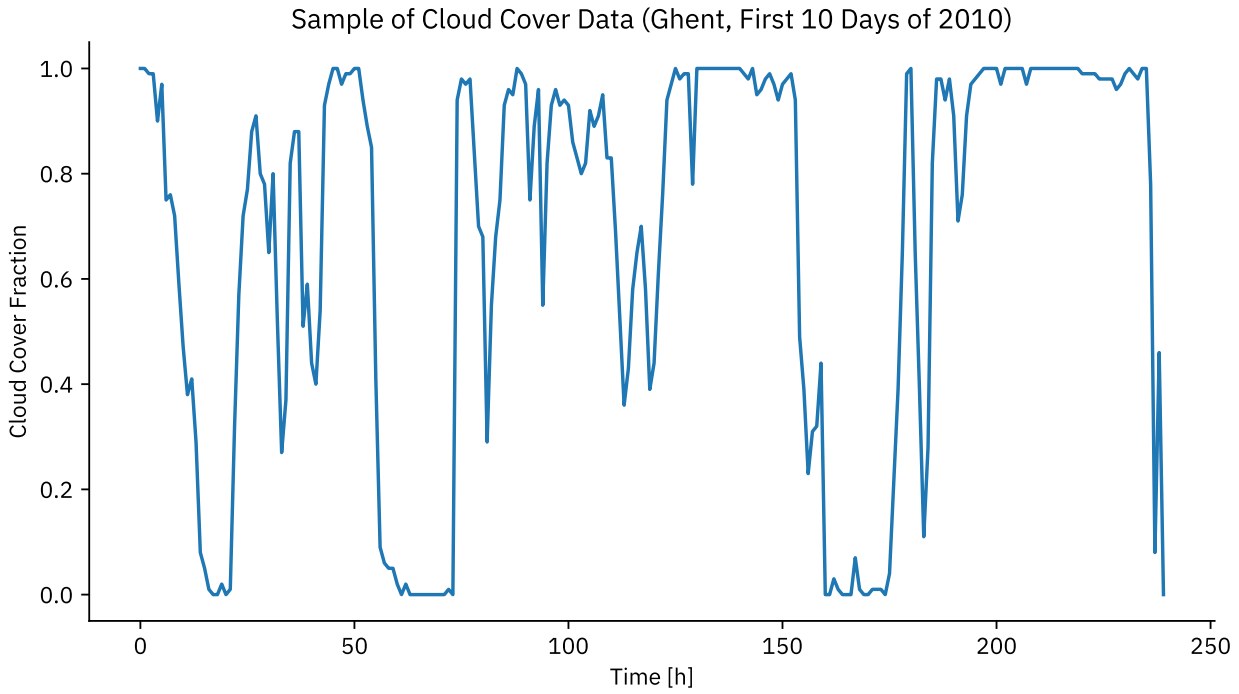
```
# You normally do not need to change this path.
# It only needs to be overridden when building the documentation.
DATA_ROOT = Path(os.getenv("DATA_ROOT", "./")) / "cloud-cover/"

# Load data and convert to a fraction from 0 to 1.
cover = (
    np.loadtxt(
        DATA_ROOT / "cloud-cover-ghent-2010-2020.csv",
        delimiter=",",
        usecols=1,
        skiprows=4,
    )
    / 100
)
print(f"Number of measurements: {cover.shape}")
```

```
Number of measurements: (87672,)
```

5.10.3 Sample of the Cloud Cover Data

To get a first impression of the data, we plot a small sample.

```
def plot_sample():
    plt.close("sample")
    _, ax = plt.subplots(num="sample")
    time = np.arange(240)
    ax.plot(time, cover[:240], "C0-")
    ax.set_xlabel("Time [h]")
    ax.set_ylabel("Cloud Cover Fraction")
    ax.set_title("Sample of Cloud Cover Data (Ghent, First 10 Days of 2010)")
    ax.set_ylim(-0.05, 1.05)


plot_sample()
```



5.10.4 Histogram of Cloud Cover Data

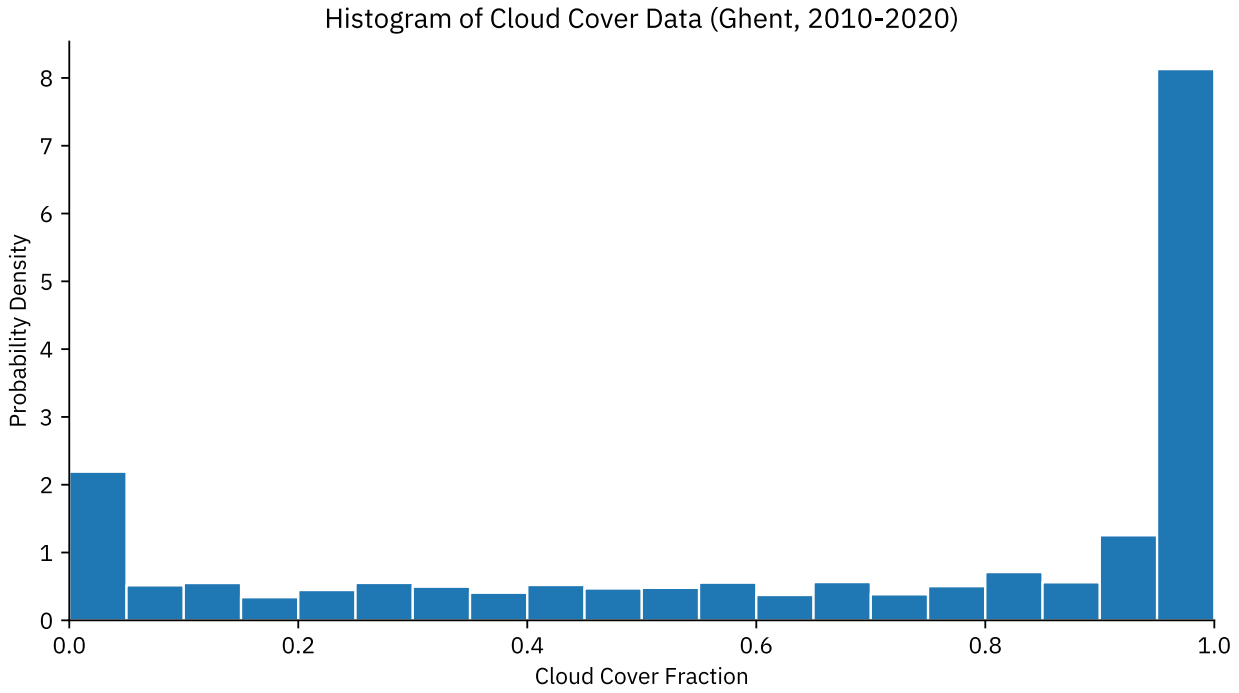
The histogram below shows the distribution of cloud cover values.

```
def plot_histogram():
    plt.close("histogram")
    _, ax = plt.subplots(num="histogram")
    ax.hist(cover, bins=np.linspace(0, 1, 21), density=True, ec="w")
    ax.set_xlabel("Cloud Cover Fraction")
    ax.set_ylabel("Probability Density")
    ax.set_xlim(0, 1)
```

(continues on next page)

(continued from previous page)

```
    ax.set_title("Histogram of Cloud Cover Data (Ghent, 2010-2020)")


plot_histogram()
```



This histogram reflects the typical weather pattern in Belgium, with plenty of cloudy days. This is also reflected in the normalized standard deviation (NS), as defined by Jones [Jon92]:

```
cc_mean = np.mean(cover)
cc_std = np.std(cover)
cc_ns = cc_std / np.sqrt(cc_mean * (1 - cc_mean))
print(f"    Mean cloud cover: {cc_mean:.3f}")
print(f"  Standard deviation: {cc_std:.3f}")
print(f"Normalized std. dev.: {cc_ns:.3f}")
```

```
    Mean cloud cover: 0.668
  Standard deviation: 0.369
Normalized std. dev.: 0.783
```

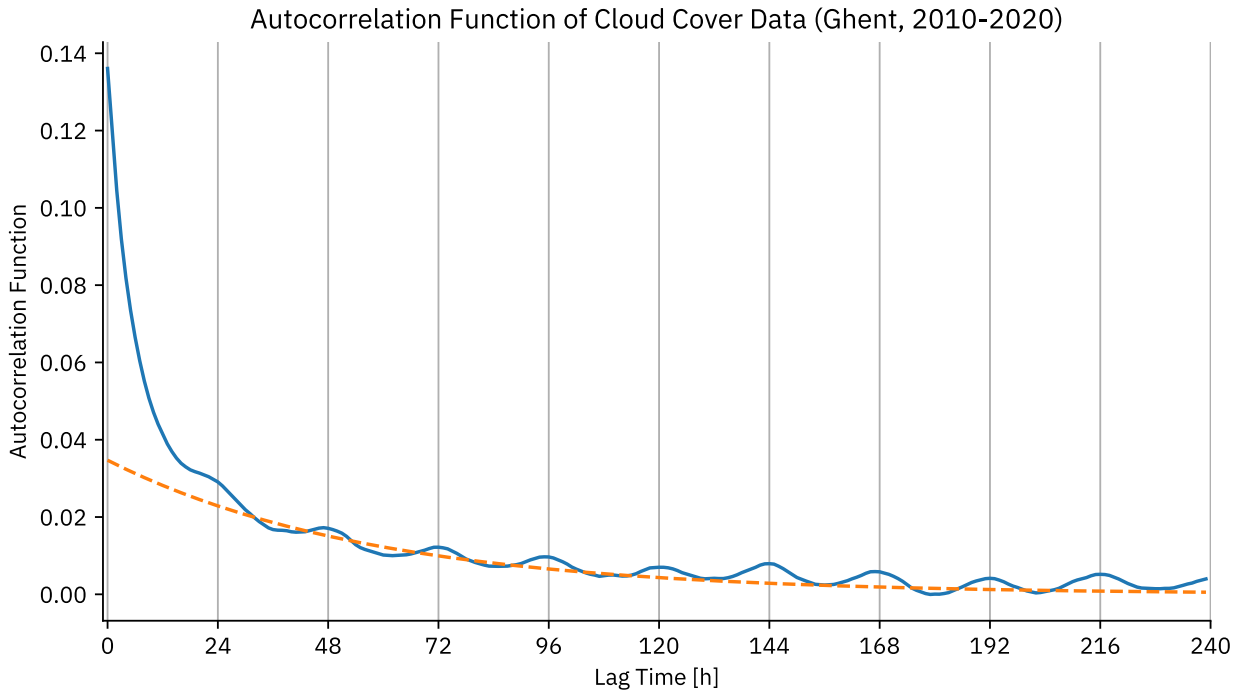
Compared to the values in Figure 10 of Jones's work, this is a relatively high NS, which has been associated with longer correlation times.

5.10.5 Autocorrelation Function

Before applying STACIE to the data, let's first compute and plot the autocorrelation function (ACF) directly.

Mind the normalization: due to the zero padding in `np.correlate`, the number of terms contributing to the ACF decreases with lag time. Furthermore, the ACF is not normalized to 1 in this plot, as we want to keep the amplitude information.

```
C_1 = 0.0347  # From STACIE Lorentz $C_1$ parameter, see below.
TAU_EXP = 57.6  # From STACIE $tau_\text{exp}$, see below.


def plot_acf():
    plt.close("acf")
    _, ax = plt.subplots(num="acf")
    delta = cover - np.mean(cover)
    acf = np.correlate(delta, delta, mode="same")
    nkeep = acf.size // 2
    acf = acf[nkeep:] / (len(acf) - np.arange(nkeep))
    time = np.arange(240)
    ax.plot(time, acf[:240], "C0-")
    ax.plot(time, C_1 * np.exp(-time / TAU_EXP), "C1--")
    xticks = np.arange(0, 241, 24)
    ax.set_xlim(-1, 240)
    ax.set_xticks(xticks)
    ax.grid(True, axis="x")
    ax.set_xlabel("Lag Time [h]")
    ax.set_ylabel("Autocorrelation Function")
    ax.set_title("Autocorrelation Function of Cloud Cover Data (Ghent, 2010-2020)")


plot_acf()
```



The ripples in the ACF are due to diurnal cycles, resulting in weak correlations at multiples of 24 hours. Superimposed on these diurnal effects, an overall exponential decay of the ACF can be observed. However, it is difficult to fit an exponential function to the ACF due to (i) the ripples and (ii) non-exponential short-time effects. Hence, fitting an exponential function can at best provide a rough estimate of the correlation time.

Below, it is shown how to use STACIE instead to perform a more robust analysis. The exponential decay shown in the plot is derived from STACIE's output below. It is only expected to be

representative at long lag times.

