Hierarchical Equations of Motion in the Libra Software Package

Story Temen[1], Amber Jain[2], and Alexey Akimov[1]

[1]University at Buffalo - The State University of New York
[2]Indian Institute of Technology Bombay

April 28, 2020

Abstract

We report implementation of a hierarchical equations of motion (HEOM) module within the open-source Libra software. It includes the standard and scaled HEOM algorithms for computing the dynamics of open quantum systems interacting with a harmonic bath. The module allows computing evolution of the reduced density matrix as well as spectral lineshapes. The truncation, filtering, and "update list" schemes as well as OpenMP parallelization allow for further computational saving. The package is written in a mix of C++ and Python languages, delivering the best compromise between user friendliness and efficiency. The Python layer of the package takes advantage of standard Python libraries, such as h5py which allows efficient storage and retrieval of the generated results. The package can be seamlessly used within Jupyter notebooks; its careful design shall provide the maximal convenience and intuitiveness to its users.

Department of Chemistry, University at Buffalo, The State University of New York, Buffalo, NY 14260

[2]*Department of Chemistry, Indian Institute of Technology Bombay, Mumbai, 400076*

We report implementation of a hierarchical equations of motion (HEOM) module within the open-source Libra software. It includes the standard and scaled HEOM algorithms for computing the dynamics of open quantum systems interacting with a harmonic bath. The module allows computing evolution of the reduced density matrix as well as spectral lineshapes. The truncation, filtering, and "update list" schemes as well as OpenMP parallelization allow for further computational saving. The package is written in a mix of C++ and Python languages, delivering the best compromise between user friendliness and efficiency. The Python layer of the package takes advantage of standard Python libraries, such as h5py which allows efficient storage and retrieval of the generated results. The package can be seamlessly used within Jupyter notebooks; its careful design shall provide the maximal convenience and intuitiveness to its users.

Keywords — nonadiabatic dynamics, excited states, system-bath, dissipative environment, open quantum systems, molecular dynamics

1. Introduction

The hierarchical equations of motion formalism (HEOM) is a numerically exact, nonperturbative method to study the dynamics of open quantum systems interacting with harmonic baths. Pioneered by works of Kubo and Tanimura,[1] further developed by Ishizaki, Tanimura, and Fleming,[2] HEOM has found applications in a variety of contexts, such as in spectroscopy and excitation energy and charge transfer studies.[2–8] In particular, the method has found numerous uses in studying exciton transfer and coherence dynamics in biological systems, such as in the Fenna-Mathews-Olson (FMO) complexes.[9–11] The HEOM calculations allowed accurate calculations at low temperatures, whereas perturbative approaches such as Redfield[12,13] or Forster[12,14]fail.[15–22] The HEOM has often been used to obtain numerically exact results for spin-Boson problems across several regimes, and it can be used as a reference methodology for other approximate methods[8,23–25] applied to modeling open quantum systems.

Despite the great utility and potential of the method, it has been a rather niche approach, likely due to its computational complexity. As a result, only a handful of open-source software implementing HEOM dynamics and spectra calculations has been made available. Notably, these are the Fortran codes from the Tanimura group,[26]including a GPU implementation,[27] the PHI code from the Schulten group,[5,6] which emphasizes an efficient CPU parallelization, as well the nanoHub implementation of Kreisbeck and Kramer.[28] Although these codes have been utilized in the past, they are designed as standalone programs with fixed user interaction protocol. A modular all-Python implementation of HEOM is also available within the QuTiP2 program for quantum optics.[29] Although it is flexible in its design, it is primarily meant to be used within Python programs and may not be easily used within C++ codes.

In this work, we present our own implementation of the HEOM method within the Libra library.[30] Libra is a comprehensive, open-source, quantum dynamics package. Adding HEOM to the Libra repertoire not only provides researchers with easy access to the method, but also allows them to integrate the method easily into already established workflows. The HEOM implementation reported in this work emphasizes modularity, transferability, and user friendliness via intuitive and flexible Python interface. Our code includes multiple distinctive features, such as the ability to compute absorption spectra, general system bath coupling, and more. The essential components of the HEOM code are written in C++ in a very modular fashion, so that they can be used within other C++ codes. The openmp parallelization and scaled HEOM are enabled for better efficiency. The low-level components are exposed to Python for their use in Python programs, including in Jupyter notebooks. They can be used in various independent contexts or organized into pre-defined workflows. Our higher-level Python modules provide implementation for the corresponding workflows for computing population dynamics and absorption spectra as well as provide auxiliary functions for storing and analyzing the data produced.

This paper aims to establish the required vocabulary and provide a brief overview of the underlying concepts and methodologies for users and developers to be able make a connection between the implementation and the underlying theory. We explain the package organization and some key algorithmic details. We illustrate the use cases and explain how users can setup the calculations with the HEOM module of the Libra package. Some qualitative features of the dynamics in various types of environments are demonstrated.

2. Scientific Background

2.1. Key definitions of HEOM

The dynamics of a quantum system embedded into an environment ("bath") can often be described by the Hamiltonian of the form:

$$H = \sum_{n,m=0}^{N-1} \ |n\rangle\, H_{\mathrm{nm}}\, \langle m| \ + \sum_{n=0}^{N-1} \ |n\rangle \sum_{b=0}^{N_n-1} \left(\frac{p_{b,n}^2}{2} + \frac{1}{2}\omega_{b,n}^2 x_{b,n}^2 \right) \langle n| \ + \sum_{n=0}^{N-1} \ |n\rangle\, F_n\, \langle n| \ . \quad (1)$$

Here, the first sum represents the electronic Hamiltonian of the quantum system, the second term represents the phonon Hamiltonian describing the dynamics of the bath modes on every electronic state, and the last term represents the linear electron-phonon (vibronic) coupling Hamiltonian describing the perturbation of the quantum system by the environmental degrees of freedom. The matrix elements H_{nn} correspond to state energies, and $H_{\mathrm{nm}} = H_{\mathrm{mn}},\ n \neq m$correspond to electronic (or excitonic, depending on the interpretation) couplings. The numbers are assumed to be independent of the bath degrees of freedom (DOF). The number N represents the total number of quantum states, $N_n$is the number of the bath modes coupled to the n-th quantum state, $p_{b,n}$ and $x_{b,n}$ are the mass-weighted normal modes' momenta and coordinates for a mode b

in a quantum state$n$, and $\omega_{b,n}$ are the normal mode frequencies for a mode b in a quantum state n . Finally, the variableF_n describes the way a quantum state n is coupled to various modes and can be expressed as the following:

$F_n = \sum_{b=0}^{N_b-1} f_{b,n} x_{b,n}$. (2)

Often, the quantum states $\{\ |n\rangle\}$are understood as site (molecular) states, but one can also associate them with a set of quantum states of a single system. The bath operators, F_n, can be more general and can be considered as a user-defined control parameter describing system-bath interactions.

