TorchNISE

Release 0.1.0

Yannick Holkamp, Emiliano Godinez, Ulrich Kleinekathöfer

Apr 16, 2025

CONTENTS:

Add your content using `reStructuredText` syntax. See the reStructuredText documentation for details.

CHAPTER

ONE

TORCHNISE

1.1 torchnise package

1.1.1 Submodules

1.1.2 torchnise.absorption module

This module provides functions to compute time domain absorption and convert it to an absorption spectrum using FFT. It allows for optional damping to simulate signal decay over time.

torchnise.absorption.**absorb_time_to_freq**(*absorb_time*, *config*)

Convert time domain absorption to an absorption spectrum.

Parameters

- **absorb_time** (*numpy.ndarray*) – Time domain absorption.
- **config** (*dict*) – Configuration dictionary containing parameters: - total_time (float): Total time duration of the absorption signal. - dt (float): Time step size. - pad (int): Number of zero padding points for higher frequency resolution. - smoothdamp (bool): Whether to smooth the transition to the padded region with an exponential damping. - smoothdamp_start_percent (int): Percentage of the time domain absorption affected by smoothing.

Returns

(numpy.ndarray, numpy.ndarray)

- Absorption spectrum in the frequency domain.
- Corresponding frequency axis.

Return type

tuple

torchnise.absorption.**absorption_time_domain**(*time_evolution_operator*, *dipole_moments*, *use_damping=False*, *lifetime=1000*, *dt=1*)

Calculate the time domain absorption based on the time evolution operator.

Parameters

- **time_evolution_operator** (*numpy.ndarray*) – Time evolution operator with dimensions (realizations, timesteps, n_sites, n_sites).
- **dipole_moments** (*numpy.ndarray*) – Dipole moments with either shape (realizations, timesteps, n_sites, 3) for time-dependent cases, or shape (n_sites, 3) for time-independent cases.

- **use_damping** (*bool, optional*) – Whether to apply exponential damping to account for the lifetime of the state. Default is False.
- **lifetime** (*float, optional*) – Lifetime for the damping. Default is 1000. Units are not important as long as dt and lifetime have the same unit.
- **dt** (*float, optional*) – Time step size. Default is 1.

Returns
Time domain absorption.

Return type
numpy.ndarray

Notes

This function calculates the time domain absorption by summing over the contributions of different realizations, timesteps, and sites. An optional exponential damping factor can be applied to simulate the decay of the signal over time.

1.1.3 torchnise.averaging_and_lifetimes module

This module provides functions for averaging populations and estimating lifetimes of quantum states using various methods.

torchnise.averaging_and_lifetimes.**averaging**(*population*, *averaging_type*, *lifetimes=None*, *step=None*, *coherence=None*, *weight=None*)

Average populations using various methods.

Parameters

- **population** (*torch.Tensor*) – Population tensor.
- **averaging_type** (*str*) – Type of averaging to perform. Options are “standard”, “boltzmann”, “interpolated”.
- **lifetimes** (*torch.Tensor, optional*) – Lifetimes of the states, required for “interpolated” averaging.
- **step** (*float, optional*) – Time step size, required for “interpolated” averaging.
- **coherence** (*torch.Tensor, optional*) – Coherence matrix, needed for “boltzmann” and “interpolated”.
- **weight** (*torch.Tensor, optional*) – Weights for averaging.

Returns

(torch.Tensor, torch.Tensor) - Averaged population and coherence.

Return type
tuple

torchnise.averaging_and_lifetimes.**blend_and_normalize_populations**(*pop1*, *pop2*, *lifetimes*, *delta_t*)

Blend and normalize populations based on lifetimes.

Parameters

- **pop1** (*torch.Tensor*) – Initial population tensor.

- **pop2** (*torch.Tensor*) – Final population tensor to blend towards.
- **lifetimes** (*torch.Tensor*) – Lifetimes of the states.
- **delta_t** (*float*) – Time step size.

Returns
Normalized blended population tensor.

Return type
torch.Tensor

torchnise.averaging_and_lifetimes.**estimate_lifetime**(*u_tensor*, *delta_t*, *method='oscillatory_fit_mae'*)

Estimate lifetimes of quantum states using various fitting methods.

Parameters

- **u_tensor** (*torch.Tensor*) – Time evolution operator with dimensions (realizations, timesteps, n_sites, n_sites).
- **delta_t** (*float*) – Time step size.
- **method** (*str, optional*) – Method to use for lifetime estimation. Options are "oscillatory_fit_mae", "oscillatory_fit_mse", "simple_fit", "reverse_cummax", "simple_fit_mae". Default is "oscillatory_fit_mae".

Returns
Estimated lifetimes of each state.

Return type
torch.Tensor

torchnise.averaging_and_lifetimes.**estimate_lifetime_population**(*full_population*, *delta_t*, *method='oscillatory_fit_mae'*, *equilib=None*)

Estimate lifetimes of quantum states using various fitting methods.

Parameters

- **u_tensor** (*torch.Tensor*) – Time evolution operator with dimensions (realizations, timesteps, n_sites, n_sites).
- **delta_t** (*float*) – Time step size.
- **method** (*str, optional*) – Method to use for lifetime estimation. Options are "oscillatory_fit_mae", "oscillatory_fit_mse", "simple_fit", "reverse_cummax", "simple_fit_mae". Default is "oscillatory_fit_mae".

Returns
Estimated lifetimes of each state.

Return type
torch.Tensor

torchnise.averaging_and_lifetimes.**objective**(*tau*, *equilib_i*, *time_array*, *population*)

Objective function using Mean Squared Error (MSE) for fitting decays.

Parameters

- **tau** (*float*) – Decay constant.
- **equilib_i** (*float*) – Expected equilibrium population of state i.
- **time_array** (*torch.Tensor*) – Array of time steps.

- **population** (*torch.Tensor*) – Population tensor for the state.

Returns
Mean squared error (MSE) between the population and the fit.

Return type
float

torchnise.averaging_and_lifetimes.**objective_mae**(*tau*, *equilib_i*, *time_array*, *population*)

Objective function using Mean Absolute Error (MAE) for fitting decays.

Parameters

- **tau** (*float*) – Decay constant.
- **equilib_i** (*float*) – Expected equilibrium population of state i.
- **time_array** (*torch.Tensor*) – Array of time steps.
- **population** (*torch.Tensor*) – Population tensor for the state.

Returns
Mean absolute error (MAE) between the population and the fit.