5.10.6 Autocorrelation Time

The following cells perform a standard time correlation analysis using STACIE. The `prefactors` argument is set so that the resulting autocorrelation integral is the *variance of the mean* cloud cover. To reduce the noise of the spectrum, the data is split into 20 blocks, and the resulting spectra are averaged.

```
# Compute spectrum
spectrum = compute_spectrum(
    split(cover, 20),
    include_zero_freq=False,
    prefactors=2.0 / len(cover),
)

# Estimate autocorrelation time
uc = UnitConfig(
    time_unit_str="h",
    freq_unit_str="1/h",
    time_fmt=".1f",
    freq_fmt=".3f",
)
result = estimate_acint(spectrum, LorentzModel(), verbose=True, uc=uc)
```

```
CUTOFF FREQUENCY SCAN cv2l(125%)
     neff   criterion  fcut [1/h]
---------  ----------  ----------
     15.0        32.7    0.003
     16.0        31.2    0.004
     17.1        29.9    0.004
     18.2        29.0    0.004
     19.4        28.4    0.004
     20.7        28.0    0.005
     22.0        27.6    0.005
     23.5        27.2    0.005
     25.0        27.0    0.006
     26.7        27.0    0.006
     28.4        27.2    0.006
     30.3        27.5    0.007
     32.3        27.8    0.007
     34.4        27.9    0.008
     36.7        27.9    0.008
     39.1        27.9    0.009
     41.6        27.8    0.009
     44.3        27.7    0.010
     47.2        27.8    0.011
     50.3        28.2    0.011
     53.6        28.8    0.012
     57.1        29.6    0.013
     60.8        30.6    0.014
     64.7        31.5    0.015
     68.9        32.5    0.015
     73.4        33.3    0.016
     78.2        34.0    0.018
```

(continues on next page)

(continued from previous page)

```
     83.3        34.6    0.019
     88.7        35.2    0.020
     94.4        35.8    0.021
    100.5        36.5    0.022
    107.1        37.1    0.024
    114.0        37.6    0.025
    121.4        38.0    0.027
    129.2        38.9    0.029
    137.6        41.2    0.031
    146.5        46.5    0.033
    156.0        56.6    0.035
    166.1        72.8    0.037
    176.8        94.2    0.039
    188.3       118.1    0.042
    200.4       139.9    0.045
Cutoff criterion exceeds incumbent + margin: 27.0 + 100.0.

INPUT TIME SERIES
    Time step:                      1.0 h
    Simulation time:                4383.0 h
    Maximum degrees of freedom:     40.0

MAIN RESULTS
    Autocorrelation integral:       5.16e+00 ± 5.66e-01
    Integrated correlation time:    19.0 ± 2.1 h

SANITY CHECKS (weighted averages over cutoff grid)
    Effective number of points:     31.7 (ideally > 60)
    Regression cost Z-score:        -0.4 (ideally < 2)
    Cutoff criterion Z-score:       0.0 (ideally < 2)

MODEL lorentz() | CUTOFF CRITERION cv2l(125%)
    Number of parameters:           3
    Average cutoff frequency:       0.007 1/h
    Exponential correlation time:   57.6 ± 16.9 h

RECOMMENDED SIMULATION SETTINGS (EXPONENTIAL CORR. TIME)
    Block time:                     < 18.1 ± 9.4 h
    Simulation time:                > 3618.5 ± 133.6 h
```

As expected, the correlation times are on the order of one or a few days. The exponential correlation time, which is about 60 hours, is the longest because it captures the slowest relaxation process. The integrated correlation time is notably shorter at about 20 hours.

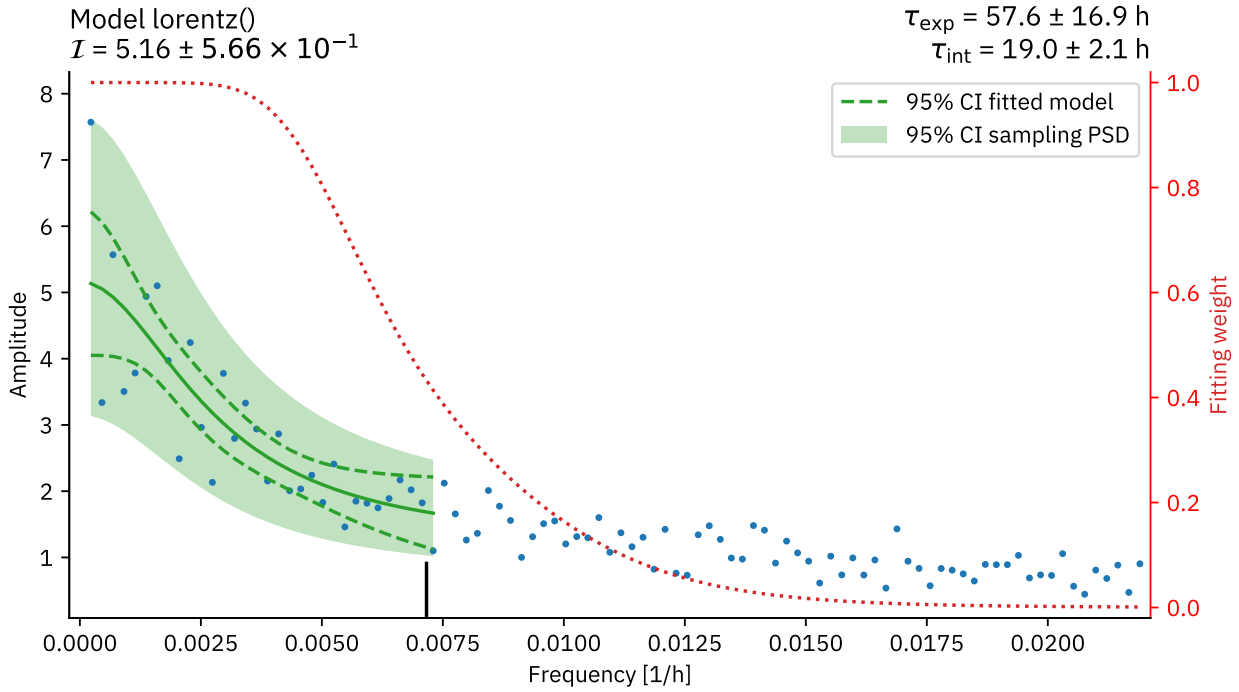
In the code below, we also derive the B parameter of the Lorentz model, which has been used to plot the exponential decay in the ACF above.

```
c_1 = result.props["pars_lorentz"][1]
tau_exp = result.corrtime_exp
print(f"C_1 = {c_1:.4f}")
print(f"TAU_EXP = {tau_exp:.4f} h")
```

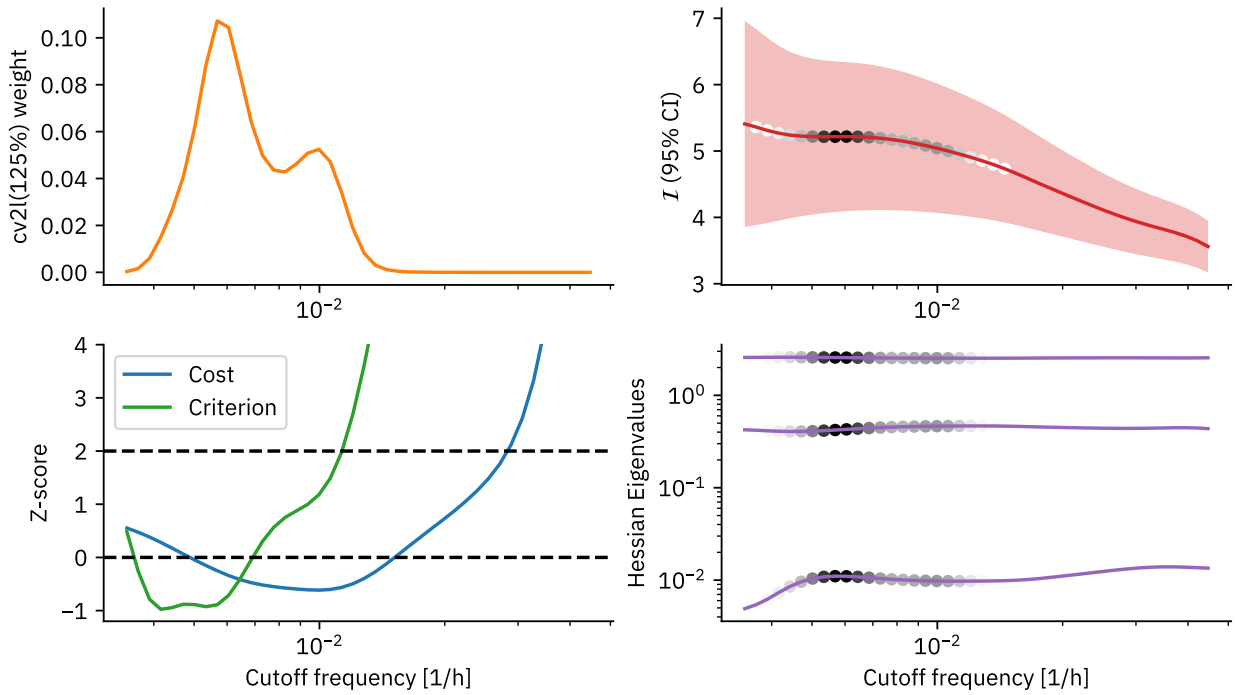
```
C_1 = 0.0347
TAU_EXP = 57.5898 h
```

The plots below show the fitted spectrum and additional diagnostics. These plots can be used to evaluate the quality of the fit and confirm that the Lorentz model is appropriate in this case.

```
plt.close("fitted")
_, ax = plt.subplots(num="fitted")
plot_fitted_spectrum(ax, uc, result)
```




```
plt.close("extras")
_, axs = plt.subplots(2, 2, num="extras")
plot_extras(axs, uc, result)
```



5.10.7 Regression Tests

If you are experimenting with this notebook, you can ignore any exceptions below. The tests are only meant to pass for the notebook in its original form.

```
if abs(result.acint - 5.1612) > 5e-3:
    raise ValueError(f"Wrong acint: {result.acint:.4e}")
if abs(result.corrtime_exp - 57.590) > 5e-2:
    raise ValueError(f"Wrong corrtime_exp: {result.corrtime_exp:.4e}")
if abs(result.props["pars_lorentz"][0] - 1.1683) > 5e-3:
    raise ValueError(f"Wrong lorentz C_0: {result.props['pars_lorentz'][0]:.4e}")
if abs(result.props["pars_lorentz"][1] - 0.0347) > 5e-3:
    raise ValueError(f"Wrong lorentz C_1: {result.props['pars_lorentz'][1]:.4e}")
```

In addition to the worked examples in STACIE's documentation, we also recommend checking out the AutoCorrelation Integral Drill (ACID) Test Set, with which we have validated STACIE's performance:

- ACID GitHub repository: https://github.com/molmod/acid
- ACID Zenodo archive: https://doi.org/10.5281/zenodo.15722903

CHAPTER 6

References

[AW70] B. J. Alder and T. E. Wainwright. Decay of the velocity autocorrelation function. *Phys. Rev. A*, 1(1):18–21, 1970. doi:10.1103/physreva.1.18.

[AT17] Michael P. Allen and Dominic J. Tildesley. *Computer Simulation of Liquids (second edition)*. Oxford University Press, 2017. ISBN 9780198803195. doi:10.1093/oso/9780198803195.001.0001.

[BUNO22] Hiromi Baba, Ryo Urano, Tetsuro Nagai, and Susumu Okazaki. Prediction of self-diffusion coefficients of chemically diverse pure liquids by all-atom molecular dynamics simulations. *J. Comput. Chem.*, 43(28):1892–1900, 2022. doi:10.1002/jcc.26975.

[Bar80] M. S. Bartlett. *Introduction to Stochastic Processes With Special Reference to Methods and Applications*. Cambridge University Press, 1980. ISBN 9780521215855.

[BS13] Joseph E. Basconi and Michael R. Shirts. Effects of temperature control algorithms on transport properties and kinetics in molecular dynamics simulations. *J. Chem. Theory Comput.*, 9(7):2887–2899, 2013. doi:10.1021/ct400109a.

[BH61] J. O'M. Bockris and G. W. Hooper. Self-diffusion in molten alkali halides. *Discuss. Faraday Soc.*, 32:218–236, 1961. doi:10.1039/DF9613200218.

[BBW19] Paul Boone, Hasan Babaei, and Christopher E. Wilmer. Heat flux for many-body interactions: corrections to lammps. *J. Chem. Theory Comput.*, 15(10):5579–5587, August 2019. URL: http://dx.doi.org/10.1021/acs.jctc.9b00252, doi:10.1021/acs.jctc.9b00252.

[Bos96] Georgi N. Boshnakov. Bartlett's formulae—closed forms and recurrent equations. *Ann. Inst. Stat. Math.*, 48(1):49–59, 1996. doi:10.1007/bf00049288.