The parameters, $\{f_{b,n}\}$, reflect the type of the electron-phonon interactions the bath induces, which can be quantified by the bath spectral density:

$J_n\left(\omega\right) = \frac{\pi}{2} \sum_{b=0}^{N_n-1} \frac{f_{b,n}^2}{\omega_{b,n}} \delta\left(\omega - \omega_{b,n}\right)$. (3)

In many chemical applications, the baths are well described by the Debye spectral density, which describes an overdamped Brownian motion of the energy gaps in a high-temperature regime:

$J_n\left(\omega\right) = \frac{\eta\gamma\omega^2}{}$

english$\omega^2 + \gamma^2$. (4)

Here, η is the bath reorganization energy, and$\gamma = \frac{\Gamma}{\hbar}$ is the frequency that corresponds to the system-bath coupling energy, Γ. Throughout this account and in the Libra software, we utilize the atomic units, in which$\hbar = 1$, so $\hbar$ may be omitted in some equations, and, numerically, $\gamma = \Gamma$ (although the two variables have different units). Within the HEOM framework, the dynamics of the system's reduced density matrix (RDM), $\rho_{\mathbf{0}}$, can be described by the equations:[25,31,32]

$\dot{\rho}_{\mathbf{n}} = -i\left[H, \rho_{\mathbf{n}}\right] - \sum_{m=0}^{M-1}\left(\sum_{k=0}^{K} n_{\mathrm{mk}}\gamma_{\mathrm{mk}}\right)\rho_{\mathbf{n}} + \rho_{\mathbf{n}}^{(+)} + \rho_{\mathbf{n}}^{(-)} + T_{\mathbf{n}}$. (5a)

$\rho_{\mathbf{n}}^{(+)} = -i\sum_{m=0}^{M-1}\left[Q_m, \sum_{k=0}^{K}\rho_{\mathbf{n}_{\mathrm{mk}}^{+}}\right]$. (5b)

$\rho_{\mathbf{n}}^{(-)} = -i\sum_{m=0}^{M-1}\sum_{k=0}^{K} n_{\mathrm{mk}}(F_{\mathrm{mk}}c_{\mathrm{mk}}\rho_{\mathbf{n}_{\mathrm{mk}}^{-}} - c_{\mathrm{mk}}^{*}\rho_{\mathbf{n}_{\mathrm{mk}}^{-}}F_{\mathrm{mk}})$. (5c)

$T_{\mathbf{n}} = \sum_{m=0}^{M-1}\Delta_K\left[Q_m, \left[Q_m, \rho_{\mathbf{n}}\right]\right]$. (5d)

The HEOM method provides an exact, non-perturbative way of describing the dynamics of a quantum system (the reduced density matrix evolution) in the presence of complex environment, whose degrees of freedom are removed through integration. Although we report the implementation of the HEOM for the spectral density given in Eq. 4, the formulations for more general spectral densities are possible.[4,33–36]

The first term on the right-hand side of Eq. 5a describes the quantum dynamics of an isolated quantum system, the second term – the "quantum friction" - is due to the presence of the environment. The terms Eq. 5b and 5c describe coupling between ADMs of various tiers, and the term Eq. 5d describes the correction to the HEOM truncation.

Here, $\{\gamma_k\}$ and $\{c_k\}$ are the Matsubara frequencies and expansion coefficients, respectively, which describe the decay of the autocorrelation function of the collective coordinate of the bath:

$C\left(t > 0\right) = \sum_{k=0}^{K} c_k exp(-\gamma_k t)$. (6)

Here, K defines the number of Matsubara frequencies. The Matsubara frequencies are defined by the system-bath "interaction" time,$1/\gamma$ (so that γ is a characteristic frequency of the bath) and by temperature $\beta = \frac{1}{k_B T}$:

$\gamma_0 = \gamma$. (7a)

$\gamma_n = \frac{2\pi\nu}{}$

english$\beta = 2\pi\nu k_B T, n \geq 1$, (7b)

With the definition of the spectral density, Eq. 4, the Matsubara expansion coefficients are given by:[31]

$c_0 = \frac{1}{2}\gamma\eta \ \left(\left[\tan\left(\frac{\gamma}{2k_BT}\right)\right]^{-1} - i\right)$, (8a)

$c_k = \frac{4\mathrm{n}}{(2\mathrm{k})^2-()^2} = \frac{4\mathrm{n}}{\beta^2\left[\left(\frac{2\mathrm{n}}{\beta}\right)^2-(\gamma)^2\right]} = 2\eta k_BT\frac{\gamma_0\gamma_n}{\gamma_n^2-\gamma_0^2}, \ k \geq 1.$ (8b)

The parameter Δ_K entering Eq. 5d is a residual sum that is used to truncate the hierarchy:

$\Delta_K = \sum_{n=0}^{\infty}\frac{c_{K+n}}{\gamma_{K+n}}.$ (9)

Finally, the matrices$Q_n = \sum_{a,b=0}^{M-1} \ |a\rangle\, Q_{ab,n}\,\langle b|$ are the matrices describing how a phonon n is coupled to all other electronic states,$\{\ |a\rangle\,,\ a = 0,\ldots,M-1\}$. Here, M is the number of phonon modes. The off-diagonal terms $Q_{ab,n}$ correspond to the coupling of a phonon n to electronic couplings between pairs of states a and b. In the simplest case, when each electronic state is coupled to a single (distinct) phonon, the operator Q takes the form: $Q_n = \ |n\rangle\langle n|$ and $M = N$.

Eq. 5d is a good approximation when the hierarchy of equations is truncated at a finite number of equations. It is not present in the original formulation, with the infinite hierarchy of equations.

2.2. Indexing and hierarchy tracking

Within HEOM, a set of auxiliary density matrices (ADMs),$\{\rho_{\mathbf{n}}\}$, is introduced, with the $\rho_{\mathbf{0}}$being the RDM of the quantum system, and is the main property of interest. The ADMs are indexed by the multi-dimensional index (bolded):

$\mathbf{n} = (n_{00}, n_{01}, \ldots,\ n_{0K},\ n_{10}, n_{11}, \ldots,\ n_{1K}, \ldots, n_{M-1,0}, n_{M-1,1}, \ldots,\ n_{M-1,K}).$ (10a)

Here, M is the number of phonon modes in the system and K+1 is the number of Matsubara frequencies. For simplicity, we take the indexing convention that quantum states' indexing starts with 0 and ends at$M-1$, whereas the Matsubara frequencies' indexing starts with 0 and ends at K. Thus, the length of the multi-dimensional index vector is $d = M*(K+1)$.

$\mathbf{n}_{\mathrm{mk}}^{+} = (n_{00}, \ldots,\ n_{0K}, \ldots, n_{m0}, \ldots,\ n_{\mathrm{mk}}+1, \ldots, n_{\mathrm{mK}}, \ldots, n_{M-1,0}, n_{M-1,1}, \ldots, n_{M-1,K}).$ (10b)

$\mathbf{n}_{\mathrm{mk}}^{-} = (n_{00}, \ldots,\ n_{0K}, \ldots, n_{m0}, \ldots,\ n_{\mathrm{mk}}-1, \ldots, n_{\mathrm{mK}}, \ldots, n_{M-1,0}, n_{N-1,1}, \ldots, n_{M-1,K})$.(10c)

The tier of the ADM, n, is defined as the sum of all elements of the multi-component index vector $\mathbf{n}$:

$n = \mathrm{tier}\,(\mathbf{n}) = \sum_{m=0}^{M-1}\sum_{k=0}^{K} n_{\mathrm{mk}}.$ (11)

2.3. Scaled HEOM

In the current version of Libra, we have also implemented the "scaled" HEOM of Shi et al.[25] According to the method, the scaled ADMs are constructed as:

$\tilde{\rho}_{\mathbf{n}} = \left(\prod_{m=0}^{M-1}\prod_{k=0}^{K} n_{\mathrm{mk}}!\,|c_{\mathrm{mk}}|^{n_{\mathrm{mk}}}\right)^{-1/2}\rho_{\mathbf{n}}.$ (12)

Note that $\tilde{\rho}_{\mathbf{0}} = \rho_{\mathbf{0}}$, that is one may propagate the dynamics in terms of the scaled ADMs, but the evolution of the first (zero tier) member of the hierarchy would correspond to that of the original reduced density matrix of the quantum system. In terms of the scaled ADMs, the HEOM takes the form:

$\frac{d\tilde{\rho}_{\mathbf{n}}}{\mathrm{dt}} = -i\,[H, \tilde{\rho}_{\mathbf{n}}] - \sum_{m=0}^{M-1}\left(\sum_{k=0}^{K} n_{\mathrm{mk}}\gamma_{\mathrm{mk}}\right)\tilde{\rho}_{\mathbf{n}} + \tilde{\rho}_{\mathbf{n}}^{(+)} + \tilde{\rho}_{\mathbf{n}}^{(-)} + \tilde{T}_{\mathbf{n}}.$ (13a)

$\tilde{\rho}_{\mathbf{n}}^{(+)} = -i\sum_{m=0}^{M-1}\left[Q_m, \sum_{k=0}^{K}\sqrt{(n_{\mathrm{mk}}+1)\,|c_{\mathrm{mk}}|}\tilde{\rho}_{\mathbf{n}_{\mathrm{mk}}^{+}}\right].$ (13b)

$\tilde{\rho}_{\mathbf{n}}^{(-)} = -i\sum_{m=0}^{M-1}\sum_{k=0}^{K}\sqrt{n_{\mathrm{mk}}/\,|c_{\mathrm{mk}}|}(F_{\mathrm{mk}}c_{\mathrm{mk}}\tilde{\rho}_{\mathbf{n}_{\mathrm{mk}}^{-}} - c_{\mathrm{mk}}^{*}\tilde{\rho}_{\mathbf{n}_{\mathrm{mk}}^{-}}F_{\mathrm{mk}}).$ (13c)

$\tilde{T}_{\mathbf{n}} = \sum_{m=0}^{M-1} \Delta_K \left[Q_m, \left[Q_m, \tilde{\rho}_{\mathbf{n}}\right]\right]$. (13d)

These equations are isomorphic to Eqs. 5 and reduce to them when $\sqrt{(n_{\mathrm{mk}}+1)\left|c_{\mathrm{mk}}\right|} \rightarrow 1$in Eq. 13b and $\sqrt{n_{\mathrm{mk}}/\left|c_{\mathrm{mk}}\right|} \rightarrow n_{\mathrm{mk}}$in Eq. 13c. The main advantage of the scaled HEOM Eqs. 13 is that the scaled densities become negligible for high tiers much more quickly than the unscaled ones. Based on this property, the truncation and filtering can become much more efficient, allowing the converged results to be achieved using smaller K and L values, where L is the maximal tier of ADM.

2.4. Filtering

The main complexity of the HEOM is the large (factorially-growing) number of ADMs and the corresponding equations to solve. To combat this, the hierarchy is truncated at a certain tier level, L, and a minimal number of Matsubara frequencies should be used. However, the results of the calculations should be converged with respect to both parameters. The use of a finite number of ADMs is compensated by the truncation terms, Eqs. 5d and 13d.

In addition to the use of scaled version of HEOM and truncation correction, the current implementation allows filtering equations based on the $\left|\rho_{\mathbf{n}}\right| < tol_1$ or$\left|\dot{\rho}_{\mathbf{n}}\right| < tol_2$ criteria (for scaled or unscaled ADMs and their derivatives). The norm of the matrix (whether ADM or its time-derivative) is computed as the maximal magnitude of all matrix elements.[25] The first criterion is used to turn off the computation of any terms involving very small ADMs. The second criterion is used to turn off the computation of the time-derivatives of the ADMs when such derivatives are expected to be very small. We construct two lists to keep track of the information on whether to flag out (with 0) or flag in (with 1) the use of the corresponding ADMs or recalculation of the corresponding time-derivatives. These lists can be updated with a user-defined frequency to minimize the overhead in their updates. This approach is similar to the Verlet-list technique often used in classical molecular dynamics simulations.

2.5. Spectra

The system's RDM, propagated via HEOM,$\rho_{\mathbf{0}}(t)$, can be used to compute the line shapes of the absorption spectrum as a Fourier transform of the dipole autocorrelation function (ACF):

$I(\omega) = \frac{1}{\pi} \operatorname{Re}\left[\int_0^{\infty} \mathrm{d}t e^{i\omega t} \left\langle u(t)\, u(0)\right\rangle_g\right]$, (14)

The ACF is computed as a trace of the product of the time-dependent density matrix, $\rho_{\mathbf{0}}(t)$, and the (time-independent) transition dipole moment matrix, μ, assuming one starts in the ground electronic state:

$\left\langle u(t)\, u(0)\right\rangle_g = Tr\left[\rho_{\mathbf{0}}(t)\,\mu\right]$. (15)

The initial density matrix is chosen as the ground state,$\rho_0(0) = \ |g\rangle\langle g|$.

3. Software Description

3.1. Software Architecture

The package consists of the Python and C++ layers. The former is represented by the "libra_py.dynamics.heom" module and implements the high-level functions of immediate importance to user. The latter is represented by the "dyn/heom" library (libheom when compiled) in the core of the Libra software and implements various functions that are used to construct the algorithms at the higher level, including in the "libra_-py.dynamics.heom" module (Figure 2). These functions are implemented in C++, although most of them are exposed to Python via the Boost.Python library.[37] They may be of higher interest to the methodology developers.

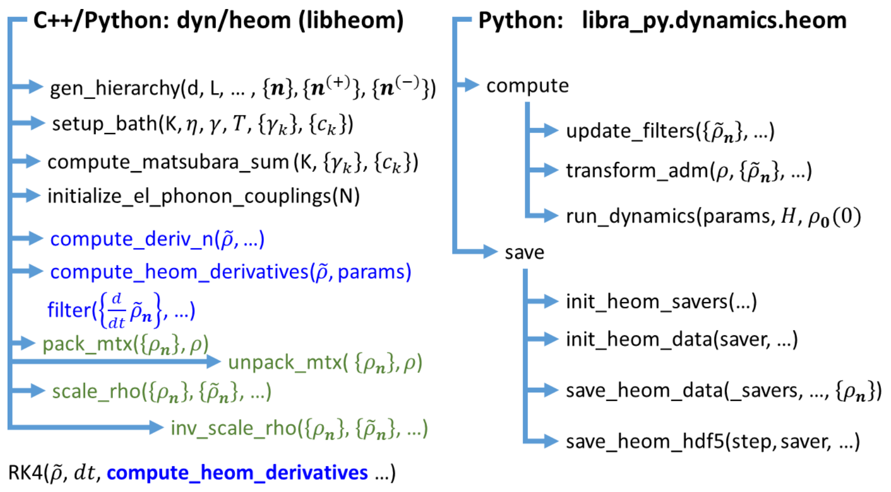
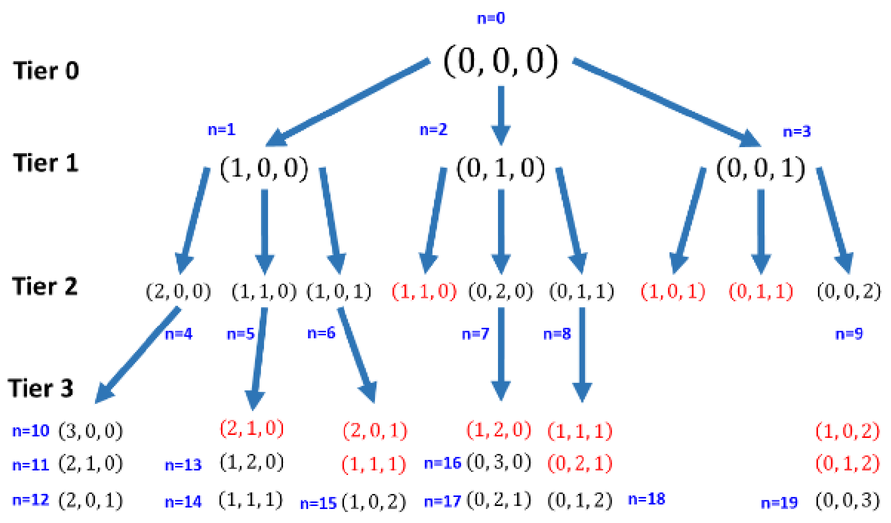