Return type
float

torchnise.averaging_and_lifetimes.**objective_oscil_mae**(*tau_oscscale_oscstrength*, *equilib_i*, *time_array*, *population*)

Objective function using Mean Absolute Error (MAE) for fitting oscillatory decays.

Parameters

- **tau_oscscale_oscstrength** (*tuple*) – Tuple containing tau, osc_scale, and osc_strength parameters.
- **equilib_i** (*float*) – Expected equilibrium population of state i.
- **time_array** (*torch.Tensor*) – Array of time steps.
- **population** (*torch.Tensor*) – Population tensor for the state.

Returns
Mean absolute error (MAE) between the population and the fit.

Return type
float

torchnise.averaging_and_lifetimes.**objective_oscil_mse**(*tau_oscscale_oscstrength*, *equilib_i*, *time_array*, *population*)

Objective function using Mean Squared Error (MSE) for fitting oscillatory decays.

Parameters

- **tau_oscscale_oscstrength** (*tuple*) – Tuple containing tau, osc_scale, and osc_strength parameters.
- **equilib_i** (*float*) – Expected equilibrium population of state i.
- **time_array** (*torch.Tensor*) – Array of time steps.
- **population** (*torch.Tensor*) – Population tensor for the state.

Returns
Mean squared error (MSE) between the population and the fit.

Return type
float

torchnise.averaging_and_lifetimes.**objective_reverse_cummax**(*tau*, *equilib_i*, *time_array*, *population*)

Objective function using reverse cumulative maximum for fitting decays.

Parameters

- **tau** (*float*) – Decay constant.
- **equilib_i** (*float*) – Expected equilibrium population of state i.
- **time_array** (*torch.Tensor*) – Array of time steps.
- **population** (*torch.Tensor*) – Population tensor for the state.

Returns

Weighted mean squared error (MSE) between the population and
the fit.

Return type
float

torchnise.averaging_and_lifetimes.**reshape_weights**(*weight*, *population*, *coherence*)

Reshape weights for averaging.

Parameters

- **weight** (*torch.Tensor*) – Weight tensor.
- **population** (*torch.Tensor*) – Population tensor.
- **coherence** (*torch.Tensor, optional*) – Coherence matrix.

Returns

(torch.Tensor, torch.Tensor) - Reshaped weights for population
and coherence.

Return type
tuple

1.1.4 torchnise.fft_noise_gen module

This file implements the fftNoiseGEN algorithm for time correlated Noise.

torchnise.fft_noise_gen.**gen_noise**(*spectral_funcs*, *dt*, *shape*, *use_h5*, *dtype=torch.float32*)

Generates time-correlated noise following the power spectrums provided in spectral_funcs.

Parameters

- **spectral_funcs** (*list(callable)*) – Must have either len 1 if all sites follow the same power spectrum, or len n_sites=shape[-1] to provide a separate power spectrum for each site.
- **dt** (*float*) – Time step size.
- **shape** (*tuple*) – Shape of the output noise array. The first dimension is the number of realizations, the second dimension is the number of steps, and the remaining dimension is the number of sites.
- **use_h5** (*bool*) – If True, uses h5py to save tensor to disk.
- **dtype** (*torch.dtype*) – Data type of the output noise array.

Returns
Time-correlated noise with the specified shape.

Return type
torch.Tensor

torchnise.fft_noise_gen.**inverse_sample**(*dist*, *shape*, *x_min=-100*, *x_max=100*, *n=100000.0*, ***kwargs*)

Generates samples from a given distribution using the inverse transform sampling method.

Parameters

- **dist** (*callable*) – Probability density function (PDF) of the desired distribution.
- **shape** (*tuple*) – Shape of the output samples.
- **x_min** (*float*) – Minimum x value for the range of the distribution.
- **x_max** (*float*) – Maximum x value for the range of the distribution.
- **n** (*int*) – Number of points used to approximate the cumulative distribution function (CDF).
- ****kwargs** – Additional arguments to pass to the PDF function.

Returns
Samples drawn from the specified distribution.

Return type
np.ndarray

torchnise.fft_noise_gen.**noise_algorithm**(*shape*, *dt*, *spectral_func*, *axis=-1*, *sample_dist=None*, *discard_half=True*, *save=False*, *save_name=None*)

Generates time-correlated noise following the power spectrum provided in spectral_func.

Parameters

- **shape** (*tuple*) – Shape of the output noise array.
- **dt** (*float*) – Time step size.
- **spectral_func** (*callable*) – Function that defines the power spectrum of the noise.
- **axis** (*int, optional*) – The axis along which the noise should be correlated. Default is -1 (last axis).
- **sample_dist** (*callable, optional*) – Function to generate an array of random numbers for non-normal distribution.
- **discard_half** (*bool, optional*) – If True, generates noise for twice the number of steps and discards the second half. Default is True.
- **save** (*bool, optional*) – If True, saves the generated noise array to a file.
- **save_name** (*str, optional*) – Name of the file to save the noise array. Required if save is True.

Returns
Time-correlated noise with the specified shape.

Return type
np.ndarray

torchnise.fft_noise_gen.**noise_algorithm_torch**(*shape*, *dt*, *spectral_func*, *axis=-1*, *sample_dist=None*, *discard_half=True*, *save=False*, *save_name=None*)

Pytorch vesrsion: Generates time-correlated noise following the power spectrum provided in spectral_func.

Parameters

- **shape** (*tuple*) – Shape of the output noise array.
- **dt** (*float*) – Time step size.
- **spectral_func** (*callable*) – Function that defines the power spectrum of the noise.
- **axis** (*int, optional*) – The axis along which the noise should be correlated. Default is -1 (last axis).
- **sample_dist** (*callable, optional*) – Function to generate an array of random numbers for non-normal distribution.
- **discard_half** (*bool, optional*) – If True, generates noise for twice the number of steps and discards the second half. Default is True.
- **save** (*bool, optional*) – If True, saves the generated noise array to a file.
- **save_name** (*str, optional*) – Name of the file to save the noise array. Required if save is True.

Returns
Time-correlated noise with the specified shape.