[DE94] Peter J. Daivis and Denis J. Evans. Comparison of constant pressure and constant volume nonequilibrium simulations of sheared model decane. *J. Chem. Phys.*, 100(1):541–547, 1994. doi:10.1063/1.466970.

[EMB17] Loris Ercole, Aris Marcolongo, and Stefano Baroni. Accurate thermal conductivities from optimally short molecular dynamics simulations. *Sci. Rep.*, 7:15835, 2017. doi:10.1038/s41598-017-15843-2.

[FMP12] George S. Fanourgakis, J. S. Medina, and R. Prosmiti. Determining the bulk viscosity of rigid water models. *J. Phys. Chem. A*, 116(10):2564–2570, March 2012. URL: http://dx.doi.org/10.1021/jp211952y, doi:10.1021/jp211952y.

[FZ09] Christian Francq and Jean-Michel Zakoïan. Bartlett's formula for a general class of nonlinear processes. *J. Time Analysis*, 30(4):449–465, 2009. doi:10.1111/j.1467-9892.2009.00623.x.

[FS02] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation*. Elsevier, 2002. ISBN 9780122673511. doi:10.1016/b978-0-12-267351-1.x5000-7.

[FC70] R. Friedberg and J. E. Cameron. Test of the monte carlo method: fast simulation of a small ising lattice. *J. Chem. Phys.*, 52(12):6049–6058, 1970. doi:10.1063/1.1672907.

[Fue26] E. Fues. Das eigenschwingungsspektrum zweiatomiger moleküle in der undulationsmechanik. *Ann. Phys.*, 385(12):367–396, 1926. doi:10.1002/andp.19263851204.

[Ful95] W.A. Fuller. *Introduction to Statistical Time Series*. Wiley, 1995. ISBN 9780471552390.

[GB19] Federico Grasselli and Stefano Baroni. Topological quantization and gauge invariance of charge transport in liquid insulators. *Nat. Phys.*, 15(9):967–972, 2019. doi:10.1038/s41567-019-0562-0.

[Gre52] Melville S. Green. Markoff random processes and the statistical mechanics of time-dependent phenomena. *J. Chem. Phys.*, 20(8):1281–1295, 1952. doi:10.1063/1.1700722.

[Gre54] Melville S. Green. Markoff random processes and the statistical mechanics of time-dependent phenomena. ii. irreversible processes in fluids. *J. Chem. Phys.*, 22(3):398–413, 1954. doi:10.1063/1.1740082.

[HM13] Jean-Pierre Hansen and Ian R. McDonald. *Theory of Simple Liquids (Fourth Edition)*. Academic Press, 2013. ISBN 978-0-12-387032-2. doi:10.1016/B978-0-12-387032-2.00013-1.

[Hel60] Eugene Helfand. Transport coefficients from dissipation in a canonical ensemble. *Phys. Rev.*, 119(1):1–9, 1960. doi:10.1103/physrev.119.1.

[JWB+19] Seyed Hossein Jamali, Ludger Wolff, Tim M. Becker, Mariëtte de Groen, Mahinder Ramdin, Remco Hartkamp, André Bardow, Thijs J. H. Vlugt, and Othonas A. Moultos. Octp: a tool for on-the-fly calculation of transport properties of fluids with the order-nalgorithm in lammps. *J. Chem. Inf. Model.*, 59(4):1290–1294, February 2019. URL: http://dx.doi.org/10.1021/acs.jcim.8b00939, doi:10.1021/acs.jcim.8b00939.

[JDL+68] G J Janz, F W Dampier, G R Lakshminarayanan, P K Lorenz, and R P T Tomkins. Molten salts ::volume 1. electrical conductance, density, and viscosity data. 1968-01-01 05:01:00 1968. doi:10.6028/NBS.NSRDS.15.

[Jon92] P. A. Jones. Cloud-cover distributions and correlations. *Journal of Applied Meteorology*, 31(7):732–741, 1992. doi:10.1175/1520-0450(1992)031<0732:ccdac>2.0.co;2.

[KGL+22] Qia Ke, Xiaoting Gong, Shouwei Liao, Chongxiong Duan, and Libo Li. Effects of thermostats/barostats on physical properties of liquids by molecular dynamics simulations. *J. Mol. Liq.*, 365:120116, November 2022. URL: http://dx.doi.org/10.1016/j.molliq.2022.120116, doi:10.1016/j.molliq.2022.120116.

[Kra20] A. Kratzer. Die ultraroten rotationsspektren der halogenwasserstoffe. *Z. Phys.*, 3(5):289–307, 1920. doi:10.1007/bf01327754.

[Kub57] Ryogo Kubo. Statistical-mechanical theory of irreversible processes. i. general theory and simple applications to magnetic and conduction problems. *J. Phys. Soc. Jpn.*, 12(6):570–586, 1957. doi:10.1143/JPSJ.12.570.

[Mac05] David J.C. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2005. ISBN 9780521642989.

[MMC+20] Edward J. Maginn, Richard A. Messerly, Daniel J. Carlson, Daniel R. Roe, and J. Richard Elliot. Best practices for computing transport properties 1. self-diffusivity and viscosity from equilibrium molecular dynamics [article v1.0]. *Living J. Comput. Mol. Sci.*, 2(1):6324, 2020. doi:10.33011/livecoms.1.1.6324.

[MLK04a] Karsten Meier, Arno Laesecke, and Stephan Kabelac. Transport coefficients of the lennard-jones model fluid. i. viscosity. *J. Chem. Phys.*, 121(8):3671–3687, 2004. doi:10.1063/1.1770695.

[MLK04b] Karsten Meier, Arno Laesecke, and Stephan Kabelac. Transport coefficients of the lennard-jones model fluid. iii. bulk viscosity. *J. Chem. Phys.*, December 2004. URL: http://dx.doi.org/10.1063/1.1828040, doi:10.1063/1.1828040.

[MFF23] Luís Fernando Mercier Franco and Abbas Firoozabadi. Computation of shear viscosity by a consistent method in equilibrium molecular dynamics simulations: applications to 1-decene oligomers. *J. Phys. Chem. B*, 127(46):10043–10051, 2023. doi:10.1021/acs.jpcb.3c04994.

[Mil11] Russel B. Millar. *Maximum Likelihood Estimation and Inference With Examples in R, SAS and ADMB*. John Wiley & Sons, 2011. ISBN 978-0-470-09482-2.

[OSB99] A.V. Oppenheim, R.W. Schafer, and J.R. Buck. *Discrete-time Signal Processing*. Prentice Hall, 1999. ISBN 9780137549207.

[PDGB25] Paolo Pegolo, Enrico Drigo, Federico Grasselli, and Stefano Baroni. Transport coefficients from equilibrium molecular dynamics. *J. Chem. Phys.*, February 2025. URL: http://dx.doi.org/10.1063/5.0249677, doi:10.1063/5.0249677.

[Pri82] M B Priestley. *Spectral analysis and time series, two-volume set: Volume 1-2*. Academic Press, October 1982. ISBN 9780125649223.

[RW05] Carl Edward Rasmussen and Christopher K I Williams. *Gaussian processes for machine learning*. MIT Press, November 2005. ISBN 0-262-18253-X.

[RS08] G. Rowlands and J. C. Sprott. A simple diffusion model showing anomalous scaling. *Phys. Plasmas*, 2008. doi:10.1063/1.2969429.

[SSWZ20] Yunqi Shao, Keisuke Shigenobu, Masayoshi Watanabe, and Chao Zhang. Role of viscosity in deviations from the nernst–einstein relation. *J. Phys. Chem. B*, 124(23):4774–4780, 2020. doi:10.1021/acs.jpcb.0c02544.

[SS17] R.H. Shumway and D.S. Stoffer. *Time Series Analysis and Its Applications: With R Examples*. Springer International Publishing, 2017. ISBN 9783319524528.

[Sok97] A. Sokal. *Monte Carlo Methods in Statistical Mechanics: Foundations and New Algorithms*, pages 131–192. Springer US, Boston, MA, 1997. doi:10.1007/978-1-4899-0319-8_6.

[SMKO19] Donatas Surblys, Hiroki Matsubara, Gota Kikugawa, and Taku Ohara. Application of atomic stress to compute heat flux via molecular dynamics for systems with many-body interactions. *Phys. Rev.*, May 2019. URL: http://dx.doi.org/10.1103/PhysRevE.99.051301, doi:10.1103/physreve.99.051301.

[SMKO21] Donatas Surblys, Hiroki Matsubara, Gota Kikugawa, and Taku Ohara. Methodology and meaning of computing heat flux via atomic stress in systems with constraint dynamics. *J. Appl. Phys.*, December 2021. URL: http://dx.doi.org/10.1063/5.0070930, doi:10.1063/5.0070930.

[TF64] M.P. Tosi and F.G. Fumi. Ionic sizes and born repulsive parameters in the nacl-type alkali halides—ii: the generalized huggins-mayer form. *J. Phys. Chem. Solids*, 25(1):45–52, 1964. doi:10.1016/0022-3697(64)90160-X.

[Tuc23] Mark E. Tuckerman. *Statistical Mechanics: Theory and Molecular Simulation*. Oxford University Press, 2023. ISBN 9780198825562. doi:10.1093/oso/9780198825562.001.0001.

[VSG07a] S. Viscardy, J. Servantie, and P. Gaspard. Transport and helfand moments in the lennard-jones fluid. i. shear viscosity. *J. Chem. Phys.*, 126(18):184512, 2007. doi:10.1063/1.2724820.

[VSG07b] S. Viscardy, J. Servantie, and P. Gaspard. Transport and helfand moments in the lennard-jones fluid. ii. thermal conductivity. *J. Chem. Phys.*, 126(18):184513, 2007. doi:10.1063/1.2724821.

[WDZ+20] Haimeng Wang, Ryan S. DeFever, Yong Zhang, Fei Wu, Santanu Roy, Vyacheslav S. Bryantsev, Claudio J. Margulis, and Edward J. Maginn. Comparison of fixed charge and polarizable models for predicting the structural, thermodynamic, and transport properties of molten alkali chlorides. *J. Chem. Phys.*, 153(21):214502, 12 2020. doi:10.1063/5.0023225.

[WSLY14] Jia Wang, Ze Sun, Guimin Lu, and Jianguo Yu. Molecular dynamics simulations of the local structures and transport coefficients of molten alkali chlorides. *J. Phys. Chem. B*, 118(34):10196–10206, 2014. doi:10.1021/jp5050332.

[YH04] In-Chul Yeh and Gerhard Hummer. System-size dependence of diffusion coefficients and viscosities from molecular dynamics simulations with periodic boundary conditions. *J. Phys. Chem. B*, 108(40):15873–15879, 2004. doi:10.1021/jp0477147.

[ZZF22] Lili Zhao, Minna Zhi, and Gernot Frenking. The strength of a chemical bond. *Int. J. Quantum Chem.*, 122(8):e26773, 2022. doi:10.1002/qua.26773.

CHAPTER 7

Glossary

ACF
Autocorrelation function. A measure of the correlation of a signal with itself at different time lags.

LAMMPS
Large-scale atomic/molecular massively parallel simulator. A software package for simulating molecular dynamics. See https://www.lammps.org/

MD
Molecular dynamics. A computational method used to simulate the physical movements of atoms and molecules.

NpT
Isothermal-isobaric ensemble. A statistical ensemble that represents a system in thermal equilibrium with a heat bath at constant temperature (T), pressure (p), and number of particles (N).

NVE
Microcanonical ensemble. A statistical ensemble that represents a closed system with fixed energy (E), volume (V), and number of particles (N).

NVT
Canonical ensemble. A statistical ensemble that represents a system in thermal equilibrium with a heat bath at constant temperature (T), volume (V), and number of particles (N).

Uncertainty
An estimate of the standard deviation of a result if the analysis would have been repeated many times with independent inputs. This is also known as the standard error.

PSD
Power spectral density. A measure of the power of a signal as a function of frequency. The Fourier transform of the autocorrelation function.

CHAPTER 8

Application Programming Interface

8.1 stacie package

8.1.1 Submodules

stacie.conditioning module

Cost function pre-conditioning.

class ConditionedCost(*cost*, *par_scales*, *cost_scale*)

Bases: `object`

A wrapper for the cost function taking care of pre-conditioning.

The goal of the pre-conditioner is to let the optimizer work with normalized parameters, and to scale the cost function to a normalized range, such that all quantities are close to 1, even if the spectra and the frequencies have very different orders of magnitude.

> **Parameters**
>
> - **cost** (*Callable[[ndarray[tuple[Any, ...], dtype[float]], int], list[ndarray[tuple[Any, ...], dtype[float]]]]*)
> - **par_scales** (*ndarray[tuple[Any, ...], dtype[float]]*)
> - **cost_scale** (*float*)

__call__(*pars*, ** (Keyword-only parameters separator (PEP 3102))*, *deriv=0*)

Evaluate the pre-conditioned cost function.