C++/Python: dyn/heom (libheom)
gen_hierarchy(d, L, ... , $\{\boldsymbol{n}\}, \{\boldsymbol{n}^{(+)}\}, \{\boldsymbol{n}^{(-)}\}$)
setup_bath(K, $\eta, \gamma, T, \{\gamma_k\}, \{c_k\}$)
compute_matsubara_sum (K, $\{\gamma_k\}, \{c_k\}$)
initialize_el_phonon_couplings(N)
compute_deriv_n($\tilde{\rho}$, ...)
compute_heom_derivatives($\tilde{\rho}$, params)
filter($\{\frac{d}{dt}\tilde{\rho}_{\boldsymbol{n}}\}$, ...)
pack_mtx($\{\rho_{\boldsymbol{n}}\}, \rho$)
unpack_mtx($\{\rho_{\boldsymbol{n}}\}, \rho$)
scale_rho($\{\rho_{\boldsymbol{n}}\}, \{\tilde{\rho}_{\boldsymbol{n}}\}$, ...)
inv_scale_rho($\{\rho_{\boldsymbol{n}}\}, \{\tilde{\rho}_{\boldsymbol{n}}\}$, ...)
RK4($\tilde{\rho}$, dt, compute_heom_derivatives ...)
Python: libra_py.dynamics.heom
compute
update_filters($\{\tilde{\rho}_{\boldsymbol{n}}\}$, ...)
transform_adm($\rho, \{\tilde{\rho}_{\boldsymbol{n}}\}$, ...)
run_dynamics(params, H, $\rho_0(0)$)
save
init_heom_savers(...)
init_heom_data(saver, ...)
save_heom_data(_savers, ..., $\{\rho_{\boldsymbol{n}}\}$)
save_heom_hdf5(step, saver, ...)

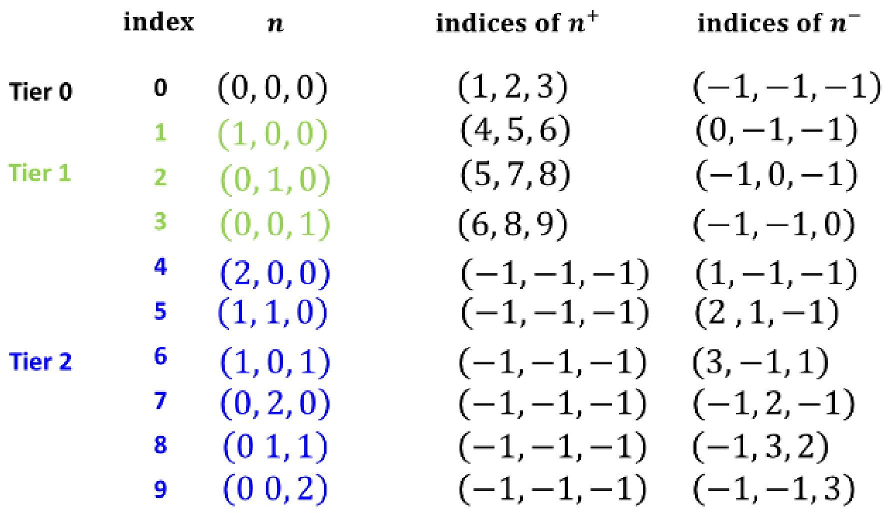
Figure 1. The architecture of the HEOM modules (Python and C++) in Libra.

The C++ functions can be classified into 3 main groups: a) initialization (Figure 1, left, black); b) computation (Figure 1, left, blue); and c) auxiliary functions (Figure 1, left, green). The first group includes the functions: "gen_hierarchy" that constructs the list of multidimensional index vectors, Eq. 10a, "setup_bath" that computes the Matsubara frequencies, $\{\gamma_n\}$, and the corresponding coefficients, $\{c_n\}$, all according to Eqs. 7 and 8, respectively. "compute_matsubara_sum" that computes the sum of type Eq. 9, "initialize_el_phonon_couplings" that computes the matricesQ_m used in Eqs. 5 or 13 as$Q_m = \ |m\rangle \langle m|$. The latter is useful when user does not desire to specify such matrices in the input and wants to use the assumption that each states is coupled to only one bath mode.

The generation of the ADMs' indices, the indexing convention, and the calculation of the mappings by the function "gen_hierarchy" is worth additional discussion. In our implementation, the hierarchy of ADMs is constructed as shown in Figure 2. The function creates the tree structure representing the desired hierarchy (Figure 2a) via a recursive algorithm. Starting with a d-dimensional all-zero vector, one can increment the index of each of the d components by one to generate d new vectors. These vectors are the "proposed" children nodes for a given iteration (hierarchy level increment). Before each of the new proposed vectors is added to the final list of all the hierarchy indices$\{\mathbf{n}\}$, it is compared to all of the previously added children of the same tier (level) to ensure it is unique. The duplication is not possible among the vectors of different tiers, so the comparison to all the previously added vectors is not needed and is not efficient computationally. Considering only the members of the same tier in the uniqueness determination is one of the time-consuming elements of the algorithm. Generating the hierarchy with all possible (non-unique) vectors first and discarding them later is very time and memory consuming algorithm, and it can become unpractical already for relatively low maximal tiers. Therefore, the uniqueness of the added vectors is verified on the fly during the construction. This greatly reduces both CPU and memory requirements of the algorithm.

The newly added unique vectors at a given tier level (children of the nodes of a lower tier) are used as the input for another iteration. Each of them would be considered a starting vector, to which the procedure described above would be applied. As a result, a tree structure is created and traversed as illustrated in Figure 2a. The traversal is organized in the width-first fashion, such that the lower-tier indices appear in the $\{\mathbf{n}\}$ list before the d-dimensional indices of higher tiers (Figures 1a, 1b). According to Eq. 10, the dimensionality of each multi-index vectors (a node of the tree) is $d = M(K + 1)$.





	index	n	indices of n^+	indices of n^-
Tier 0	0	(0, 0, 0)	(1, 2, 3)	(−1, −1, −1)
	1	(1, 0, 0)	(4, 5, 6)	(0, −1, −1)
Tier 1	2	(0, 1, 0)	(5, 7, 8)	(−1, 0, −1)
	3	(0, 0, 1)	(6, 8, 9)	(−1, −1, 0)
	4	(2, 0, 0)	(−1, −1, −1)	(1, −1, −1)
	5	(1, 1, 0)	(−1, −1, −1)	(2 , 1, −1)
Tier 2	6	(1, 0, 1)	(−1, −1, −1)	(3, −1, 1)
	7	(0, 2, 0)	(−1, −1, −1)	(−1, 2, −1)
	8	(0 1, 1)	(−1, −1, −1)	(−1, 3, 2)
	9	(0 0, 2)	(−1, −1, −1)	(−1, −1, 3)

(a) (b)

Figure 2. Schematics describing the algorithm to construct the hierarchy of ADMs and compute their mappings. In this example,$d = 3$. a) Each ADM of a given tier is coupled to d ADMs of the following tier. Indices in red are duplicates of the already added ADM indices and are discarded. b) List of enumerated d-dimensional index vector for tiers 0, 1, and 2. Note that vectors of a given tier follow before the vectors of the following tier and there are not duplicate vectors. The matrices n^+ and n^- contain the indices of the ADMs that are coupled to a given ADM via raising or lowering one of its components by one.