Return type
torch.tensor

1.1.5 torchnise.mlnise_dataset module

mlnise_dataset.py

This module provides a dataset class (MLNiseDrudeDataset) for training ML-based NISE models using HEOM calculations from PyHEOM. It includes a runHeomDrude function that demonstrates how to generate time-dependent populations for random Hamiltonians with Drude spectral density noise.

class torchnise.mlnise_dataset.**MLNiseDrudeDataset**(*length: int = 1000*, *total_time: float = 1000.0*, *dt_fs: float = 1.0*, *n_sites: int = 2*, *seed: int = 42*, *min_energy: float = -500.0*, *max_energy: float = 500.0*, *min_coupling: float = -200.0*, *max_coupling: float = 200.0*, *min_tau: float = 10.0*, *max_tau: float = 200.0*, *min_temp: float = 300.0*, *max_temp: float = 300.0*, *min_reorg: float = 0.0*, *max_reorg: float = 500.0*, *depth: int = 9*, *dataset_folder: str = 'GeneratedHeom'*, *dataset_name: str = 'mlnise_example_dataset'*, *nn_coupling: str = 'ring'*, *generate_if_missing: bool = True*)

Bases: Dataset

A Dataset class that uses PyHEOM to generate or load population dynamics for training ML-based NISE. Each sample's Hamiltonian and population are stored on disk or generated on the fly.

Each item yields:
inputs: (H, T, E_reorg, tau, total_time, dt, psi0, n_sites) target: population array of shape (time_steps, n_sites)

The shape of H is (time_steps, n_sites, n_sites), and population is (time_steps, n_sites).

torchnise.mlnise_dataset.**runHeomDrude**(*n_state: int*, *H: ndarray*, *tau: float*, *Temp: float*, *E_reorg: float*, *dt__unit: float*, *initiallyExcitedState: int*, *totalTime: float*, *tier: int*, *matrix_type: str = 'sparse'*) → ndarray

Generate time-dependent populations via PyHEOM for a system with Drude spectral density. Each site is coupled to a bath with reorganization energy E_reorg and correlation time derived from tau.

Parameters

- **n_state** (*int*) – Number of states (sites).
- **H** (*np.ndarray*) – Hamiltonian of shape (..., n_state, n_state). Typically just (n_state, n_state) for static usage, or (time_steps, n_state, n_state) if time-dependent.
- **tau** (*float*) – Bath correlation time (fs).
- **Temp** (*float*) – Temperature (K).
- **E_reorg** (*float*) – Reorganization energy (cm^-1).
- **dt__unit** (*float*) – Timestep in fs.
- **initiallyExcitedState** (*int*) – Index of the initially excited site.
- **totalTime** (*float*) – Total simulation time in fs.
- **tier** (*int*) – HEOM hierarchy depth.
- **matrix_type** (*str*) – PyHEOM matrix type ("sparse", "dense"), etc.

Returns
2D array of populations, shape (time_steps, n_state).

Return type
np.ndarray

1.1.6 torchnise.nise module

This file contains the main module Implementing the NISE calculations

class torchnise.nise.**MLNISEModel**

Bases: Module

Neural network model to predict correction factors for non-adiabatic coupling based on input features.

fc1

First fully connected layer.

Type
nn.Linear

fc2

Second fully connected layer.

Type
nn.Linear

fc3

Third fully connected layer.

Type
nn.Linear

fc4

Output fully connected layer.

Type
nn.Linear

forward(*mlnise_inputs*, *de*, *kbt*, *phi_b*, *s*, *jj*, *ii*, *realizations*, *device='cpu'*)

Forward pass through the MLNISE model to calculate correction factors.

Parameters

- **mlnise_inputs** (*tuple*) – Inputs for the MLNISE model, containing reorganization energy and correlation time.
- **de** (*torch.Tensor*) – Energy differences between states.
- **kbt** (*float*) – Thermal energy (k_B * T).
- **phi_b** (*torch.Tensor*) – Wavefunction in the eigenbasis.
- **s** (*torch.Tensor*) – Non-adiabatic coupling matrix.
- **jj** (*int*) – Index of the target state.
- **ii** (*int*) – Index of the current state.
- **realizations** (*int*) – Number of noise realizations.
- **device** (*str, optional*) – Device for computation ("cpu" or "cuda"). Defaults to "cpu".

Returns

Correction factor for the non-adiabatic coupling
matrix.

Return type
torch.Tensor

torchnise.nise.**apply_t_correction**(*s*, *n_sites*, *realizations*, *device*, *e*, *eold*, *t_correction*, *kbt*, *mlnise_model*, *mlnise_inputs*, *phi_b*, *aranged_realizations*)

torchnise.nise.**nise_averaging**(*hfull*, *realizations*, *psi0*, *total_time*, *dt*, *temperature*, *save_interval=1*, *t_correction='None'*, *averaging_method='standard'*, *lifetime_factor=5*, *device='cpu'*, *save_coherence=True*, *save_u=False*, *save_multi_pop=False*, *save_multi_slice=None*, *mlnise_inputs=None*, *mlnise_model=None*, *use_h5=False*, *constant_v=False*, *site_noise=None*, *v_time_dependent=None*, *v_dt=None*)

Run NISE propagation with different averaging methods to calculate averaged population dynamics.

Parameters

- **hfull** (*torch.Tensor*) – Hamiltonian of the system over time for different realizations.
- **realizations** (*int*) – Number of noise realizations to simulate.
- **psi0** (*torch.Tensor*) – Initial state of the system.
- **total_time** (*float*) – Total time for the simulation.
- **dt** (*float*) – Time step size.
- **temperature** (*float*) – Temperature for thermal corrections.
- **save_interval** (*int, optional*) – Interval for saving results. Defaults to 1.

- **t_correction** (*str, optional*) – Method for thermal correction ("None", "TNISE", "MLNISE"). Defaults to "None".
- **averaging_method** (*str, optional*) – Method for averaging results ("standard", "boltzmann", "interpolated"). Defaults to "standard".
- **lifetime_factor** (*int, optional*) – Factor to scale estimated lifetimes. Defaults to 5.
- **device** (*str, optional*) – Device for computation ("cpu" or "cuda"). Defaults to "cpu".
- **save_coherence** (*bool, optional*) – If True, save coherences. Defaults to False.
- **save_u** (*bool, optional*) – If True, save time evolution operators. Defaults to False.
- **save_multi_pop** (*bool, optional*) – if True, save the population of the sites defined by save_multi_slice, Defaults to False
- **save_multi_slice** (*slice, optional*) – defines the slice of sites to save
- **mlnise_inputs** (*tuple, optional*) – Inputs for MLNISE model. Defaults to None.
- **mlnise_model** (*nn.Module, optional*) – Machine learning model for MLNISE corrections. Defaults to None.#
- **use_h5** (*bool, optional*) – saves all tensors that are not currently used to the disk in HDF5 format. reduces memory foorprint at some performance cost usually worth it for big systems. Defaults to False
- **constant_v** (*bool, optional*) – uses the constant coupling mode, hfull is considered to be time independent and the noise is expected to be provided via site_noise, the full hamilatonian is then hfull+diag_embedd(site_noise). Defaults to false
- **site_noise** (*torch.tensor*) – site noise used for constant coupling mode or v_time_dependent mode. Should have shape (steps,realizations,n_sites) Defaults to None.
- **v_time_dependent** (*torch.tensor*) – if not None these couplings will be used with the specified timestep v_dt. Should only be used if v_dt is larger than dt, otherwise use the Full Hamiltonian in hfull. Defaults to None.
- **v_dt** (*float*) – timestep for the time dependent coupling.