> **Parameters**
>
> - **pars** (ndarray[tuple[Any, ...], dtype[float]]) – The parameters to evaluate the cost function at, in the original space. For vectorized calculations, use

N-dimensional inputs of which the last axis corresponds to the parameters.

- **deriv** (int) – The order of the derivative to compute.

Return type
list[ndarray[tuple[Any, ...], dtype[float]]]

Returns
results – The cost function value and its derivatives. In vectorized calculations, the last axis of the gradient and the last two of the Hessian correspond to the parameters.

cost: Callable[[ndarray[tuple[Any, ...], dtype[float]], int], list[ndarray[tuple[Any, ...], dtype[float]]]]

cost_scale: float

from_reduced(*pars*)

Convert parameters from the reduced to the original space.

Parameters
pars (ndarray[tuple[Any, ...], dtype[float]]) – The parameters to convert, in the reduced space.

Return type
ndarray[tuple[Any, ...], dtype[float]]

Returns
pars_orig – The parameters in the original space.

funcgrad(*pars*)

Compute the cost function and the gradient.

Parameters
pars (ndarray[tuple[Any, ...], dtype[float]]) – The parameters, in the reduced space.

Return type
tuple[float, ndarray[tuple[Any, ...], dtype[float]]]

Returns
cost_reduced – The cost normalized function value.

hess(*pars*)

Compute the Hessian matrix of the cost function.

Return type
ndarray[tuple[Any, ...], dtype[float]]

Parameters
pars (*ndarray[tuple[Any, ...], dtype[float]]*)

par_scales: ndarray[tuple[Any, ...], dtype[float]]

to_reduced(*pars*)

Convert parameters from the original to the reduced space.

> **Parameters**
> **pars** (ndarray[tuple[Any, ...], dtype[float]]) – The parameters to convert, in the original space.
>
> **Return type**
> ndarray[tuple[Any, ...], dtype[float]]
>
> **Returns**
> *pars_reduced* – The parameters in the reduced space.

stacie.cost module

Cost function to optimize models for the low-frequency part of the spectrum.

class LowFreqCost(*freqs*, *ndofs*, *amplitudes*, *weights*, *model*)

Bases: object

Cost function to fit a model to the low-frequency part of the spectrum.

> **Parameters**
> - **freqs** (*ndarray[tuple[Any, ...], dtype[float]]*)
> - **ndofs** (*ndarray[tuple[Any, ...], dtype[int]]*)
> - **amplitudes** (*ndarray[tuple[Any, ...], dtype[float]]*)
> - **weights** (*ndarray[tuple[Any, ...], dtype[float]]*)
> - **model** (SpectrumModel)

__call__(*pars*, ***, *deriv=0*)

Evaluate the cost function and its derivatives.

> **Parameters**
> - **pars** (ndarray[tuple[Any, ...], dtype[float]]) – The parameter vector for which the loss function must be computed.
> - **deriv** (int) – The order of derivatives of the cost function to include.
>
> **Return type**
> list[ndarray[tuple[Any, ...], dtype[float]]]
>
> **Returns**
> *results* – A list with the cost function and the requested derivatives.

amplitudes: ndarray[tuple[Any, ...], dtype[float]]

The actual spectrum amplitudes at frequencies in `self.freqs`.

expected(*pars*)

Compute the expected value and variance of the cost function.

Parameters
pars (ndarray[tuple[Any, ...], dtype[float]]) – The model parameters. Vectorization is not supported yet.

Return type
ndarray[tuple[Any, ...], dtype[float]]

Returns
expected, variance – The expected value and variance of the cost function.

freqs: `ndarray[tuple[Any, ...], dtype[float]]`
The frequencies for which the spectrum amplitudes are computed.

model: *SpectrumModel*
The model to be fitted to the spectrum.

ndofs: `ndarray[tuple[Any, ...], dtype[int]]`
The number of independent contributions to each spectrum amplitude.

weights: `ndarray[tuple[Any, ...], dtype[float]]`
The fitting weights for each grid point.

entropy_gamma(*alpha*, *theta*, *, *deriv=0*)
Compute the entropy of the Gamma distribution.

Parameters
- **alpha** (ndarray[tuple[Any, ...], dtype[float]]) – The shape parameter.
- **theta** (ndarray[tuple[Any, ...], dtype[float]]) – The scale parameter.
- **deriv** (int) – The order of the derivatives toward theta to compute: 0, 1 or 2.

Return type
list[ndarray[tuple[Any, ...], dtype[float]]]

Returns
results – A list of results (function value and requested derivatives.) All elements have the same shape as the `alpha` and `theta` arrays.

logpdf_gamma(*x*, *alpha*, *theta*, *, *deriv=0*)
Compute the logarithm of the probability density function of the Gamma distribution.

Parameters
- **x** (ndarray[tuple[Any, ...], dtype[float]]) – The argument of the PDF (random variable). Array with shape `(nfreq,)`.
- **alpha** (ndarray[tuple[Any, ...], dtype[float]]) – The shape parameter. Array with shape `(nfreq,)`.
- **theta** (ndarray[tuple[Any, ...], dtype[float]]) – The scale parameter. Array with shape `( ... , nfreq,)`.
- **deriv** (int) – The order of the derivatives toward theta to compute: 0, 1 or 2.

Return type
list[ndarray[tuple[Any, ...], dtype[float]]]

Returns
results – A list of results (function value and requested derivatives.) All elements have the same shape as the `theta` array.

varlogp_gamma(*alpha*)
Compute the variance of the log-probability density function of the Gamma distribution.

Parameters
alpha (ndarray[tuple[Any, ...], dtype[float]]) – The shape parameter.

Return type
ndarray[tuple[Any, ...], dtype[float]]

Returns
var – The variance of the log-probability density function. Array with shape `(alpha,)`.

stacie.cutoff module

Criteria for selecting the part of the spectrum to fit to.

class CV2LCriterion(**, fcut_factor=1.25, log=False, cond=1000000.0, precondition=True, regularize=True*)
Bases: *CutoffCriterion*

Criterion based on the difference between fits to two halves of the spectrum.

Parameters
- **fcut_factor** (*float*)
- **log** (*bool*)
- **cond** (*float*)
- **precondition** (*bool*)
- **regularize** (*bool*)

__call__(*spectrum, model, props*)
The disparity between fits to two different parts of the spectrum.

Return type
dict[str, float]

Parameters
- **spectrum** (Spectrum)
- **model** (SpectrumModel)
- **props** (*dict[str, ndarray[tuple[Any, ...], dtype[float]]]*)

cond: float
: The threshold for the condition number of the preconditioned covariance matrix.

  Due to the preconditioning, the condition number should be close to 1.0. If not, the linear dependence of the parameters is too strong, making the fit unreliable. In this case, "inf" is returned as the criterion.

fcut_factor: float
: The scale factor to apply to the cutoff frequency.

  If 1.0, the same part of the spectrum is used as in the full non-linear regression. By using a larger value, the default, the criterion also tests whether the fitted parameters can (somewhat) extrapolate to larger frequencies, which reduces the risk of underfitting. This results in less bias on the autocorrelation integral, but slightly larger variance.

log: bool
: Whether to fit a linearized model to the logarithm of the spectrum.

property name: str
: The name of the criterion.

precondition: bool
: Whether to precondition the covariance eigendecomposition.

  This option is only disabled for testing. Always leave it enabled in production.

regularize: bool
: Whether to regularize the linear regression.

  This option is only disabled for testing. Always leave it enabled in production. It will only have an impact on very ill-conditioned fits.

class CutoffCriterion

Bases: object

Base class for cutoff criteria.

Subclasses should implement the `__call__` method.

__call__(*spectrum*, *model*, *props*)
: Compute the criterion for the given spectrum and model.

  Parameters

  - **spectrum** (*Spectrum*) – The spectrum object containing the input data.
  - **model** (*SpectrumModel*) – The model to be fitted to the spectrum.
  - **props** (dict[str, ndarray[tuple[Any, ...], dtype[float]]]) – The property dictionary being constructed in the *stacie.estimate.fit_model_spectrum()* function.

  Return type
  dict[str, float]

  Returns
  results – A dictionary with at least the following fields:

- "criterion": minus the logarithm of a likelihood of "a good fit".
- "criterion_expected": expected value of the negative log likelihood.
- "criterion_var": expected variance of the negative log likelihood.
- "msg": optional message explaining failure to compute the criterion.
- "stop": optional flag to terminate the cutoff scan early.

property name: `str`

The name of the criterion.

linear_weighted_regression(*dm*, *ev*, *ws*, *lc=None*, *ridge=0.0*)

Perform a linear regression with multiple weight vectors.

This is a helper function for cv2l_criterion.

Parameters

- **dm** (ndarray[tuple[Any, ...], dtype[float]]) – The design matrix. Shape `(neq, npar)`, where `neq` is the number of equations and `npar` is the number of parameters.
- **ev** (ndarray[tuple[Any, ...], dtype[float]]) – The expected values, with standard normal measurement errors. Shape `(neq,)`.
- **ws** (ndarray[tuple[Any, ...], dtype[float]]) – A set of weight vectors for the rows of dm (equations). Shape `(nw, neq)`, where `nw` is the number of weight vectors.
- **lc** (ndarray[tuple[Any, ...], dtype[float]] | None) – Linear combinations of solutions for different weights to be computed. Shape `(nlc, nw)`, where `nlc` is the number of linear combinations. If None, the identity matrix is used with shape `(nw, nw)`.
- **ridge** (*float*)

Return type

tuple[ndarray[tuple[Any, ...], dtype[float]], ndarray[tuple[Any, ...], dtype[float]]]

Returns

- *xs* – The regression coefficients for each weight vector. Shape `(nw, npar)`.
- *cs* – The covariance matrices for each combination of weight vector. Shape `(nw, npar, nw, npar)`.

switch_func(*x*, *cutoff*, *exponent*)

Evaluate the switching function at a given points x.

Return type

ndarray[tuple[Any, ...], dtype[float]]

Parameters

- **x** (*ndarray[tuple[Any, ...], dtype[float]]*)
- **cutoff** (*float*)
- **exponent** (*float*)

stacie.estimate module

Algorithm to estimate the autocorrelation integral.

class Result(*spectrum*, *model*, *cutoff_criterion*, *props*, *history*)

Bases: `object`

Container class holding all the results of the autocorrelation integral estimate.

> **Parameters**
>
> - **spectrum** (Spectrum)
> - **model** (SpectrumModel)
> - **cutoff_criterion** (CutoffCriterion)
> - **props** (*dict[str]*)
> - **history** (*list[dict[str]]*)

property acint: float

The autocorrelation integral.

property acint_std: float

The uncertainty of the autocorrelation integral.

property corrtime_exp: float

The exponential correlation time.

property corrtime_exp_std: float

The uncertainty of the exponential correlation time.

property corrtime_int: float

The integrated correlation time.

property corrtime_int_std: float

The uncertainty of the integrated correlation time.

cutoff_criterion: *CutoffCriterion*

The criterion used to select or weight cutoff frequencies.

property fcut: int

The weighted average of the cutoff frequency.

history: list[dict[str]]

History of the cutoff optimization.

Each item is a dictionary returned by *fit_model_spectrum()*, containing the intermediate results of the fitting process. They are sorted from low to high cutoff frequency.

model: *SpectrumModel*

The model used to fit the low-frequency part of the spectrum.

property ncut: int

The number of points where the fitting weight is larger than 1/1000.

property neff: int

The weighted average of the effective number of frequencies used in the fit.

props: dict[str]

The properties marginalized over the ensemble of cutoff frequencies.