The tree traversal returns an enumerated list of d-dimensional vectors. In computing time-derivatives of the ADMs, Eqs. 5 and 13, one also needs to quickly access the $\rho_{\mathbf{n}^{(+)}}$ and$\rho_{\mathbf{n}^{(-)}}$ ADMs knowing just the index of the$d$-dimensional vector $\mathbf{n}$. For quick access, the "gen_hierarchy" function also computes the mapping tables for such quick access (Figure 2b). The mapping $\mathbf{n}^+$, for instance, can be thought as a $N_{\mathrm{adm}} \times d$ matrix, such that its element $(i,\ j)$ contains a sequential index (scalar integer) of the $\rho_{\mathbf{n}_j^+}$, if the analogous sequential (scalar) index of the vector $\mathbf{n}$ is i. Likewise, the element $(i,\ j)$ of the matrix$\mathbf{n}^-$ contains the sequential index of the$\rho_{\mathbf{n}_j^-}$ such that the index of the vector$\mathbf{n}$ is i. In other words, if we start with the i-th vector, $\mathbf{n}$, and increment its j-th element by one, the element $n^+[i][j]$ will return the index of such a multi-component vector in the overall set of vectors. If there is no such vector (e.g. due to the truncation of the hierarchy), the matrix elements are set to -1. This concept is better understood from a careful examination of Figure 2b.

The second group of C++/Python functions includes: “compute_deriv_n” which computes the time-derivative of a given ADM, n, according to Eqs. 5 or 13, “compute_heom_derivatives” that performs the computation of time-derivatives of all ADMs (this is where the openMP parallelization is currently involved), and “filter” function that updates the flags which determine whether each of the ADMs should be used in computing the right-hand side of the ADM time-derivatives in Eqs. 5 and 13. In addition, the Python-exposed function “compute_heom_derivatives” has a particular importance in constructing the solver of the HEOM.

The integration of the HEOM is done via a general-purpose 4-th order Runge-Kutta integrator (RK4) method, implemented in Libra in the “libintegrators” sub-library. It solves a set of differential equations that can be represented in a matrix format as:

$\dot{X} = f(X; params)$. (16)

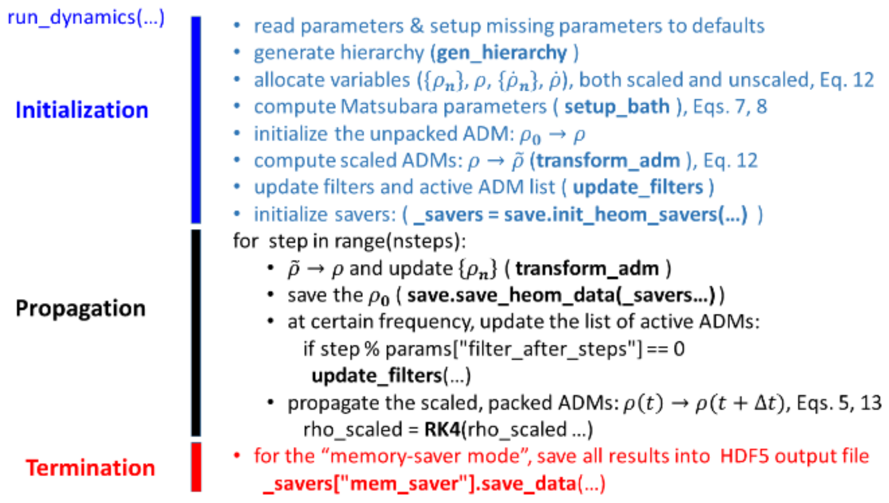
One of the parameters the RK4 function accepts is a function itself, the one that computes the time-derivatives of the ADMs. Passing functions as arguments is one of the convenient features of Python, since the functions are treated as objects and are similar to other types of arguments in this regard. Unlike C++, where one needs to specify the signature of the function that is being passed, Python doesn’t require such extra care. Of course, the RK4 function that calls the argument function would need to know the signature of the latter as well as how to unpack the returned results, so the developer needs to ensure the consistency of the expected function call invocation with the function’s signature and expected output. However, these details can be hidden in the function being called, whereas the interface of the RK4 function won’t change.

Since every function passed to the RK4 general-purpose solver may have a varied number of parameters, the RK4 function also expects a dictionary of parameters. These parameters would then be passed to the RK4 argument-function. In the present context, the “compute_heom_derivatives” is the function that is being passed to the “RK4” solver. The parameters of “compute_heom_derivatives” are therefore passed to the “RK4” function a dictionary of arguments. This dictionary contains parameters such as Matsubara frequencies, Matsubara coefficients, electron-phonon coupling matrices, and so on. The parameters dictionary can have extra key-value pairs, not needed directly by the “compute_heom_derivatives”, but used to control the execution of the RK4 algorithm. This way, one may define a common dictionary of all parameters and pass it to many functions with the idea that only the parameters that are relevant to the calculations will be utilized.

The feature of passing functions as arguments of other functions is also utilized in other Libra modules, such as in the trajectory surface hopping approaches. There, one can define a Python function that would compute some essential properties of the system (Hamiltonian, derivatives, etc.) for a given input of coordinates. Such a Python function can be passed to the dynamical algorithms (e.g. propagation) using a common interface. The user is then free to redefine the arguments functions to suit their needs – to either define a model Hamiltonian or to setup a call of external electronic structure packages and extracting the needed information from their output. In this mode, the user doesn’t need to know the details of the implementation of the dynamical procedures and only needs to care about the input/output signatures and the internal computations of the function passed as an argument.

Finally, the third group of C++/Python functions comprises the auxiliary functions “scale_rho” and “inv_-scale_rho” to convert between pristine and scaled ADMs, Eqs. 12, and the functions “pack_mtx” and “un-pack_mtx” to transform between two representations of the ADMs. Namely, the set of all ADMs used in HEOM calculations,$\{\rho_n,\ n = 0, \ldots, N_{\text{adm}} - 1\}$, can be represented as a list of $M \times M$ matrices. However, the RK4 function implemented in the “libintegrators” library expects a single matrix of the function arguments. Because of this expected format, the list of the $M \times\ M$ ADMs is packed into a single$N_{\text{adm}} M \times M$ matrix ρ, which can then be passed into the RK4 function. The function $f(\ldots)$ appearing in Eq. 16 is the “compute_heom_derivatives”. Internally, the input ADMs matrix is unpacked inside of this function, and the derivatives are computed for each ADM independently. In fact, this is where we have added the OpenMP parallelization. The computed time-derivatives,$\{\frac{d}{\mathrm{dt}}\tilde{\rho}_{\mathbf{n}}\}$, are then packed into a single matrix format to be used within the general-purpose RK4 integrator.