Returns

(torch.Tensor, torch.Tensor, torch.Tensor, torch.Tensor) -
Averaged populations, coherences, time evolution operators, and lifetimes.

Return type
tuple

torchnise.nise.**nise_propagate**(*hfull*, *realizations*, *psi0*, *total_time*, *dt*, *temperature*, *save_interval=1*, *t_correction='None'*, *device='cpu'*, *save_u=False*, *save_coherence=False*, *mlnise_inputs=None*, *mlnise_model=None*, *use_h5=False*, *constant_v=False*, *site_noise=None*, *v_time_dependent=None*, *v_dt=None*)

Propagate the quantum state using the NISE algorithm with optional thermal
corrections.

Parameters

- **hfull** (*torch.Tensor*) – Hamiltonian of the system over time for different realizations. Shape should be (time_steps,realizations,n_sites,n_sites) except for constant_V or v_time_dependent mode where it is (n_sites,n_sites)
- **realizations** (*int*) – Number of noise realizations to simulate.

- **psi0** (*torch.Tensor*) – Initial state of the system.
- **total_time** (*float*) – Total time for the simulation.
- **dt** (*float*) – Time step size.
- **temperature** (*float*) – Temperature for thermal corrections.
- **save_interval** (*int, optional*) – Interval for saving results. Defaults to 1.
- **t_correction** (*str, optional*) – Method for thermal correction ("None", "TNISE", "MLNISE"). Defaults to "None".
- **device** (*str, optional*) – Device for computation ("cpu" or "cuda"). Defaults to "cpu".
- **save_u** (*bool, optional*) – If True, save time evolution operators. Defaults to False.
- **save_coherence** (*bool, optional*) – If True, save coherences. Defaults to False.
- **mlnise_inputs** (*tuple, optional*) – Inputs for MLNISE model. Defaults to None.
- **mlnise_model** (*nn.Module, optional*) – Machine learning model for MLNISE corrections. Defaults to None.#
- **use_h5** (*bool, optional*) – saves all tensors that are not currently used to the disk in HDF5 format. reduces memory foorprint at some performance cost usually worth it for big systems. Defaults to False
- **constant_v** (*bool, optional*) – uses the constant coupling mode, hfull is considered to be time independent and the noise is expected to be provided via site_noise, the full hamilatonian is then hfull+diag_embedd(site_noise). Defaults to false
- **site_noise** (*torch.tensor*) – site noise used for constant coupling mode or v_time_dependent mode. Should have shape (steps,realizations,n_sites) Defaults to None.
- **v_time_dependent** (*torch.tensor*) – if not None these couplings will be used with the specified timestep v_dt. Should only be used if v_dt is larger than dt, otherwise use the Full Hamiltonian in hfull. Defaults to None.
- **v_dt** (*float*) – timestep for the time dependent coupling.

Returns
(torch.Tensor, torch.Tensor, torch.Tensor) - Populations, coherences, and time evolution operators.

Return type
tuple

torchnise.nise.**run_nise**(*h*, *realizations*, *total_time*, *dt*, *initial_state*, *temperature*, *spectral_funcs*, *save_interval=1*, *save_u=False*, *save_u_file=None*, *save_multi_pop=False*, *save_multi_slice=None*, *save_pop_file=None*, *t_correction='None'*, *mode='Population'*, *mu=None*, *absorption_padding=10000*, *averaging_method='standard'*, *lifetime_factor=5*, *max_reps=100000*, *mlnise_inputs=None*, *mlnise_model=None*, *device='cpu'*, *use_h5=False*, *v_time_dependent=None*, *v_dt=None*)

Main function to run NISE simulations for population dynamics or absorption spectra.

Parameters

- **h** (*torch.Tensor*) – Hamiltonian of the system over time or single Hamiltonian with noise.
- **realizations** (*int*) – Number of noise realizations to simulate.
- **total_time** (*float*) – Total time for the simulation.

- **dt** (*float*) – Time step size.
- **initial_state** (*torch.Tensor*) – Initial state of the system.
- **temperature** (*float*) – Temperature for thermal corrections.
- **spectral_funcs** (*list(callable)*) – Spectral density functions for noise generation.
- **save_interval** (*int, optional*) – Interval for saving results. Defaults to 1.
- **save_u** (*bool, optional*) – if the time_evolution operator should be saved Defaults to False
- **save_u_file** (*str, optional*) – if not none, u will be saved to this file
- **save_multi_pop** (*bool, optional*) – if True, save the population of the sites defined by save_multi_slice, Defaults to False
- **save_multi_slice** (*slice, optional*) – defines the slice of sites to save
- **save_pop_file** (*str, optional*) – if not none, the population of all sites will be saved to this file
- **t_correction** (*str, optional*) – Method for thermal correction ("None", "TNISE", "MLNISE"). Defaults to "None".
- **mode** (*str, optional*) – Simulation mode ("Population" or "Absorption"). Defaults to "Population".
- **mu** (*torch.Tensor, optional*) – Dipole moments for absorption calculations. Defaults to None.
- **absorption_padding** (*int, optional*) – Padding for absorption spectra calculation. Defaults to 10000.
- **averaging_method** (*str, optional*) – Method for averaging results ("standard", "boltzmann", "interpolated"). Defaults to "standard".
- **lifetime_factor** (*int, optional*) – Factor to scale estimated lifetimes. Defaults to 5.
- **max_reps** (*int, optional*) – Maximum number of realizations per chunk. Defaults to 100000.
- **mlnise_inputs** (*tuple, optional*) – Inputs for MLNISE model. Defaults to None.
- **mlnise_model** (*nn.Module, optional*) – Machine learning model for MLNISE corrections. Defaults to None.#
- **device** (*str, optional*) – Device for computation ("cpu" or "cuda"). Defaults to "cpu".
- **use_h5** (*bool, optional*) – saves all tensors that are not currently used to the disk in HDF5 format. reduces memory foorprint at some performance cost usually worth it for big systems. Defaults to False
- **v_time_dependent** (*torch.tensor*) – if not None these couplings will be used with the specified timestep v_dt. Should only be used if v_dt is larger than dt, otherwise use the Full Hamiltonian in hfull. Defaults to None.
- **v_dt** (*float*) – timestep for the time dependent coupling.