The following properties documented in *fit_model_spectrum()* are estimated as weighted averages over the cutoff frequencies:

- `amplitudes_model`: model amplitudes at the included frequencies
- `acint`: autocorrelation integral
- `acint_std`: uncertainty of the autocorrelation integral
- `acint_var`: variance of the autocorrelation integral
- `cost_zscore`: z-score of the cost function
- `criterion_zscore`: z-score of the cutoff criterion
- `fcut`: cutoff frequency
- `pars`: model parameters
- `pars_covar`: covariance matrix of the parameters

Some properties are not averaged over cutoff frequencies:

- `ncut`: number of points included in the fit, i.e. with weight larger than WEIGHT_EPS
- `switch_exponent`: exponent used to construct the cutoff
- `weights`: the weights used to combine the spectrum points in the fit

When using *stacie.model.LorentzModel*, the following properties are added (derived from the marginalized parameters and their covariance):

- `pars_lorentz`: Lorentz parameters (converted from the Padé parameters)
- `pars_lorentz_covar`: covariance matrix of the Lorentz parameters
- `corrtime_exp`: exponential correlation time
- `corrtime_exp_var`: variance of the exponential correlation time
- `corrtime_exp_std`: standard error of the exponential correlation time
- `exp_simulation_time`: recommended simulation time based on the exponential correlation time
- `exp_block_time`: recommended block time based on the exponential correlation time

When using *stacie.model.ExpPolyModel*, the following additional properties are added (derived from the marginalized parameters and their covariance):

- `log_acint`: the logarithm of the autocorrelation integral
- `log_acint_var`: variance of the logarithm of the autocorrelation integral
- `log_acint_std`: standard error of the logarithm of the autocorrelation integral

spectrum: *Spectrum*

The input spectrum from which the autocorrelation integral is estimated.

estimate_acint(*spectrum*, *model*, ***, *neff_min=None*, *neff_max=1000*, *fcut_min=None*, *fcut_max=None*, *fcut_spacing=0.5*, *switch_exponent=8.0*, *cutoff_criterion=None*, *rng=None*, *nonlinear_budget=100*, *criterion_high=100*, *verbose=False*, *uc=None*)

Estimate the integral of the autocorrelation function.

It is recommended to leave the keyword arguments to their default values, except for methodological testing.

This function fits a model to the low-frequency portion of the spectrum and derives an estimate of the autocorrelation (and its uncertainty) from the fit. It repeats this for a range of cutoff frequencies on a logarithmic grid. Finally, an ensemble average over all cutoffs is computed, by using `-np.log` of the cutoff criterion as weight.

The loop over all cutoff frequencies is performed in *scan_frequencies()*, while the marginalization over cutoff frequencies is done in *marginalize_properties()*. The function *fit_model_spectrum()* performs the actual fitting of the model for a given cutoff frequency.

The cutoff frequency grid is logarithmically spaced, with the ratio between two successive cutoff frequencies given by

$$\frac{f_{k+1}}{f_k} = \exp(g_{\mathrm{sp}}/\beta)$$

where g_{sp} is `fcut_spacing` and β is `switch_exponent`.

Parameters

- **spectrum** (*Spectrum*) – The power spectrum and related metadata, used as inputs for the estimation of the autocorrelation integral. This object can be prepared with the function: *stacie.spectrum.compute_spectrum()*.
- **model** (*SpectrumModel*) – The model used to fit the low-frequency part of the spectrum.
- **neff_min** (`int` | `None`) – The minimum effective number of frequency data points to include in the fit. (The effective number of points is the sum of weights in the smooth cutoff.) If not provided, this is set to 5 times the number of model parameters as a default.
- **neff_max** (`int` | `None`) – The maximum number of effective points to include in the fit. This parameter limits the total computational cost. Set to None to disable this stopping criterion.
- **fcut_min** (`float` | `None`) – The minimum cutoff frequency to use. If given, this parameter can only increase the minimal cutoff derived from `neff_min`.
- **fcut_max** (`float` | `None`) – If given, cutoffs beyond this maximum are not considered.
- **fcut_spacing** (`float`) – Dimensionless parameter that controls the spacing between cutoffs in the grid.
- **switch_exponent** (`float`) – Controls the sharpness of the cutoff. Lower values lead to a smoother cutoff, and require fewer cutoff grid points. Higher values sharpen the cutoff, reveal more details, but a finer cutoff grid.
- **cutoff_criterion** (*CutoffCriterion* | `None`) – The criterion function that is minimized to find the best cutoff frequency and, consequently, the optimal number of points included in the fit. If not given, the default is an instance of *stacie.cutoff.CV2LCriterion*.
- **rng** (`Generator` | `None`) – A random number generator for sampling guesses of the nonlinear parameters. If not provided, `np.random.default_rng(42)` is used. The seed is fixed by default for reproducibility.
- **nonlinear_budget** (`int`) – The number of samples used for the nonlinear parameters, calculated as `nonlinear_budget ** num_nonlinear`.

- **criterion_high** (float) – An high increase in the cutoff criterion value, used to terminate the search for the cutoff frequency.
- **verbose** (bool) – Set this to True to print progress information of the frequency cutoff search to the standard output.
- **uc** (*UnitConfig* | None) – Unit configuration object used to format the screen output. If not provided, the default unit configuration is used. See *stacie.utils.UnitConfig* for details. This only affects the screen output (if any) and not the results!

Return type
Result

Returns
result – The inputs, intermediate results and outputs of the algorithm.

finalize_properties(*props*, *model*)

Add remaining properties in-place.

Parameters

- **props** (dict[str]) – The properties dictionary to finalize. This is either the output of *fit_model_spectrum()* or the marginalized properties obtained by *marginalize_properties()*. This dictionary is modified in-place to add model-specific properties and to compute standard errors from variances.
- **model** (*SpectrumModel*) – The model used to fit the spectrum.

fit_model_spectrum(*spectrum*, *model*, *fcut*, *switch_exponent*, *cutoff_criterion*, *rng*, *nonlinear_budget*)

Optimize the parameter of a model for a given spectrum and cutoff frequency.

Parameters

- **spectrum** (*Spectrum*) – The spectrum object containing the input data.
- **model** (*SpectrumModel*) – The model to be fitted to the spectrum.
- **fcut** (float) – The cutoff frequency (in frequency units) used to construct the weights.
- **switch_exponent** (float) – Controls the sharpness of the cutoff. Lower values lead to a smoother cutoff. Higher values sharpen the cutoff.
- **cutoff_criterion** (*CutoffCriterion*) – The criterion function that is minimized to find the optimal cutoff (and thus determine the number of points to include in the fit).
- **rng** (Generator) – A random number generator for sampling guesses of the nonlinear parameters.
- **nonlinear_budget** (int) – The number of samples to use for the nonlinear parameters is `nonlinear_budget ** num_nonlinear`

Return type
dict[str]

Returns

> *props* – A dictionary containing various intermediate results of the cost function calculation. See Notes for details.

Notes

The returned dictionary contains at least the following items, irrespective of whether the fit succeeds or fails:

- `fcut`: cutoff frequency used
- `ncut`: number of points included in the fit, i.e. with weight larger than WEIGHT_EPS
- `switch_exponent`: exponent used to construct the cutoff
- `neff`: effective number of points used in the fit (sum of weights)
- `pars_init`: initial guess of the parameters
- `criterion`: value of the cutoff criterion, or infinity if the fit fails.
- `msg`: error message, if the fit fails

If the fit succeeds, the following additional statistical estimates are also set:

- `acint`: autocorrelation integral
- `acint_var`: variance of the autocorrelation integral
- `acint_std`: standard error of the autocorrelation integral
- `cost_value`: cost function value
- `cost_grad`: cost gradient vector (if `deriv ≥ 1`)
- `cost_hess`: cost Hessian matrix (if `deriv == 2`)
- `cost_hess_scales`: Hessian rescaling vector, see `robust_posinv`.
- `cost_hess_rescaled_evals`: Rescaled Hessian eigenvalues
- `cost_hess_rescaled_evecs`: Rescaled Hessian eigenvectors
- `cost_expected`: expected value of the cost function
- `cost_var`: expected variance of the cost function
- `cost_zscore`: z-score of the cost function
- `criterion_expected`: expected value of the cutoff criterion
- `criterion_var`: expected variance of the cutoff criterion
- `criterion_zscore`: z-score of the cutoff criterion
- `ll`: log likelihood
- `pars`: model parameters
- `pars_covar`: covariance matrix of the model parameters

When using *`stacie.model.LorentzModel`*, the following estimates are added:

- `pars_lorentz`: Lorentz parameters (converted from the Padé parameters)
- `pars_lorentz_covar`: covariance matrix of the Lorentz parameters
- `corrtime_exp`: exponential correlation time, the slowest time scale in the sequences
- `corrtime_exp_var`: variance of the exponential correlation time
- `corrtime_exp_std`: standard error of the exponential correlation time

- `exp_simulation_time`: recommended simulation time based on the exponential correlation time
- `exp_block_time`: recommended block time based on the exponential correlation time

When using *stacie.model.ExpPolyModel*, the following estimates are added:

- `log_acint`: the logarithm of the autocorrelation integral
- `log_acint_var`: variance of the logarithm of the autocorrelation integral
- `log_acint_std`: standard error of the logarithm of the autocorrelation integral

marginalize_properties(*spectrum*, *model*, *history*, ***, *switch_exponent=8.0*, *cutoff_criterion=None*)

Marginalize the properties over the ensemble of cutoff frequencies.

:param See *estimate_acint()* for parameter descriptions other than *history*.: :param The *history* parameter is the list of dictionaries returned by *scan_frequencies()*.:

Return type
Result

Returns
result – The inputs, intermediate results and outputs of the algorithm. This object is returned by the function *estimate_acint()*.

Parameters

- **spectrum** (Spectrum)
- **model** (SpectrumModel)
- **history** (*list[dict[str]]*)
- **switch_exponent** (*float*)
- **cutoff_criterion** (CutoffCriterion | *None*)

scan_frequencies(*spectrum*, *model*, ***, *neff_min=None*, *neff_max=1000*, *fcut_min=None*, *fcut_max=None*, *fcut_spacing=0.5*, *switch_exponent=8.0*, *cutoff_criterion=None*, *rng=None*, *nonlinear_budget=100*, *criterion_high=100*, *verbose=False*, *uc=None*)

Scan over cutoff frequencies and fit a model for each cutoff.

:param See *estimate_acint()* for parameter descriptions.:

Return type
list[dict[str]]

Returns
history – A list of dictionaries, one for each cutoff frequency, each containing various intermediate results of the fitting.

Parameters

- **spectrum** (Spectrum)
- **model** (SpectrumModel)
- **neff_min** (*int* | *None*)
- **neff_max** (*int* | *None*)
- **fcut_min** (*float* | *None*)

- **fcut_max** (*float | None*)
- **fcut_spacing** (*float*)
- **switch_exponent** (*float*)
- **cutoff_criterion** (*CutoffCriterion | None*)
- **rng** (*Generator | None*)
- **nonlinear_budget** (*int*)
- **criterion_high** (*float*)
- **verbose** (*bool*)
- **uc** (*UnitConfig | None*)

summarize_results(*res*, *uc=None*)

Return a string summarizing the Result object.

Parameters

- **res** (*Result | list[Result]*)
- **uc** (*UnitConfig | None*)

stacie.model module

Models to fit the low-frequency part of the spectrum.

class ExpPolyModel(*degrees*)

Bases: *SpectrumModel*

Exponential function of a linear combination of simple monomials.

Parameters

degrees (*ndarray[tuple[Any, ...], dtype[int]]*)

bounds()

Return parameter bounds for the optimizer.

Return type

list[tuple[float, float]]

compute(*freqs*, *pars*, ***, *deriv=0*)

See *SpectrumModel.compute()*.

Return type

list[ndarray[tuple[Any, ...], dtype[float]]]

Parameters

- **freqs** (*ndarray[tuple[Any, ...], dtype[float]]*)

- **pars** (*ndarray[tuple[Any, ...], dtype[float]]*)
- **deriv** (*int*)

degrees: ndarray[tuple[Any, ...], dtype[int]]

The degree of the monomials.

derive_props(*props*)

Add the autocorrelation integral (and other properties) derived from the parameters.

Parameters

props (*dict[str, ndarray[tuple[Any, ...], dtype[float]]]*)

get_par_nonlinear()

Return a boolean mask for the nonlinear parameters.

Return type

ndarray[tuple[Any, ...], dtype[bool]]

property name

property npar

Return the number of parameters.

property par_scales: ndarray[tuple[Any, ...], dtype[float]]

Return the scales of the parameters and the cost function.

solve_linear(*freqs*, *ndofs*, *amplitudes*, *weights*, *nonlinear_pars*)

Use linear linear regression to solve a subset of the parameters.

This is a specialized implementation that rewrites the problem in a different form to solve all parameters with a linear regression.

Return type

ndarray[tuple[Any, ...], dtype[float]]

Parameters

- **freqs** (*ndarray[tuple[Any, ...], dtype[float]]*)
- **ndofs** (*ndarray[tuple[Any, ...], dtype[float]]*)
- **amplitudes** (*ndarray[tuple[Any, ...], dtype[float]]*)
- **weights** (*ndarray[tuple[Any, ...], dtype[float]]*)
- **nonlinear_pars** (*ndarray[tuple[Any, ...], dtype[float]]*)

class LorentzModel(***, *ratio_weight=1.0*, *ratio_threshold=100.0*)

Bases: *PadeModel*

A model for the spectrum with a Lorentzian peak at zero frequency plus some white noise.

This is a special case of the PadeModel with `numer_degrees = [0, 2]` and `denom_degrees = [2]`. Furthermore, it will only accept parameters that correspond to a well-defined exponential

correlation time.