The "scaled" ADMs and their time-derivatives are considered fundamental in our implementation: the integration and the computation of the time-derivatives, as well as filtering and truncation, are done within the framework of the "scaled" variable, with the "unscaled" ones being a special case. The unscaled variables are only updated before storing them to the results (memory or on disk), but do not directly participate in the dynamics.



```
• read parameters & setup missing parameters to defaults
• generate hierarchy (gen_hierarchy )
• allocate variables ($\{\rho_n\}$, $\rho$, $\{\dot{\rho}_n\}$, $\dot{\rho}$), both scaled and unscaled, Eq. 12
• compute Matsubara parameters ( setup_bath ), Eqs. 7, 8
• initialize the unpacked ADM: $\rho_0 \rightarrow \rho$
• compute scaled ADMs: $\rho \rightarrow \tilde{\rho}$ (transform_adm ), Eq. 12
• update filters and active ADM list ( update_filters )
• initialize savers: ( _savers = save.init_heom_savers(...) )
for  step in range(nsteps):
    • $\tilde{\rho} \rightarrow \rho$ and update $\{\rho_n\}$ ( transform_adm )
    • save the $\rho_0$ ( save.save_heom_data(_savers...) )
    • at certain frequency, update the list of active ADMs:
        if step % params["filter_after_steps"] == 0
         update_filters(...)
    • propagate the scaled, packed ADMs: $\rho(t) \rightarrow \rho(t + \Delta t)$, Eqs. 5, 13
        rho_scaled = RK4(rho_scaled ...)
• for the "memory-saver mode", save all results into  HDF5 output file
   _savers["mem_saver"].save_data(...)
```

Figure 3. The computational workflow of the "run_dynamics" function of the "dynamics.heom" module.

The second part of the current HEOM implementation is the Python module, "dynamics.heom" (Figure 1, right), which encompasses a number of workflows for intuitive and convenient user-software interaction. The module contains two main sub-modules "compute" and "save". The "compute" sub-module defines high-level functions for transforming and handling data as well as a "run_dynamics" function, which serves as the entry point for users to execute HEOM calculations. The workflow of this function (Figure 3) involves calling both the Python-level functions and the Python-exposed C++ functions of the "dyn/heom" module.

It is worth describing the structure of the main computational unit – the "run_dynamics" function – in somewhat more detail. First, the simulation parameters are passed into this function as a Python dictionary. A local copy is first created. The keywords present in this dictionary are checked and if some expected keywords are not found (e.g. user didn't specify them), they are initialized to the default values defined in the "run_dynamics" function. The operation is done via the "check_input" function of the util.libutil library of Libra:

import util.libutil as comn

comn.check_input(params, default_params, critical_params)

One can specify a list of keywords that are considered critical (needed for the execution of the code), but that cannot generally be assigned to default values (e.g. system-specific). The program terminates if the input dictionary does not provide any key-value pair with the key names listed in the "critical_params" variable. The function "check_input" would modify the local copy of the control parameters dictionary, to setup the undetermined variables to the default values, but it would not affect the original instance of the parameters dictionary, which is important in a situation when the same parameters dictionary is passed into several consecutive calls of the "run_dynamics" function. On the implementation side, we exercise caution to copy the input dictionary by value (via copy constructor) to the internal variables, whose key-value pairs may be reset to some default values. The approach of setting up the default values and indicating the critical parameters is widely used in other modules of the Libra code, including the modules for fully quantum ("exact") and trajectory-surface hopping ("tsh") calculations.

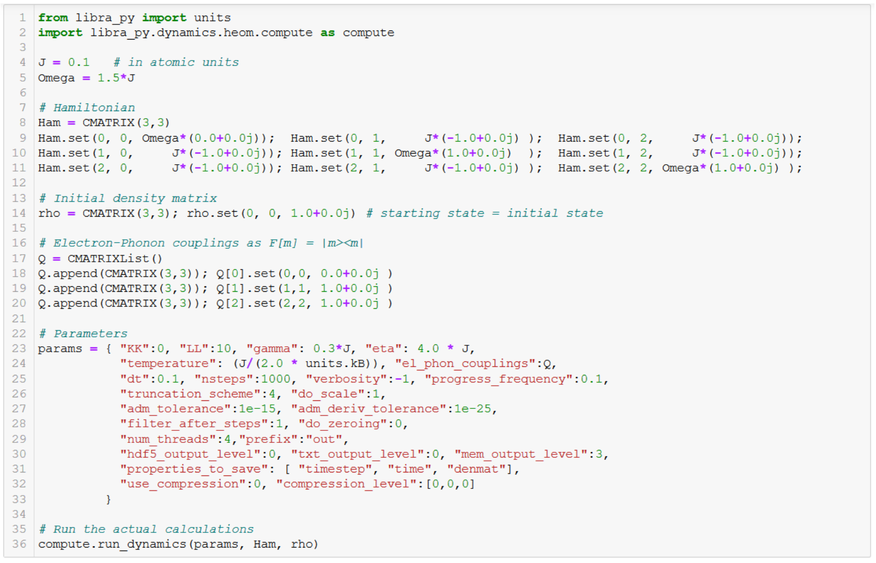
Another sub-module of the "dynamics.heom" is the "save" module. It implements methods to store results of simulations in HDF5 format. The archiving of the results computed in the dynamics can be done in a regular mode, such that the HDF5 files are updated on the fly. However, this mode is very time-consuming and an alternative mode, "memory" ("mem"), is preferred. In the "mem" mode, all the propagated variables are stored in the operating system (RAM) for the entire duration of the calculations and are saved to the disk only when the propagation phase is completed. This saving mode is much faster, but it has an obvious drawback of potential data loss if the calculations are terminated prematurely. In addition, lengthy and complex simulations in this mode may require an increased amount of operating memory the user needs to reserve on their computational systems.

Finally, the "plot" module is present in the "dynamics.heom" module and is intended to provide "out-of-box" recipes for plotting results of calculations. However, such printing is rather straightforward and may be easier to customize in the users' scripts. For these reasons, the "plot" module can be regarded as a placeholder for the time being. It is worth mentioning that an analogous "plot" module is present in other higher-level modules of Libra, such as "tsh" and "exact". The plotting of the results of those calculations may be somewhat more involved and the "plot" does define certain plotting recipes.

3.2. Software Functionalities and Sample code snippets.

3.2.1. Calculation of density matrix evolution

An example snippet to run the HEOM calculations using Libra is shown in Figure 4. As we discuss above, the main function to call is the "run_dynamics" of the "libra_py.dynamics.heom" module. The major fraction of the snippet only sets up the parameters needed to run the calculations. In particular, one can define the system's Hamiltonian matrix as one of the Libra's built-in data types "CMATRIX" that holds the complex-valued matrices of arbitrary dimensions. In this example, we illustrate a setup for a 3-level system, similar to the one utilized previously by Shi et al.[25] In this model, two higher energy states are degenerate and are separated from the ground state by the $\Omega = 1.5\ J$ gap, where J is the electronic coupling. Unlike the model of Shi et al., in this demonstration we make all pairs of states coupled to each other through the matrix elements$H_{\mathrm{ij}} = H_{\mathrm{ji}} = -J,\ i \neq j$. We start the simulation with a ground-state density matrix,$\rho = \ |0\rangle\langle 0|$.



```
from libra_py import units
import libra_py.dynamics.heom.compute as compute

J = 0.1   # in atomic units
Omega = 1.5*J

# Hamiltonian
Ham = CMATRIX(3,3)
Ham.set(0, 0, Omega*(0.0+0.0j));  Ham.set(0, 1,     J*(-1.0+0.0j) );  Ham.set(0, 2,     J*(-1.0+0.0j));
Ham.set(1, 0,     J*(-1.0+0.0j)); Ham.set(1, 1, Omega*(1.0+0.0j)  );  Ham.set(1, 2,     J*(-1.0+0.0j));
Ham.set(2, 0,     J*(-1.0+0.0j)); Ham.set(2, 1,     J*(-1.0+0.0j) );  Ham.set(2, 2, Omega*(1.0+0.0j) );

# Initial density matrix
rho = CMATRIX(3,3); rho.set(0, 0, 1.0+0.0j) # starting state = initial state

# Electron-Phonon couplings as F[m] = |m><m|
Q = CMATRIXList()
Q.append(CMATRIX(3,3)); Q[0].set(0,0, 0.0+0.0j )
Q.append(CMATRIX(3,3)); Q[1].set(1,1, 1.0+0.0j )
Q.append(CMATRIX(3,3)); Q[2].set(2,2, 1.0+0.0j )

# Parameters
params = { "KK":0, "LL":10, "gamma": 0.3*J, "eta": 4.0 * J,
           "temperature": (J/(2.0 * units.kB)), "el_phon_couplings":Q,
           "dt":0.1, "nsteps":1000, "verbosity":-1, "progress_frequency":0.1,
           "truncation_scheme":4, "do_scale":1,
           "adm_tolerance":1e-15, "adm_deriv_tolerance":1e-25,
           "filter_after_steps":1, "do_zeroing":0,
           "num_threads":4,"prefix":"out",
           "hdf5_output_level":0, "txt_output_level":0, "mem_output_level":3,
           "properties_to_save": [ "timestep", "time", "denmat"],
           "use_compression":0, "compression_level":[0,0,0]
         }

# Run the actual calculations
compute.run_dynamics(params, Ham, rho)
```