Returns

Depending on mode, returns either (np.ndarray, np.ndarray) for
absorption spectrum and frequency axis, or (torch.Tensor, torch.Tensor) for averaged populations and time axis.

Return type
tuple

1.1.7 torchnise.pytorch_utility module

This file contains some utility functions

class torchnise.pytorch_utility.**H5Tensor**(*data=None*, *h5_filepath=None*, *requires_grad=False*, *dtype=torch.float32*, *shape=None*)

Bases: object

A class for handling PyTorch tensors stored in HDF5 files.

This class provides a way to work with large tensors that do not fit into memory.

to(*device*)

to_tensor()

Convert the H5Tensor to a PyTorch tensor. If the data is stored in HDF5, it will load it into memory.

torchnise.pytorch_utility.**batch_trace**(*A: Tensor*, *dim1: int = -1*, *dim2: int = -2*) → Tensor

Compute the batch trace of a tensor along specified dimensions.

Parameters

- **A** (*torch.Tensor*) – Input tensor.
- **dim1** (*int*) – First dimension to compute trace along.
- **dim2** (*int*) – Second dimension to compute trace along.

Returns
Trace of the input tensor along the specified dimensions.

Return type
torch.Tensor

torchnise.pytorch_utility.**clean_temp_files**()

Remove all temporary .bin files.

torchnise.pytorch_utility.**create_empty_mmap_tensor**(*shape*, *dtype=torch.float32*) → Tensor

A custom function to create memory-mapped tensors.

This function handles the creation of memory-mapped tensors, ensuring that the data is efficiently managed and temporary files are cleaned up properly. The mmaped tensors are always on cpu.

Parameters
tensor (*torch.Tensor*) – Input tensor.

Returns
Memory-mapped tensor.

Return type
torch.Tensor

torchnise.pytorch_utility.**delete_file**(*filename: str*) → None

Delete the temporary file.

Parameters
filename (*str*) – Path to the file to be deleted.

torchnise.pytorch_utility.**golden_section_search**(*func*, *a*, *b*, *tol*)

Perform a golden section search to find the minimum of a unimodal function on a closed interval [a, b].

Parameters

- **func** (*callable*) – The unimodal function to minimize.
- **a** (*float*) – The lower bound of the search interval.
- **b** (*float*) – The upper bound of the search interval.
- **tol** (*float*) – The tolerance for stopping the search. The search stops when the interval length is less than this value.

Returns
The point at which the function has its minimum within the interval [a, b].

Return type
float

torchnise.pytorch_utility.**is_memory_mapped**(*tensor*) → bool

Check if a given PyTorch tensor is memory-mapped from a file.

A memory-mapped tensor is created by mapping a file into the tensor's storage. This function inspects the storage of the given tensor to determine if it was created from a memory-mapped file.

Parameters
tensor (*torch.Tensor*) – The PyTorch tensor to check.

Returns
True if the tensor is memory-mapped, False otherwise.

Return type
bool

Raises
Warning – If the tensor's storage does not have a filename attribute, (usually because pytorch version is less than 2.2) it can not be determined if the tensor is memory mapped. It is assumed that it is not.

torchnise.pytorch_utility.**matrix_logh**(*A: Tensor*, *dim1: int = -1*, *dim2: int = -2*, *epsilon: float = 1e-05*) → Tensor

Compute the Hermitian matrix logarithm of a square matrix or a batch of square matrices.It is the unique hermitian matrix logarithm see math.stackexchange.com/questions/4474139/logarithm-of-a-positive-definite-matrix

Parameters

- **A** (*torch.Tensor*) – Input tensor with square matrices in the last two dimensions.
- **dim1** (*int*) – First dimension of the square matrices. Default is -1.
- **dim2** (*int*) – Second dimension of the square matrices. Default is -2.
- **epsilon** (*float*) – Small value to add to the diagonal to avoid numerical issues.

Returns
Matrix logarithm of the input tensor.

Return type
torch.Tensor

torchnise.pytorch_utility.**renorm**(*phi: Tensor*, *eps: float = 1e-08*, *dim: int = -1*) → Tensor

Renormalize a batch of wavefunctions.

Parameters

- **phi** (*torch.Tensor*) – Batch of wavefunctions to be renormalized.
- **eps** (*float*) – Small threshold to avoid division by zero.
- **dim** (*int*) – Dimension along which the wavefunctions are stored. Default is -1.

Returns
Renormalized wavefunctions.

Return type
torch.Tensor

torchnise.pytorch_utility.**smooth_damp_to_zero**(*f_init*, *start*, *end*)

Smoothly damp a segment of an array to zero using an exponential damping function.

Parameters

- **f_init** (*numpy.ndarray*) – Initial array to be damped.
- **start** (*int*) – Starting index of the segment to damp.
- **end** (*int*) – Ending index of the segment to damp.

Returns
Array after applying the damping.

Return type
numpy.ndarray

torchnise.pytorch_utility.**tensor_to_mmap**(*tensor*) → Tensor

A custom function to create memory-mapped tensors.

This function handles the creation of memory-mapped tensors, ensuring that the data is efficiently managed and temporary files are cleaned up properly. The mmaped tensors are always on cpu.

Parameters
tensor (*torch.Tensor*) – Input tensor.

Returns
Memory-mapped tensor.

Return type
torch.Tensor

torchnise.pytorch_utility.**weighted_mean**(*tensor: Tensor*, *weights: Tensor*, *dim=0*) → Tensor

Compute the weighted mean of 'tensor' along dimension 'dim' using 'weights' as the weights for each slice along 'dim'.

Parameters

- **tensor** (*torch.Tensor*) – Input data of shape (..., N, ...)
- **weights** (*torch.Tensor*) – 1D weights of shape (N,).
- **dim** (*int*) – The dimension along which to apply the weights.