For too small cutoffs (covering only the peak of the Lorentzian and not its decay), the estimates of the Lorentzian width, and consequently the exponential correlation time, become statistically unreliable. In this regime, the assumption of maximum a posteriori probability (MAP), on which STACIE relies to fit the model and estimate parameter uncertainties, also breaks down. Unreliable MAP estimates are inferred from the relative error of the exponential correlation time divided by the relative error of the autocorrelation integral. This implementation uses the ratio in two ways:

1. When the ratio exceeds a predefined threshold (default value 100), the cutoff criterion is set to infinity.
2. If the ratio remains below this threshold, the ratio times a weight (default value 1.0) is added to the cutoff criterion.

Note that this is an empirical penalty to mitigate MAP-related issues. Because the penalty is expressed as a ratio of relative errors, it is dimensionless and insensitive to the overall uncertainty of the spectrum.

The hyperparameters `ratio_weight` and `ratio_threshold` may be tuned to adjust the sensitivity of the heuristic, but it is recommended to keep their default values.

> **Parameters**
>
> - **ratio_weight** (*float*)
> - **ratio_threshold** (*float*)

denom_degrees: ndarray[tuple[Any, ...], dtype[int]]
: The degrees of the monomials in the denominator.

  Note that the leading term is always 1, and there is no need to include degree zero.

derive_props(*props*)
: Add the autocorrelation integral (and other properties) derived from the parameters.

  The exponential correlation time is derived from the parameters, if the fitted model has a maximum at zero frequency. If not, the "criterion" is set to infinity and the "msg" is set accordingly, to discard the current fit from the average over the cutoff frequencies.

  > **Parameters**
  > **props** (*dict[str, ndarray[tuple[Any, ...], dtype[float]]]*)

property name

numer_degrees: ndarray[tuple[Any, ...], dtype[int]]
: The degrees of the monomials in the numerator.

ratio_threshold: float
: A threshold for the ratio of relative errors used to set the cutoff criterion to Inf.

ratio_weight: float
: The penalty for the cutoff criterion is this weight times the ratio of relative errors.

class PadeModel(*numer_degrees*, *denom_degrees*)

Bases: *SpectrumModel*

A rational function model for the spectrum, a.k.a. a Padé approximation.

> **Parameters**
>
> - **numer_degrees** (*ndarray[tuple[Any, ...], dtype[int]]*)
> - **denom_degrees** (*ndarray[tuple[Any, ...], dtype[int]]*)

bounds()

Return parameter bounds for the optimizer.

> **Return type**
> list[tuple[float, float]]

compute(*freqs*, *pars*, ***, *deriv=0*)

See *SpectrumModel.compute()*.

> **Return type**
> list[ndarray[tuple[Any, ...], dtype[float]]]
>
> **Parameters**
>
> - **freqs** (*ndarray[tuple[Any, ...], dtype[float]]*)
> - **pars** (*ndarray[tuple[Any, ...], dtype[float]]*)
> - **deriv** (*int*)

denom_degrees: ndarray[tuple[Any, ...], dtype[int]]

The degrees of the monomials in the denominator.

Note that the leading term is always 1, and there is no need to include degree zero.

derive_props(*props*)

Add the autocorrelation integral (and other properties) derived from the parameters.

> **Parameters**
> **props** (*dict[str, ndarray[tuple[Any, ...], dtype[float]]]*)

get_par_nonlinear()

Return a boolean mask for the nonlinear parameters.

> **Return type**
> ndarray[tuple[Any, ...], dtype[bool]]

property name

property npar
Return the number of parameters.

numer_degrees: ndarray[tuple[Any, ...], dtype[int]]
The degrees of the monomials in the numerator.

property par_scales: ndarray[tuple[Any, ...], dtype[float]]
Return the scales of the parameters and the cost function.

solve_linear(*freqs*, *ndofs*, *amplitudes*, *weights*, *nonlinear_pars*)
Use linear linear regression to solve a subset of the parameters.

This is a specialized implementation that rewrites the problem in a different form to solve all parameters with a linear regression.

> **Return type**
> ndarray[tuple[Any, ...], dtype[float]]
>
> **Parameters**
> - **freqs** (*ndarray[tuple[Any, ...], dtype[float]]*)
> - **ndofs** (*ndarray[tuple[Any, ...], dtype[float]]*)
> - **amplitudes** (*ndarray[tuple[Any, ...], dtype[float]]*)
> - **weights** (*ndarray[tuple[Any, ...], dtype[float]]*)
> - **nonlinear_pars** (*ndarray[tuple[Any, ...], dtype[float]]*)

class SpectrumModel
Bases: object

Abstract base class for spectrum models.

Subclasses must override all methods that raise NotImplementedError.

The first parameter must have a property that is used when constructing an initial guess: When the first parameter increases, the model should increase everywhere, and must allow for an arbitrary increase of the spectrum at all points. This is used to repair initial guesses that result in a partially negative spectrum.

bounds()
Return parameter bounds for the optimizer.

> **Return type**
> list[tuple[float, float]]

compute(*freqs*, *pars*, ***, *deriv=0*)
Compute the amplitudes of the spectrum model.

> **Parameters**
> - **freqs** (ndarray[tuple[Any, ...], dtype[float]]) – The frequencies for which the

model spectrum amplitudes are computed.

- **pars** (ndarray[tuple[Any, ...], dtype[float]]) – The parameter vector. For vectorized calculations, the last axis corresponds to the parameter index.
- **deriv** (int) – The maximum order of derivatives to compute: 0, 1 or 2.

Return type
list[ndarray[tuple[Any, ...], dtype[float]]]

Returns

results – A results list, index corresponds to order of derivative. The shape of the arrays in the results list is as follows:

- For `deriv=0`, the shape is `(*vec_shape, len(freqs))`.
- For `deriv=1`, the shape is `(*vec_shape, len(pars), len(freqs))`.
- For `deriv=2`, the shape is `(*vec_shape, len(pars), len(pars), len(freqs))`

If some derivatives are independent of the parameters, broadcasting rules may be used to reduce the memory footprint. This means that `vec_shape` may be replaced by a tuple of ones with the same length.

configure_scales(*timestep*, *freqs*, *amplitudes*)

Store essential scale information in the `scales` attribute.

Other methods may access this information, so this method should be called before performing any computations.

Return type
ndarray[tuple[Any, ...], dtype[float]]

Parameters

- **timestep** (*float*)
- **freqs** (*ndarray[tuple[Any, ...], dtype[float]]*)
- **amplitudes** (*ndarray[tuple[Any, ...], dtype[float]]*)

derive_props(*props*)

Add the autocorrelation integral (and other properties) derived from the parameters.

Parameters
props (dict[str, ndarray[tuple[Any, ...], dtype[float]]]) – The properties dictionary, including the parameters and their uncertainties. Subclasses may add additional properties to this dictionary.

get_par_nonlinear()

Return a boolean mask for the nonlinear parameters.

The returned parameters cannot be solved with the solve_linear method. Models are free to decide which parameters can be solved with linear regression. For example, some non-linear parameters may be solved with a linear regression after rewriting the regression problem in a different form.

Return type
ndarray[tuple[Any, ...], dtype[bool]]

property name

neglog_prior(*pars*, ***, *deriv=0*)

Minus logarithm of the prior probability density function, if any.

Subclasses may implement (a very weak) prior, if any.

Return type
list[ndarray[tuple[Any, ...], dtype[float]]]

Parameters

- **pars** (*ndarray[tuple[Any, ...], dtype[float]]*)
- **deriv** (*int*)

property npar

Return the number of parameters.

property par_scales: ndarray[tuple[Any, ...], dtype[float]]

Return the scales of the parameters and the cost function.

sample_nonlinear_pars(*rng*, *budget*)

Return samples of the nonlinear parameters.

Parameters

- **rng** (Generator) – The random number generator.
- **budget** (int) – The number of samples to generate.
- **freqs** – The frequencies for which the model spectrum amplitudes are computed.
- **par_scales** – The scales of the parameters and the cost function.

Return type
ndarray[tuple[Any, ...], dtype[float]]

Returns
samples – The samples of the nonlinear parameters, array with shape `(budget, num_nonlinear)`, where `num_nonlinear` is the number of nonlinear parameters.

scales: dict[str, float]

A dictionary with essential scale information for the parameters and the cost function.

solve_linear(*freqs*, *ndofs*, *amplitudes*, *weights*, *nonlinear_pars*)

Use linear linear regression to solve a subset of the parameters.

The default implementation in the base class assumes that the linear parameters are genuinely linear without rewriting the regression problem in a different form.

Parameters

- **freqs** (ndarray[tuple[Any, ...], dtype[float]]) – The frequencies for which the model spectrum amplitudes are computed.
- **amplitudes** (ndarray[tuple[Any, ...], dtype[float]]) – The amplitudes of the spectrum.
- **ndofs** (ndarray[tuple[Any, ...], dtype[float]]) – The number of degrees of freedom at each frequency.
- **weights** (ndarray[tuple[Any, ...], dtype[float]]) – Fitting weights, in range [0, 1], to use for each grid point.
- **nonlinear_pars** (ndarray[tuple[Any, ...], dtype[float]]) – The values of the nonlinear parameters for which the basis functions are computed.

Return type
ndarray[tuple[Any, ...], dtype[float]]

Returns

- *linear_pars* – The solved linear parameters.
- *amplitudes_model* – The model amplitudes computed with the solved parameters.

valid(*pars*)

Return True when the parameters are within the feasible region.

Return type
bool

Parameters
pars (*ndarray[tuple[Any, ...], dtype[float]]*)

which_invalid(*pars*)

Return a boolean mask for the parameters outside the feasible region.

Return type
ndarray[tuple[Any, ...], dtype[bool]]

convert_pade022_lorentz(*pars*, *covar*)

Convert parameters and covariance from Pade(0,2;2) to Lorentz model.

Parameters

- **pars** (ndarray[tuple[Any, ...], dtype[float]]) – The parameters of the Pade(0,2;2) model.
- **covar** (ndarray[tuple[Any, ...], dtype[float]]) – The covariance matrix of the Pade(0,2;2) model.

Return type
tuple[ndarray[tuple[Any, ...], dtype[float]], ndarray[tuple[Any, ...], dtype[float]]]

Returns

- *pars_lorentz* – The parameters of the Lorentz model.
- *pars_lorentz_covar* – The covariance matrix of the Lorentz model.

guess(*freqs*, *ndofs*, *amplitudes*, *weights*, *model*, *rng*, *nonlinear_budget*)

Guess initial values of the parameters for a model.

Parameters

- **freqs** (ndarray[tuple[Any, ...], dtype[float]]) – The frequencies for which the model spectrum amplitudes are computed.
- **ndofs** (ndarray[tuple[Any, ...], dtype[float]]) – The number of degrees of freedom at each frequency.
- **amplitudes** (ndarray[tuple[Any, ...], dtype[float]]) – The amplitudes of the spectrum.
- **weights** (ndarray[tuple[Any, ...], dtype[float]]) – Fitting weights, in range [0, 1], to use for each grid point.
- **model** (*SpectrumModel*) – The model for which the parameters are guessed.
- **rng** (Generator) – The random number generator.
- **nonlinear_budget** (int) – The number of samples of the nonlinear parameters is computed as `nonlinear_budget ** num_nonlinear`, where `num_nonlinear` is the number of nonlinear parameters.

Returns

pars – An initial guess of the parameters.

stacie.plot module

Plot various aspects of the results of the autocorrelation integral estimate.

fixformat(*s*)

Replace standard scientific notation with prettier unicode formatting.

Return type

str

Parameters

s (str)

plot_acint_estimates(*ax*, *uc*, *rs*)

Plot the sorted autocorrelation integral estimates and their uncertainties.

Parameters

- **ax** (*Axes*)
- **uc** (UnitConfig)

- **rs** (*list[Result]*)

plot_all_models(*ax*, *uc*, *r*)

Plot all fitted model spectra (for all tested cutoffs).

Parameters

- **ax** (*Axes*)
- **uc** (UnitConfig)
- **r** (Result)

plot_cutoff_weight(*ax*, *uc*, *r*)

Plot the cutoff criterion as a function of cutoff frequency.

Parameters

- **ax** (*Axes*)
- **uc** (UnitConfig)
- **r** (Result)

plot_evals(*ax*, *uc*, *r*)

Plot the eigenvalues of the Hessian as a function of the cutoff frequency.

Parameters

- **ax** (*Axes*)
- **uc** (UnitConfig)
- **r** (Result)

plot_extras(*axs*, *uc*, *r*)

Parameters

- **axs** (*ndarray[tuple[Any, ...], dtype[Axes]]*)
- **uc** (UnitConfig)
- **r** (Result)

plot_fitted_spectrum(*ax*, *uc*, *r*, ***, *legend=True*)

Plot the fitted model spectrum.