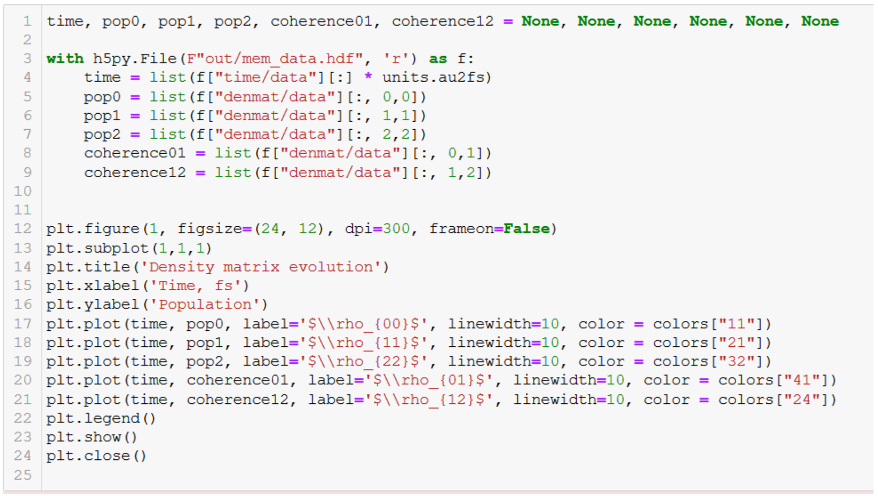
Figure 4. An example snippet to run the HEOM calculations within a Jupyter notebook.

The user needs to define the parameters of the bath, simulation, output, and choose the approximation control options. The user can also setup a desired type of electron-phonon coupling (system-bath correlation). In this example, we define the coupling matrices (representing the operators Eq. 2) to reflect one of the simplest scenarios – the coupling of a single bath mode to a single electronic state, such that$Q_m = \ |m\rangle \langle m| \ , m = 1, 2$. The zero's mode is not coupled to any of the electronic states, $Q_0 = 0$. The Q_m matrices define the type of correlation of bath and quantum degrees of freedom and may be a subject of particular investigation. For instance, one may want to couple the bath's phonon 0 to the lowest two states at the same time, in which case one could define$Q_0 = \ |0\rangle \langle 0| \ + \ |1\rangle \langle 1| \ $. One can also anti-correlate the response of states 0 and 1 to the same phonon, say 1, in which case one would define$Q_1 = \ |0\rangle \langle 0| \ - \ |1\rangle \langle 1| \ $.

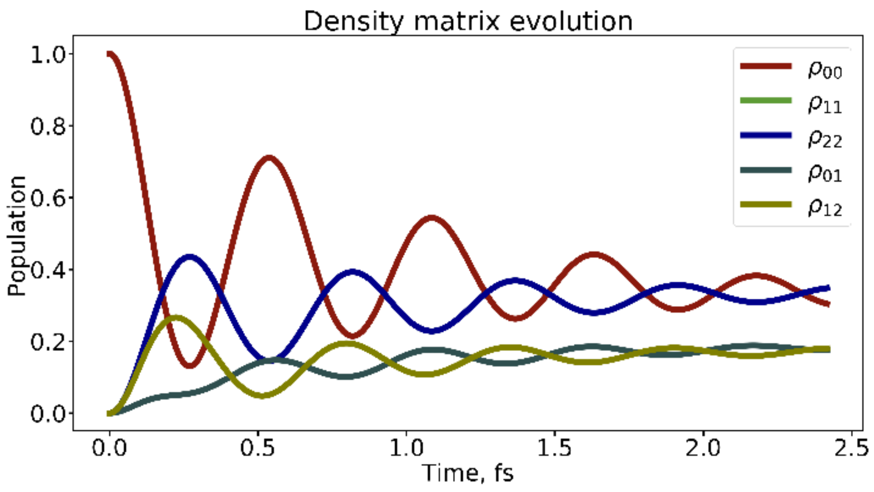
The parameters dictionary "params" in the above snippet defines the critical parameters affecting the whole HEOM set – the number "KK" that defines the number of Matsubara terms (KK + 1), the highest tier level of the hierarchy ("LL"), as well as the properties of the bath – reorganization energy ("eta"), system-environment interaction rate, which can be associated with the decoherence rate[13]("gamma"), and bath temperature ("temperature").

The snippet also illustrates the concept of the on-demand print-out/storage of the results. With the variable "mem_output_level" set to 3, all possible properties (so far "timestep", "time", and "denmat") could be saved in the output files. However, one needs to explicitly include these keywords in the list of "properties_-to_save". The output level variable determines the outputs of which dimensionality/complexity can be saved. The value of 0 would normally save the scalar data, the value of 1 would save everything up to 1 D arrays (e.g. total energy vs. time), the value of 3 would save a 3D data such as a series of density matrices (each is a 2D entry) as a function of time (adding another dimension). However, one may have many irrelevant properties (as far as the goals of a particular simulation are concerned) one doesn't want to save. The "properties_to_save" variable allows one to exclude saving such data by not listing the corresponding data names in it. Note also that listing the properties to save is required for them to be saved. Thus, just setting the "mem_output_level" parameter to 3 will not save anything if the "properties_to_save" list is empty.

A few other variables of note are the "do_scale" which turns on/off the scaled HEOM, Eq. 13, the "truncation_scheme", which chooses one of four truncation schemes, Eqs. 5d, 13d, the "adm_tolerance", which sets the threshold for discarding the ADMs,$|\tilde{\rho}_{\mathbf{n}}| < tol_1$, and the "adm_deriv_tolerance", which sets the threshold for skipping the integration of some HEOM equations,$\left|\frac{d}{\mathrm{dt}}\tilde{\rho}_{\mathbf{n}}\right| < tol_2$. With the variable "filter_after_steps" one can control the frequency of the updates of the active HEOM equations to be integrated as well as the updates of the "no-zero" ADMs to be used in computing time-derivatives. Ideally, this update can be done at every step, but one can gain some computational savings when increasing this parameter. It should be noted that the convergence with respect to these parameters should always be tested to ensure the meaningfulness of the computed results.



```
time, pop0, pop1, pop2, coherence01, coherence12 = None, None, None, None, None, None

with h5py.File(F"out/mem_data.hdf", 'r') as f:
    time = list(f["time/data"][:] * units.au2fs)
    pop0 = list(f["denmat/data"][:, 0,0])
    pop1 = list(f["denmat/data"][:, 1,1])
    pop2 = list(f["denmat/data"][:, 2,2])
    coherence01 = list(f["denmat/data"][:, 0,1])
    coherence12 = list(f["denmat/data"][:, 1,2])


plt.figure(1, figsize=(24, 12), dpi=300, frameon=False)
plt.subplot(1,1,1)
plt.title('Density matrix evolution')
plt.xlabel('Time, fs')
plt.ylabel('Population')
plt.plot(time, pop0, label='$\\rho_{00}$', linewidth=10, color = colors["11"])
plt.plot(time, pop1, label='$\\rho_{11}$', linewidth=10, color = colors["21"])
plt.plot(time, pop2, label='$\\rho_{22}$', linewidth=10, color = colors["32"])
plt.plot(time, coherence01, label='$\\rho_{01}$', linewidth=10, color = colors["41"])
plt.plot(time, coherence12, label='$\\rho_{12}$', linewidth=10, color = colors["24"])
plt.legend()
plt.show()
plt.close()