Returns
Weighted average along dimension 'dim'.

Return type
torch.Tensor

1.1.8 torchnise.spectral_density_generation module

This file contains techniques for getting spectral densities from noise

torchnise.spectral_density_generation.**adjust_tensor_length**(*a*, *l*)

Adjust the length of a tensor by trimming or padding with zeros.

Parameters

- **a** (*torch.Tensor*) – Input tensor.
- **l** (*int*) – Desired length of the output tensor.

Returns
Adjusted tensor of length l.

Return type
torch.Tensor

torchnise.spectral_density_generation.**autocorrelation**(*noise*, *i*, *n*)

Calculate the autocorrelation function for a given noise matrix.

Parameters

- **noise** (*np.ndarray*) – The input noise matrix.
- **i** (*int*) – The current index for autocorrelation calculation.
- **n** (*int*) – The total number of timesteps.

Returns
The autocorrelation value for the given index i.

Return type
np.ndarray

torchnise.spectral_density_generation.**ccalc**(*noise*, *n*, *reals*)

Calculate the autocorrelation matrix for the entire noise dataset.

Parameters

- **noise** (*np.ndarray*) – The input noise matrix.
- **n** (*int*) – The total number of timesteps.
- **reals** (*int*) – The number of realizations.

Returns
The autocorrelation matrix with size (reals, N).

Return type
np.ndarray

torchnise.spectral_density_generation.**ensure_tensor_on_device**(*array*, *device='cuda'*, *dtype=torch.float32*)

Ensure the input is a PyTorch tensor on the specified device. And move it if its not.

Parameters

- **array** (*np.ndarray or torch.Tensor*) – Input array or tensor.
- **device** (*str, optional*) – Desired device ('cuda' or 'cpu'). Defaults to 'cuda'.
- **dtype** (*torch.dtype, optional*) – Desired data type. Defaults to torch.float.

Returns
Tensor on the specified device with the desired data type.

Return type
torch.Tensor

torchnise.spectral_density_generation.**expval_auto**(*noise*, *n*, *reals*)

Calculate the expectation value of the autocorrelation function.

Parameters

- **noise** (*np.ndarray*) – The input noise matrix.
- **n** (*int*) – The total number of timesteps.
- **reals** (*int*) – The number of realizations.

Returns
The expectation value of the autocorrelation function.

Return type
np.ndarray

torchnise.spectral_density_generation.**get_auto**(*noise*)

Get the autocorrelation function for a given noise dataset.

Parameters
noise (*np.ndarray*) – The input noise matrix.

Returns
The autocorrelation function.

Return type
np.ndarray

torchnise.spectral_density_generation.**nnls_pyytorch_scipy**(*A*, *b*)

Solve the non-negative least squares problem for PyTorch Tensors using scipy.

Parameters

- **A** (*torch.Tensor*) – The input matrix A.
- **b** (*torch.Tensor*) – The input vector b.

Returns
Solution vector x_nnls as a PyTorch tensor.

Return type
torch.Tensor

torchnise.spectral_density_generation.**objective_function**(*lambda_ij*, *A*, *b*, *sparcity_penalty*, *l1_norm_penalty*, *solution_penalty*, *negative_penalty*, *ljnorm_penalty*, *j*)

Objective function with TV norm, L1 norm, and penalties for constraints.
Used for Superresolution

Parameters

- **lambda_ij** (*torch.Tensor*) – Current solution tensor.
- **A** (*torch.Tensor*) – Matrix for the linear system.
- **b** (*torch.Tensor*) – Target vector.

- **sparcity_penalty** (*float*) – Penalty term for sparsity in the solution.
- **l1_norm_penalty** (*float*) – L1 norm penalty for regularization.
- **solution_penalty** (*float*) – Penalty for the solution norm.
- **negative_penalty** (*float*) – Penalty for negative peaks.
- **ljnorm_penalty** (*float*) – L_j norm penalty for regularization.
- **j** (*float*) – Exponent for the L_j norm.

Returns
Value of the objective function.

Return type
torch.Tensor

torchnise.spectral_density_generation.**objective_function_no_penalty**(*lambda_ij*, *sparcity_penalty*, *l1_norm_penalty*, *negative_penalty*, *ljnorm_penalty*, *j*)

Objective function without the solution penalty.

Parameters

- **lambda_ij** (*torch.Tensor*) – Current solution tensor.
- **sparcity_penalty** (*float*) – Penalty term for sparsity in the solution.
- **l1_norm_penalty** (*float*) – L1 norm penalty for regularization.
- **negative_penalty** (*float*) – Penalty for negative peaks.
- **ljnorm_penalty** (*float*) – L_j norm penalty for regularization.
- **j** (*float*) – Exponent for the L_j norm.

Returns
Value of the objective function without the solution penalty.

Return type
torch.Tensor

torchnise.spectral_density_generation.**optimize_lambda**(*A*, *b*, *sparcity_penalty*, *l1_norm_penalty*, *solution_penalty*, *negative_penalty*, *ljnorm_penalty*, *j*, *eta*, *max_iter=1000*, *tol=1e-06*, *lr=0.01*, *device='cuda'*, *initial_guess=None*, *verbose=False*)

Optimization loop using PyTorch.

Parameters

- **A** (*torch.Tensor*) – Matrix for the linear system.
- **b** (*torch.Tensor*) – Target vector.
- **sparcity_penalty** (*float*) – Penalty term for sparsity in the solution.
- **l1_norm_penalty** (*float*) – L1 norm penalty for regularization.
- **solution_penalty** (*float*) – Penalty for the solution norm.
- **negative_penalty** (*float*) – Penalty for negative peaks.

- **ljnorm_penalty** (*float*) – L_j norm penalty for regularization.
- **j** (*float*) – Exponent for the L_j norm.
- **eta** (*float*) – Regularization term for optimization.
- **max_iter** (*int, optional*) – Maximum number of iterations. Defaults to 1000.
- **tol** (*float, optional*) – Tolerance for convergence. Defaults to 1e-6.
- **lr** (*float, optional*) – Learning rate for the optimization algorithm. Defaults to 0.01.
- **device** (*str, optional*) – Device for computation ('cuda' or 'cpu'). Defaults to 'cuda'.
- **initial_guess** (*torch.Tensor, optional*) – Initial guess for the optimization. Defaults to None.
- **verbose** (*bool, optional*) – decide if information should be printed

Returns
Optimized solution tensor.