Parameters

- **ax** (*Axes*)
- **uc** (UnitConfig)

- **r** (Result)
- **legend** (*bool*)

plot_qq(*ax*, *uc*, *rs*)

Make a qq-plot between the predicted and expected distribution of AC integral estimates.

This plot function assumes the true integral is known.

Parameters

- **ax** (*Axes*)
- **uc** (UnitConfig)
- **rs** (*list*[Result])

plot_results(*path_pdf*, *rs*, *uc=None*, ***, *figsize=(7.5, 4.21875)*, *legend=True*)

Generate a multi-page PDF with plots of the autocorrelation integral estimation.

Parameters

- **path_pdf** (str) – The PDF file where all the figures are stored.
- **rs** (*Result* | list[*Result*]) – A single `Result` instance or a list of them. If the (first) result instance has `spectrum.amplitudes_ref` set, theoretical expectations are included. When multiple results instances are given, only the first one is plotted in blue. All remaining ones are plotted in light grey.
- **uc** (*UnitConfig* | None) – The configuration of the units used for plotting.
- **figsize** (tuple) – The figure size tuple for matplotlib
- **legend** (*bool*)

plot_sanity(*ax*, *uc*, *r*)

Parameters

- **ax** (*Axes*)
- **uc** (UnitConfig)
- **r** (Result)

plot_spectrum(*ax*, *uc*, *s*, *nplot=None*)

Plot the empirical spectrum.

Parameters

- **ax** (*Axes*)
- **uc** (UnitConfig)
- **s** (Spectrum)

- **nplot** (*int | None*)

plot_uncertainty(*ax*, *uc*, *r*)

Plot the autocorrelation integral and uncertainty as a function fo cutoff frequency.

Parameters

- **ax** (*Axes*)
- **uc** (UnitConfig)
- **r** (Result)

rms(*x*)

stacie.spectrum module

Utility to prepare the spectrum and other inputs for given sequences.

class Spectrum(*mean*, *variance*, *timestep*, *nstep*, *freqs*, *ndofs*, *amplitudes*, *amplitudes_ref=None*)

Bases: object

Container class holding all the inputs for the autocorrelation integral estimate.

Parameters

- **mean** (*float*)
- **variance** (*float*)
- **timestep** (*float*)
- **nstep** (*int*)
- **freqs** (*ndarray[tuple[Any, ...], dtype[float]]*)
- **ndofs** (*ndarray[tuple[Any, ...], dtype[float]]*)
- **amplitudes** (*ndarray[tuple[Any, ...], dtype[float]]*)
- **amplitudes_ref** (*ndarray[tuple[Any, ...], dtype[float]] | None*)

amplitudes: ndarray[tuple[Any, ...], dtype[float]]

The spectrum amplitudes averaged over the given input sequences.

amplitudes_ref: ndarray[tuple[Any, ...], dtype[float]] | None

Optionally, the known analytical model of the power spectrum, on the same frequency grid.

freqs: ndarray[tuple[Any, ...], dtype[float]]

The equidistant frequency axis of the spectrum.

mean: float

The mean of the input sequences multiplied by the square root of the prefactor.

ndofs: `ndarray[tuple[Any, ...], dtype[float]]`

The number of independent contributions to each amplitude.

For the DC and Nyquist components (for even `nstep`), this is equal to the number of independent time series (`nindep`). For all other frequencies, this is `2 * nindep`.

property nfreq: `int`

The number of RFFT frequency grid points.

nstep: `int`

The number of time steps in the input sequences.

If the time series are given as an array with shape `(nindep, nstep)`, this corresponds to the size of the second dimension.

timestep: `float`

The time between two subsequent elements in the input sequences.

variance: `float`

The variance of the input sequences multiplied by the prefactor.

without_zero_freq()

Return a copy without the DC component.

> **Return type**
> `Self`

compute_spectrum(*sequences*, ***, *prefactors=1.0*, *timestep=1*, *include_zero_freq=True*)

Compute a spectrum and return it as a *Spectrum* object.

The spectrum amplitudes are computed as follows:

$$C_k = \frac{1}{M} \sum_{m=1}^{M} \frac{F_m h}{2N} \left| \sum_{n=0}^{N-1} x_n^{(m)} \exp\left(-i \frac{2\pi n k}{N}\right) \right|^2$$

where:

- F_m is the given prefactor (may be different for each sequence),
- h is the timestep,
- N is the number of time steps in the input sequences,
- M is the number of independent sequences,
- $x_n^{(m)}$ is the value of the m-th sequence at time step n,
- k is the frequency index.

The sum over m simply averages spectra obtained from different sequences. The factor $F_m h / 2N$ normalizes the spectrum so that its zero-frequency limit is an estimate of the auto-correlation integral.

> **Parameters**
>
> - **sequences** (`Iterable[ndarray[tuple[Any, ...], dtype[float]]] | ndarray[tuple[Any, ...], dtype[float]]`) – The input sequences, which can have several forms. If `prefactors` is not `None`, it can be:

- An array with shape `(nindep, nstep)` or `(nstep,)`. In case of a 2D array, each row is a time-dependent sequence. In case of a 1D array, a single sequence is used.
- An iterable whose items are arrays as described in the previous point. This option is convenient when a single array does not fit in memory.

If `prefactors` is `None`:

- A tuple of a prefactor (or an array of prefactors) and a sequences array, either 1D or 2D as described above.
- An iterable whose items are tuples of a prefactor (or an array of prefactors) and a sequences array, either 1D or 2D as described above.

All sequences are assumed to be statistically independent and have length `nstep`. (Time correlations within one sequence are fine, obviously.) We recommend using multiple independent sequences to reduce uncertainties. Arrays must be used. (lists of floating point values are not supported.)

- **prefactors** (`Iterable[ndarray[tuple[Any, ...], dtype[float]]] | ndarray[tuple[Any, ...], dtype[float]] | None`) – A positive factor to be multiplied with the autocorrelation function to give it a physically meaningful unit. This argument can be given in multiple forms:
  - None, in which case the sequences are assumed to be one or more (prefactors, sequences) tuples.
  - A single floating point value: the same prefactor is used for all input sequences.
  - A single array with shape `(nindep,)`: each sequence is multiplied with the corresponding prefactor.
  - An iterable whose items are of the form described in the previous two points. In this case, the sequences must also be given as an iterable with the same length.
- **timestep** (`float`) – The time step of the input sequence.
- **include_zero_freq** (`bool`) – When set to False, the DC component of the spectrum is discarded.

Return type
Spectrum

Returns
spectrum – A *Spectrum* object holding all the inputs needed to estimate the integral of the autocorrelation function. This can be used as input to *stacie.estimate.estimate_acint()*.

stacie.synthetic module

Generate synthetic time-correlated data for algorithmic testing and validation.

generate(*psd*, *timestep*, *nseq*, *nstep=None*, *rng=None*)

Generate sequences with a given power spectral density.

Parameters

- **psd** (`ndarray[tuple[Any, ...], dtype[float]]`) – The power spectral density.

The normalization of the PSD is consistent `compute_spectrum` when using `prefactors=2.0` and the given `timestep` as arguments. The empirical amplitudes of the spectrum will then be consistent with given PSD. Hence `psd[0]` is the ground truth of the autocorrelation integral.

- **timestep** (`float`) – The time between two subsequent elements in the sequence.
- **nseq** (`int`) – The number of sequences to generate.
- **nstep** (`int | None`) – The number of time steps in each sequence. When not given, the number of steps is `2 * (len(psd) - 1)`. This argument can be used to truncate the sequences, which can be useful for creating aperiodic signals.
- **rng** (`Generator | None`) – The random number generator.

Return type

`ndarray[tuple[Any, ...], dtype[float]]`

Returns

sequences – The generated sequences, a 2D array with shape `(nseq, nstep)`, where `nstep = 2 * (len(psd) - 1)` if not provided.

stacie.utils module

Utilities for preparing inputs.

exception PositiveDefiniteError

Bases: `ValueError`

Raised when a matrix is not positive definite.

class UnitConfig(**, acint_symbol='\\\\mathcal{I}', acint_unit_str='', acint_unit=1.0, acint_fmt='.2e', freq_unit_str='', freq_unit=1.0, freq_fmt='.2e', time_unit_str='', time_unit=1.0, time_fmt='.2e', clevel=0.95*)

Bases: `object`

Unit configuration for functions that print or plot values.

This class never influences numerical values in STACIE's computations, such as attributes of a result object or other variables. It only affects printed or plotted values.

The `acint_unit`, `freq_unit`, and `time_unit` attributes should be set as follows:

- The values of variables in STACIE (and your scripts using STACIE) are always in "internal units".
- The `*_unit` attributes are assumed to have the value of a "display unit" in the same internal units.

For example, if your internal time unit is 1 ps and you want times to be reported in ns, set `time_unit = 1000.0`, because your display unit (1 ns) is 1000 internal units (1 ps).

To make these conventions easy to follow (and to avoid unit hell in general), it is recommended to pick consistent internal units for your system. For example, use atomic units or SI units throughout your code:

- As soon as you load data from a file, immediately convert it to internal units.
- Only just before printing or plotting, convert to display units, which is also how this class is used in STACIE.

For example, when all variables are in SI base units and you want to display time in ns, frequency in THz, and autocorrelation integrals in cm^2/s, then create a *UnitConfig* as follows:

```
units = UnitConfig(
    time_unit=1e-9,
    time_unit_str="ns",
    freq_unit=1e12,
    freq_unit_str="THz",
    acint_unit=1e-4,
    acint_unit_str="cm^2/s",
)
```

> **Parameters**
>
> - **acint_symbol** (*str*)
> - **acint_unit_str** (*str*)
> - **acint_unit** (*float*)
> - **acint_fmt** (*str*)
> - **freq_unit_str** (*str*)
> - **freq_unit** (*float*)
> - **freq_fmt** (*str*)
> - **time_unit_str** (*str*)
> - **time_unit** (*float*)
> - **time_fmt** (*str*)
> - **clevel** (*float*)

acint_fmt: str
The format string for an autocorrelation integral.

acint_symbol: str
The symbol used for the autocorrelation integral.

acint_unit: float
The unit of an autocorrelation integral.

acint_unit_str: str
The text used for the autocorrelation integral unit.

property clb: float
The confidence lower bound used to plot confidence intervals.

clevel: float
The confidence level used to plot confidence intervals.

property cub: float
The confidence upper bound used to plot confidence intervals.

freq_fmt: str
The format string for a frequency.

freq_unit: float
The unit of frequency.

freq_unit_str: str
The text used for the frequency unit.

time_fmt: str
The format string for a time value.

time_unit: float
The unit of time.

time_unit_str: str
The text used for the time unit.

block_average(*sequences*, *size*)
Reduce input sequences by taking block averages.

This reduces the maximum frequency of the frequency axis of the spectrum, which may be useful when the time step is much shorter than the exponential autocorrelation time.

A time step $h = \tau_{\text{exp}}/(20\pi)$ (after taking block averages) is recommended, not larger.

Parameters
- **sequences** (ndarray[tuple[Any, ...], dtype[float]]) – Input sequence(s) to be block averaged, with shape `(*data_shape, nstep)`, where `data_shape` represents any number of leading dimensions. A single sequence with shape `(nstep, )` is also accepted.
- **size** (int) – The block size

Return type
ndarray[tuple[Any, ...], dtype[TypeVar(_ScalarT, bound= generic)]]

Returns
blav_sequences – Sequences of block averages, with shape `(*data_shape, nstep // size)`. If needed a few trailing elements of `sequences` are discarded to make `nstep` divisible by `size`.

label_unit(*label*, *unit_str*)
Format a label with the unit string as `label [unit]`.

When the unit is `""` or `None`, the unit is omitted.

Parameters
- **label** (str) – The label text.
- **unit_str** (str | None) – The unit string.

Return type
str

mixture_stats(*means*, *covars*, *weights*)
Compute the statistics of the (Gaussian) mixture distribution.

Parameters

- **means** (ndarray[tuple[Any, ...], dtype[float]]) – The means of the mixture components. Weighted averages are taken over the first index. Shape is `(ncomp, nfeature)` or `(ncomp,)`. If the shape is `(ncomp,)`, the means are interpreted as scalars. If the shape is `(ncomp, nfeature)`, the means are interpreted as vectors.
- **covars** (ndarray[tuple[Any, ...], dtype[float]]) – The covariances of the mixture components. If the shape matches that of the `means` argument, this array is interpreted as a diagonal covariance matrix. If the shape is `(ncomp, nfeature, nfeature)`, this array is interpreted as full covariance matrices.
- **weights** (ndarray[tuple[Any, ...], dtype[float]]) – The weights of the mixture components. Shape is `(ncomp,)`. The weights are normalized to sum to 1.

