torchnise package

Submodules

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.

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.`**`objective`**`(tau, n, time_array, population)`

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

Parameters:
- **tau** (*float*) – Decay constant.
- **n** (*int*) – Number of states.
- **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, n, time_array, population)`

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

Parameters:
- **tau** (*float*) – Decay constant.
- **n** (*int*) – Number of states.
- **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, n, 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.
- **n** (*int*) – Number of states.
- **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, n, 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.
- **n** (*int*) – Number of states.
- **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, n, time_array, population)`

Objective function using reverse cumulative maximum for fitting decays.

Parameters:
- **tau** (*float*) – Decay constant.
- **n** (*int*) – Number of states.
- **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

torchnise.example_spectral_functions module

This file implements various spectral functions returning the power spectrum.

`torchnise.example_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.example_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.example_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.example_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.example_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.example_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.fft_noise_gen module

This file implements the fftNoiseGEN algorithm for time correlated Noise.

`torchnise.fft_noise_gen.`**`gen_noise`**`(spectral_funcs, dt, shape)`

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

Parameters:
- **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.
- **dt** (*float*) – Time step size.
- **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.

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.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*)

Apply thermal corrections to the non-adiabatic coupling matrix.

Parameters:
- **s** (*torch.Tensor*) – Non-adiabatic coupling matrix.
- **n_sites** (*int*) – Number of sites in the system.
- **realizations** (*int*) – Number of noise realizations.
- **device** (*str*) – Device for computation (“cpu” or “cuda”).
- **e** (*torch.Tensor*) – Eigenvalues of the Hamiltonian at the current time step.
- **eold** (*torch.Tensor*) – Eigenvalues of the Hamiltonian at the previous time step.
- **t_correction** (*str*) – Method for thermal correction (“TNISE”, “MLNISE”).
- **kbt** (*float*) – Thermal energy (k_B * T).
- **mlnise_model** (*nn.Module*) – Machine learning model for MLNISE corrections.

- **mlnise_inputs** (*tuple, optional*) – Inputs for MLNISE model.
- **phi_b** (*torch.Tensor*) – Wavefunction in the eigenbasis.

Returns: Corrected non-adiabatic coupling matrix.

Return type: torch.Tensor

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, mlnise_inputs=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.
- **psio** (*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.
- **mlnise_inputs** (*tuple, optional*) – Inputs for MLNISE model. Defaults to None.

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*)

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.
- **realizations** (*int*) – Number of noise realizations to simulate.
- **psio** (*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.

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, t_correction='None', mode='Population', mu=None, absorption_padding=10000, averaging_method='standard', lifetime_factor=5, max_reps=100000, mlnise_inputs=None, device='cpu'*)

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.
- **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.
- **device** (*str, optional*) – Device for computation (“cpu” or “cuda”). Defaults to “cpu”.

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

torchnise.pytorch_utility module

This file contains some utility functions

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.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

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”

Module contents