Return type
torch.Tensor

torchnise.spectral_density_generation.**optimize_lambda_nnls**(*A*, *b*, *initial_guess=None*, *max_iter=1000*, *lr=0.01*, *verbose=False*)

Perform non-negative least squares optimization using PyTorch.

Parameters

- **A** (*torch.Tensor*) – Matrix for the linear system.
- **b** (*torch.Tensor*) – Target vector.
- **initial_guess** (*torch.Tensor, optional*) – Initial guess for the optimization. Defaults to None.
- **max_iter** (*int, optional*) – Maximum number of iterations. Defaults to 1000.
- **lr** (*float, optional*) – Learning rate for the optimization algorithm. Defaults to 0.01.
- **erbose** (*bool, optional*) – decide if information should be printed

Returns
Optimized solution tensor.

Return type
torch.Tensor

torchnise.spectral_density_generation.**sd_reconstruct_fft**(*auto*, *dt*, *temperature*, *min_w=None*, *max_w=None*, *damping_type=None*, *cutoff=None*, *rescale=False*)

Reconstruct the spectral density using FFT from the autocorrelation function.

Parameters

- **auto** (*np.ndarray*) – Autocorrelation function.
- **dt** (*float*) – Time step between autocorrelation points.
- **temperature** (*float*) – Temperature.
- **min_w** (*float, optional*) – Minimum frequency to consider. Defaults to None.
- **max_w** (*float, optional*) – Maximum frequency to consider. Defaults to None.

- **damping_type** (*str, optional*) – Type of damping to apply ('step', 'gauss', 'exp'). Defaults to None.
- **cutoff** (*float, optional*) – Cutoff for damping. Defaults to None.
- **rescale** (*bool, optional*) – If True, rescale the autocorrelation function. Defaults to False.

Returns

(np.ndarray, np.ndarray, np.ndarray) - Reconstructed spectral
density, frequency axis, and damped autocorrelation.

Return type
tuple

torchnise.spectral_density_generation.**sd_reconstruct_superresolution**(*auto*, *dt*, *temperature*, *sparcity_penalty=1*, *l1_norm_penalty=1*, *solution_penalty=10000*, *negative_penalty=1*, *ljnorm_penalty=0*, *j=0.5*, *lr=0.01*, *max_iter=1000*, *eta=1e-07*, *tol=1e-07*, *device='cuda'*, *cutoff=None*, *frequencies=None*, *linewidths=None*, *sample_frequencies=None*, *top_n=False*, *top_tresh=False*, *second_optimization=False*, *chunk_memory=1000000000.0*, *auto_length_debias=None*, *auto_length_return=None*)

Reconstruct the super-resolution spectral density from the autocorrelation function.

Parameters

- **auto** (*torch.Tensor*) – Autocorrelation function.
- **dt** (*float*) – Time step between autocorrelation points.
- **temperature** (*float*) – Temperature.
- **sparcity_penalty** (*float*) – Penalty term for sparsity in the solution.
- **l1_norm_penalty** (*float*) – L1 norm penalty for regularization.
- **solution_penalty** (*float*) – Penalty for the solution norm.
- **negative_penalty** (*float*) – Penalty for negative peaks.
- **ljnorm_penalty** (*float*) – L_j norm penalty for regularization.
- **j** (*float*) – j determining the L_j norm.
- **lr** (*float*) – Learning rate for the optimization algorithm.
- **max_iter** (*int*) – Maximum number of iterations for the optimization.
- **eta** (*float*) – Regularization term for optimization.
- **tol** (*float*) – Tolerance for convergence in the optimization.

- **device** (*str*) – Device for computation ('cuda' or 'cpu').
- **cutoff** (*float, optional*) – Cutoff for damping. Defaults to None.
- **frequencies** (*torch.tensor, optional*) – Frequencies for peaks that should be included in the optimization, otherwise default values are used
- **linewidths** (*torch.tensor, optional*) – Linewidths for peaks that should be included in the optimization, otherwise default values are used
- **sample_frequencies** (*torch.Tensor, optional*) – Frequencies for sampling the spectral density. Defaults to None.
- **top_n** (*int*) – If not False, only the top n coefficients are used. Defaults to False.
- **top_tresh** (*float*) – Alternative to top_n chooses all coefficients above a treshold
- **second_optimization** (*bool*) – If True, a second optimization step is performed using top n coefficients. Defaults to False.
- **chunk_memory** (*float*) – The maximum amount of memory (in bytes) to use for each chunk. Defaults to 1GB.
- **auto_length_debias** (*float*) – if not False: length of autocorrelation in fs for second optimizazion fitting, will be zero padded or cut
- **auto_length_return** (*float*) – if not False: length of the returned autocorrelation inn fs

Returns

Reconstructed spectral
density, sampled frequencies, and super-resolved autocorrelation function.

Return type
Tuple[np.ndarray, np.ndarray, np.ndarray]

torchnise.spectral_density_generation.**tv_norm_2d**(*lambda_ij*)

Calculate the total variation norm across both dimensions.

Parameters
lambda_ij (*torch.Tensor*) – Input tensor.

Returns
Total variation norm.

Return type
torch.Tensor

1.1.9 torchnise.spectral_functions module

This file implements various spectral functions returning the power spectrum.

torchnise.spectral_functions.**spectral_drude**(*w*, *gamma*, *strength*, *temperature*)

Drude spectral density function.

Parameters

- **w** (*numpy.ndarray*) – Frequency array.
- **gamma** (*float*) – Drude relaxation rate.
- **strength** (*float*) – Strength of the spectral density.
- **temperature** (*float*) – Temperature.

Returns
Spectral density.

Return type
numpy.ndarray

torchnise.spectral_functions.**spectral_drude_lorentz**(*w*, *gamma*, *strength*, *wk*, *sk*, *temperature*, *gammak*)

Combined Drude and Lorentz spectral density function.

Parameters

- **w** (*numpy.ndarray*) – Frequency array.
- **gamma** (*float*) – Drude relaxation rate.
- **strength** (*float*) – Strength of the spectral density.
- **wk** (*list*) – Frequencies of the Lorentz peaks.
- **sk** (*list*) – Strengths of the Lorentz peaks.
- **temperature** (*float*) – Temperature.
- **gammak** (*float*) – Damping factor.

Returns
Spectral density.

Return type
numpy.ndarray

torchnise.spectral_functions.**spectral_drude_lorentz_heom**(*w*, *omega_k*, *lambda_k*, *temperature*, *vk*)

Drude-Lorentz spectral density function for HEOM.