Returns

- *mean* – The mean of the mixture distribution. Shape is `(nfeature,)`.
- *covar* – If the input covariance matrix is diagonal, the output covariance matrix is also diagonal and has shape `(nfeature,)`. If the input covariance matrix is full, the output covariance matrix is also full and has shape `(nfeature, nfeature)`.

robust_dot(*scales*, *evals*, *evecs*, *other*)

Compute the dot product of a robustly diagonalized matrix with another matrix.

- The first three arguments are the output of *robust_posinv()*.
- To multiply with the inverse, just use element-wise inversion of `scales` and `evals`.

Parameters

- **scales** – The scales used to precondition the matrix.
- **evals** – The eigenvalues of the preconditioned matrix.
- **evecs** – The eigenvectors of the preconditioned matrix.
- **other** – The other matrix to be multiplied. 1D or 2D arrays are accepted.

Returns

result – The result of the dot product.

robust_posinv(*matrix*)

Compute the eigenvalues, eigenvectors and inverse of a positive definite symmetric matrix.

This function is a robust replacement for `numpy.linalg.eigh()` and `numpy.linalg.inv()` that can handle large variations in order of magnitude of the diagonal elements. If the matrix is not positive definite, a `ValueError` is raised.

Parameters

matrix (ndarray[tuple[Any, ...], dtype[float]]) – Input matrix to be diagonalized.

Return type

tuple[ndarray[tuple[Any, ...], dtype[TypeVar(_ScalarT, bound= generic)]], ndarray[tuple[Any, ...], dtype[TypeVar(_ScalarT, bound= generic)]], ndarray[tuple[Any, ...], dtype[TypeVar(_ScalarT, bound= generic)]], ndarray[tuple[Any, ...], dtype[TypeVar(_ScalarT, bound= generic)]]]

Returns

- *scales* – The scales used to precondition the matrix.

- *evals* – The eigenvalues of the preconditioned matrix.
- *evecs* – The eigenvectors of the preconditioned matrix.
- *inverse* – The inverse of the original.

split(*sequences*, *nsplit*)

Split input sequences into shorter parts of equal length.

This reduces the resolution of the frequency axis of the spectrum, which may be useful when the sequence length is much longer than the exponential autocorrelation time.

Parameters

- **sequences** (ndarray[tuple[Any, ...], dtype[float]]) – Input sequence(s) to be split, with shape `(nseq, nstep)`. A single sequence with shape `(nstep, )` is also accepted.
- **nsplit** (int) – The number of splits.

Return type

ndarray[tuple[Any, ...], dtype[TypeVar(_ScalarT, bound= generic)]]

Returns

split_sequences – Splitted sequences, with shape `(nseq * nsplit, nstep // nsplit)`.

8.1.2 Module contents

The STACIE package.

CHAPTER 9

Development

This section contains some technical details about the development of STACIE.

9.1 Contributor Guide

> This contributor guide is created with the following template: nayafia/contributing-template.

First of all, thank you for considering contributing to STACIE! STACIE is being developed by academics who also have many other responsibilities, and you are probably in a similar situation. The purpose of this guide is to make efficient use of everyone's time.

STACIE has already been used for production simulations, but we are always open to (suggestions for) improvements that fit within the goals of STACIE. New worked examples that are not too computationally demanding are also highly appreciated! Even simple things like correcting typos or fixing minor mistakes are welcome.

This section does not document how to use of Git and GitHub, or how to develop software in general. We assume that you already have the basic skills to contribute. Below are some links to documentation for those who are not familiar with these technicalities yet.

9.1.1 Ground Rules

- We want everyone to have a positive experience with their (online) interactions related to STACIE's development. Our expectations for (online) communication are outlined in the *Code of Conduct*.
- Except for minor corrections, we encourage you to open a GitHub issue before making changes to the source code. A transparent discussion before making changes can save a lot of time. Also if you have found a potential problem but are not sure how to fix it, we encourage you to open an issue.
- When you contribute, you accept that your contributions will be distributed under the same *licenses* that we currently use for source code and documentation.

9.1.2 How to Report a Bug

Create a new issue (or find an existing one) and include the following information:

1. What version of STACIE, Python and NumPy are you using?
2. What operating system and processor architecture are you using?
3. What did you do?
4. What did you expect to see?
5. What did you see instead?

9.1.3 First-Time Contributors

If you have never contributed to an open source project before, you may find the following online references helpful:

- http://makeapullrequest.com/
- http://www.firsttimersonly.com/
- https://egghead.io/series/how-to-contribute-to-an-open-source-project-on-github

If something goes wrong in the process of creating a pull request, we'll try to help out.

9.1.4 Contribution Workflow

Contributing to STACIE always involves the following steps:

1. Create an issue on GitHub to discuss your plans.
2. Fork the STACIE repository on GitHub.
3. Clone the original repository on your computer and add your fork as a second remote.
4. Install pre-commit.
5. Create a new branch. (Do not commit changes to the main branch.)
6. Make changes to the source code. New features must have unit tests and documentation.
7. Make sure all the tests pass, the documentation builds without errors or warnings, and pre-commit reports no problems.
8. Push your branch to your fork and Create a pull request on GitHub. In the pull request message (not the title), mention which issue the pull request addresses.
9. Wait for your changes to be reviewed and handle all requests for improvements during the review process.
10. If your change is accepted, it will be merged into the main branch and included in the next release of STACIE.

9.2 Development Setup

9.2.1 Repository, Tests and Documentation Build

It is assumed that you have previously installed Python, Git, pre-commit and direnv. A local installation for testing and development can be installed as follows:

```
git clone git@github.com:molmod/stacie.git
cd stacie
pre-commit install
python -m venv venv
echo 'source venv/bin/activate' > .envrc
direnv allow
pip install -U pip
pip install -e .[docs,tests]
pytest -vv
cd docs
./compile_html.sh
./compile_pdf.sh
```

9.2.2 Documentation Live Preview

The documentation is created using Sphinx.

Edit the documentation Markdown files with a live preview by running the following command *in the root* of the repository:

```
cd docs
./preview_html.sh
```

Keep this running. This will print a URL in the terminal that you open in your browser to preview the documentation. Now you can edit the documentation and see the result as soon as you save a file.

Please, use Semantic Line Breaks as it facilitates reviewing documentation changes.

9.3 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Effort-based Versioning.

9.3.1 Unreleased

(no changes yet)

9.3.2 1.2.1 - 2025-12-28

Minor documentation and dependency improvements.

Changed

- Several documentation improvements and clarifications.
- Removed unused dependency `cattrs` and lowered minimum required of `attrs` to `23.1.0` to facilitate installation in more constrained environments.

9.3.3 1.2.0 - 2025-12-23

Reformulation of the Lorentz model parameters.

Changed

- New parametrization of the linear coefficients in the Lorentz model: C_0 and C_1 instead of A and B.
- Documentation improvements and clarifications.

9.3.4 1.1.3 - 2025-11-19

Support for Python 3.10

Changed

- Add support for Python 3.10

9.3.5 1.1.2 - 2025-11-19

Additional packaging cleanups.

Changed

- Removed a few more files from the source package that could cause confusion.
- Removed an unused testing dependency.

9.3.6 1.1.1 - 2025-11-18

Smaller source package (10x size reduction) and dependency simplification.

Changed

- PyPI Package size reduction by a factor 10. Only essential files are now included in the package.
- Remove path dependency to facilitate packaging and distribution.

9.3.7 1.1.0 - 2025-11-10

Improved Lorentz model, with a more robust estimate of the exponential correlation time and its uncertainty. Improved examples.

Added

- Cloud cover example
- Comparison between STACIE's diffusion coefficient and a more conventional MSD analysis in the surface diffusion example.

Changed

- The license of the documentation has been updated to a choice of license (`CC-BY-SA-4.0 OR LGPL-3.0-or-later`). STACIE's code is still distributed under a single `LGPL-3.0-or-later` license.
- The penalty in the `LorentzModel` has been improved to exclude and down-weight results from frequency cutoffs with a high relative error of the estimated exponential correlation time. In this regime, the maximum a posteriori estimate of uncertainty is not reliable.

Fixed

- Several documentation improvements:
  - Clarify how the derivation of block-averaged velocities for diffusion and electrical conductivity, using atomic positions (or dipole vectors) as input.
  - Improve explanation on discarding the DC-component of the spectrum.
  - Add more helpful comments on how to deal with unit conversion.
  - Fixed a typo in the equation of the marginalization weights.
- Repocard images were added.
- Dataset metadata improvements.
- Several other minor issues in documentation and tooling were fixed.

9.3.8 1.0.0 - 2025-06-26

This is the first stable release of STACIE!

Changed

- Metadata and citation updates

9.3.9 1.0.0rc1 - 2025-06-25

This is the first release candidate of STACIE, with a final release expected very soon. The main remaining issues are related to (back)linking of external resources in the documentation and README files.

9.4 How to Make a Release

9.4.1 Software packaging and deployment

To make a new release of STACIE on PyPI, take the following steps

- Mark the release in `docs/changelog.md`.
- Make a new commit and tag it with `vX.Y.Z`.
- Trigger the PyPI GitHub Action: `git push origin main --tags`.

After having verified that the PyPI release was successful, update the feedstock on `conda-forge` accordingly: https://github.com/conda-forge/stacie-feedstock.

9.4.2 Documentation build and deployment

Take the following steps, starting from the root of the repository:

```
cd docs
./release_docs.sh
```

Use this script with caution, as it will push changes to the `gh-pages` branch.

CHAPTER 10

Code of Conduct

10.1 Our Pledge

We as members, contributors, and leaders pledge to make participation in our community a harassment-free experience for everyone, regardless of age, body size, visible or invisible disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, caste, color, religion, or sexual identity and orientation.

We pledge to act and interact in ways that contribute to an open, welcoming, diverse, inclusive, and healthy community.

10.2 Our Standards

Examples of behavior that contributes to a positive environment for our community include:

- Demonstrating empathy and kindness toward other people
- Being respectful of differing opinions, viewpoints, and experiences
- Giving and gracefully accepting constructive feedback
- Accepting responsibility and apologizing to those affected by our mistakes, and learning from the experience
- Focusing on what is best not just for us as individuals, but for the overall community

Examples of unacceptable behavior include:

- The use of sexualized language or imagery, and sexual attention or advances of any kind
- Trolling, insulting or derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or email address, without their explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

10.3 Enforcement Responsibilities

Community leaders are responsible for clarifying and enforcing our standards of acceptable behavior and will take appropriate and fair corrective action in response to any behavior that they deem inappropriate, threatening, offensive, or harmful.

Community leaders have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, and will communicate reasons for moderation decisions when appropriate.

10.4 Scope

This Code of Conduct applies within all community spaces, and also applies when an individual is officially representing the community in public spaces. Examples of representing our community include using an official email address, posting via an official social media account, or acting as an appointed representative at an online or offline event.

10.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported to the community leaders responsible for enforcement at Toon.Verstraelen@UGent.be All complaints will be reviewed and investigated promptly and fairly.

All community leaders are obligated to respect the privacy and security of the reporter of any incident.

10.6 Enforcement Guidelines

Community leaders will follow these Community Impact Guidelines in determining the consequences for any action they deem in violation of this Code of Conduct:

10.6.1 1. Correction

Community Impact: Use of inappropriate language or other behavior deemed unprofessional or unwelcome in the community.

Consequence: A private, written warning from community leaders, providing clarity around the nature of the violation and an explanation of why the behavior was inappropriate. A public apology may be requested.

10.6.2 2. Warning

Community Impact: A violation through a single incident or series of actions.

Consequence: A warning with consequences for continued behavior. No interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, for a specified period of time. This includes avoiding interactions in community spaces as well as external channels like social media. Violating these terms may lead to a temporary or permanent ban.

10.6.3 3. Temporary Ban

Community Impact: A serious violation of community standards, including sustained inappropriate behavior.

Consequence: A temporary ban from any sort of interaction or public communication with the community for a specified period of time. No public or private interaction with the people involved, including unsolicited interaction with those enforcing the Code of Conduct, is allowed during this period. Violating these terms may lead to a permanent ban.

10.6.4 4. Permanent Ban

Community Impact: Demonstrating a pattern of violation of community standards, including sustained inappropriate behavior, harassment of an individual, or aggression toward or disparagement of classes of individuals.

Consequence: A permanent ban from any sort of public interaction within the community.

10.7 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 2.1, available at https://www.contributor-covenant.org/version/2/1/code_of_conduct.html.

Community Impact Guidelines were inspired by Mozilla's code of conduct enforcement ladder.

For answers to common questions about this code of conduct, see the FAQ at https://www.contributor-covenant.org/faq. Translations are available at https://www.contributor-covenant.org/translations.

Python Module Index

S

Index

Non-alphabetical

A

B

C

D

E

F

G

H

L

M

N

P

R

S

T

U

V

W