Parameters

- **w** (*numpy.ndarray*) – Frequency array.
- **omega_k** (*list*) – Frequencies of the peaks.
- **lambda_k** (*list*) – Strengths of the peaks.
- **temperature** (*float*) – Temperature.
- **vk** (*float*) – Damping factor.

Returns
Spectral density.

Return type
numpy.ndarray

torchnise.spectral_functions.**spectral_log_normal**(*w*, *s_hr*, *sigma*, *wc*, *temperature*)

Log-normal spectral density function.

Parameters

- **w** (*numpy.ndarray*) – Frequency array.
- **s_hr** (*float*) – Huang-Rhys factor.
- **sigma** (*float*) – Width of the log-normal distribution.
- **wc** (*float*) – Central frequency of the log-normal distribution.
- **temperature** (*float*) – Temperature.

Returns
Spectral density.

Return type
numpy.ndarray

torchnise.spectral_functions.**spectral_log_normal_lorentz**(*w*, *wk*, *sk*, *temperature*, *gammak*, *s_hr*, *sigma*, *wc*)

Combined Log-Normal and Lorentz spectral density function.

Parameters

- **w** (*numpy.ndarray*) – Frequency array.
- **wk** (*list*) – Frequencies of the Lorentz peaks.
- **sk** (*list*) – Strengths of the Lorentz peaks.
- **temperature** (*float*) – Temperature.
- **gammak** (*float*) – Damping factor.
- **s_hr** (*float*) – Huang-Rhys factor.
- **sigma** (*float*) – Width of the log-normal distribution.
- **wc** (*float*) – Central frequency of the log-normal distribution.

Returns
Spectral density.

Return type
numpy.ndarray

torchnise.spectral_functions.**spectral_lorentz**(*w*, *wk*, *sk*, *temperature*, *gammak*)

Lorentz spectral density function.

Parameters

- **w** (*numpy.ndarray*) – Frequency array.
- **wk** (*list*) – Frequencies of the Lorentz peaks.
- **sk** (*list*) – Strengths of the Lorentz peaks.
- **temperature** (*float*) – Temperature.
- **gammak** (*float*) – Damping factor.

Returns
Spectral density.

Return type
numpy.ndarray

torchnise.spectral_functions.**spectral_numerical**(*data*, *temperature*)

Creates the poser spectrum from a numerical spectral density.

Parameters

- **data** (*numpy.ndarray*) – numerical spectral density with data[:,0] frequencies and [:,0] the spectral density in the selected unit
- **temperature** (*float*) – Temperature.

Returns
Spectral density.

Return type
numpy.ndarray

1.1.10 torchnise.train_mlnise module

This module demonstrates Hogwild-style training of an MLNISE model using a PyTorch Dataset. It is intended to be imported and called from another script (not from the command line directly).

Example usage in a separate script:

import torch from torchnise.train_mlnise import train_mlnise_hogwild from torchnise.nise import MLNISEModel from torchnise.mlnise_dataset import MLNiseDrudeDataset

Create a dataset dataset = MLNiseDrudeDataset(length=50, total_time=500.0, dt_fs=1.0, n_sites=2)

Instantiate the model model = MLNISEModel() model.share_memory() # for Hogwild

Train trained_model = train_mlnise_hogwild(

model=model, dataset=dataset, num_epochs=20, batch_size=10, num_processes=4, learning_rate=0.1, runname="testrun", realizations=1 # or however many you want

)

torchnise.train_mlnise.**suppress_stdout**()

torchnise.train_mlnise.**train_mlnise_hogwild**(*model: Module*, *dataset: Dataset*, *num_epochs: int = 10*, *num_processes: int = 4*, *learning_rate: float = 0.1*, *runname: str = 'testrun'*, *device: device | None = None*, *scheduler_class: type | None = None*, *scheduler_kwargs: dict | None = None*, *save_interval: int = 5*, *save_checkpoint_fn: Callable[[int, Module], None] | None = None*, *log_fn: Callable[[str], None] | None = None*, *realizations: int = 1*) → Module

Trains a given MLNISE model in a Hogwild (multi-process) fashion. Each process operates on a partition of the dataset with shared parameters.

Parameters

- **model** (*torch.nn.Module*) – The MLNISE model to train. Must have share_memory() called on it if using Hogwild.
- **dataset** (*Dataset*) – The dataset from which samples are drawn (e.g. MLNiseDrudeDataset).
- **num_epochs** (*int, optional*) – Number of epochs to train. Defaults to 10.
- **num_processes** (*int, optional*) – Number of parallel processes for Hogwild. Defaults to 4.
- **learning_rate** (*float, optional*) – Initial learning rate. Defaults to 0.1.
- **runname** (*str, optional*) – Name of the training run (used in logs). Defaults to "testrun".
- **device** (*torch.device, optional*) – Device to run on. If None, uses CPU.
- **scheduler_class** (*type, optional*) – A scheduler class like StepLR or ReduceLROnPlateau. If None, no scheduler is used.
- **scheduler_kwargs** (*dict, optional*) – Keyword arguments for the scheduler. If None, uses default arguments.
- **save_interval** (*int, optional*) – Save checkpoint every N epochs. Defaults to 5.

- **save_checkpoint_fn** (*Callable, optional*) – Function to handle model checkpointing, e.g. *lambda ep, m: torch.save(m.state_dict(), f"checkpoint_{ep}.pt")*. If None, no checkpoints are saved except final.
- **log_fn** (*Callable, optional*) – Logging function. If None, logs go to stdout.
- **realizations** (*int, optional*) – Number of noise realizations to pass to run_nise(...). If you want multi-realization runs, ensure your H shape is (time_steps, realizations, n_sites, n_sites). Defaults to 1.

Returns
The trained model (same object).

Return type
torch.nn.Module

1.1.11 torchnise.units module

THIS IMPLEMENTS THE UNITS AND CONSTANTS FOR THE ENTIRE PROJECT

torchnise.units.**set_units**(*e_unit='cm-1'*, *t_unit='fs'*)

set_the units for time and energy for the entire module

Parameters

- **e_unit** (*string*) – Energy unit to be used. Must be one of "cm-1", "ev", "j"
- **t_unit** (*float*) – Time unit to be used. Must be one of "fs", "ps", "s"

1.1.12 Module contents

PYTHON MODULE INDEX

t

INDEX

O

R

S

T

W