```

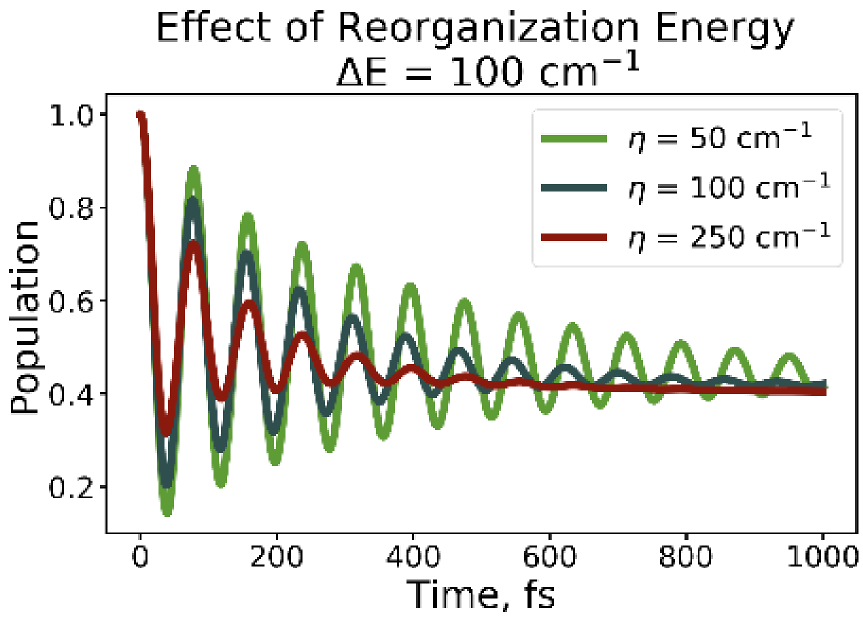


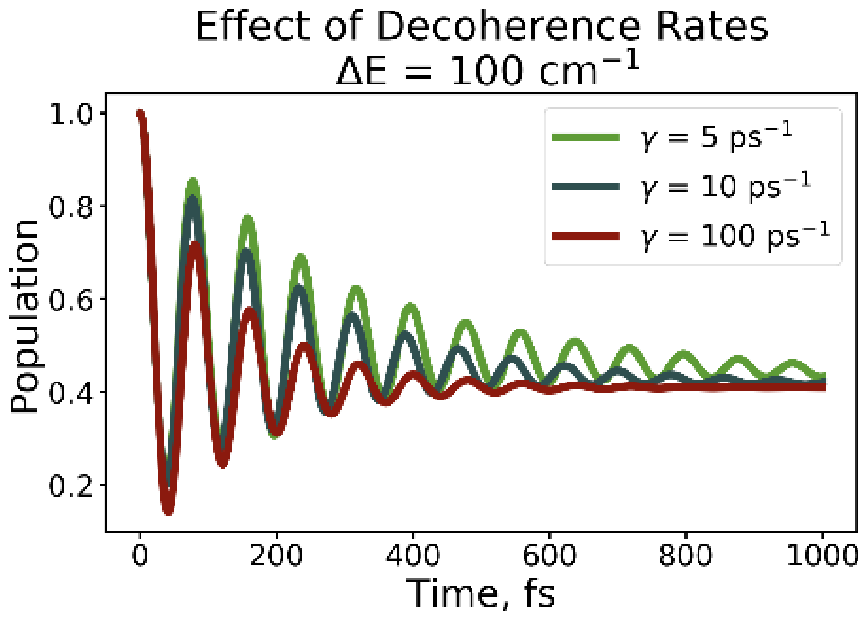
(a) (b)

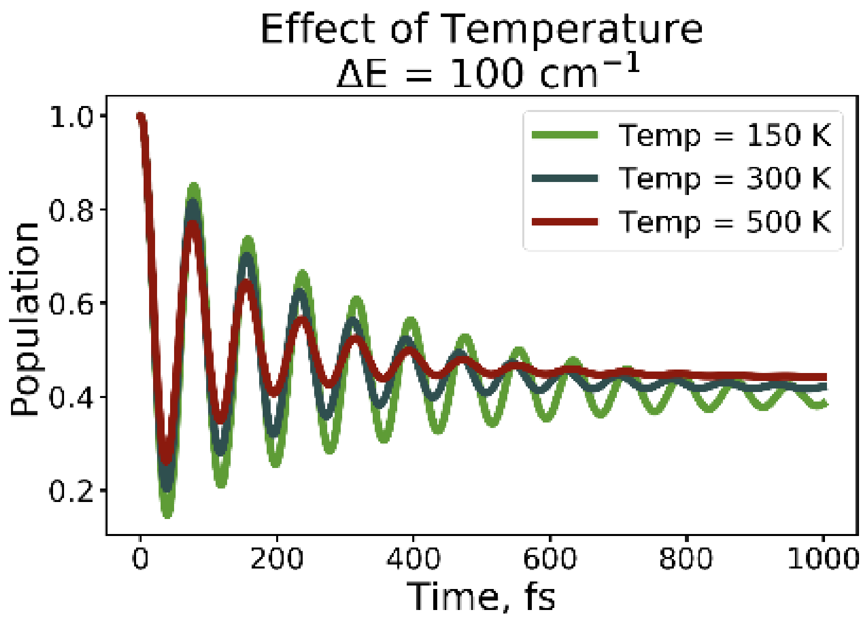
Figure 5. Density matrix evolution in the 3-state problem described in snippet, Figure 4: a) a snippet to read and plot the computed density matrix from the generated HDF5 file; b) the time-dependence of populations (diagonal matrix elements) and coherences (off-diagonal matrix elements).

The results of HEOM calculations are stored in an HDF5-format file, called by default "mem_data.hdf" and stored in the automatically-created directory defined by the "prefix" keyword of the input parameters. The "dynamics.heom" module utilizes the "libra_py.data_savers" module, which defines the convenience classes to temporarily store arbitrary data in RAM, to save it in the HDF5 files all-in-once or on-the-fly, and to control other relevant properties of the produced HDF5 files, such as internal storage type and the compression level. In turn, the "data_savers" module utilizes the "h5py" library[38] that enables Python to create and read the HDF5 files.

The HDF5 files produced by the "heom.save" module have the data groups called identically to the entries of the "properties_to_save" list in the input parameters dictionary. Each data group contains the metadata about the stored entry (e.g. the dimensionality of the archive) and the data itself. The latter is stored in the dataset always called "data", such that one can access the density matrix e.g. via the following instruction "f["denmat/data"][:, :, :]" (e.g. see Figure 5a). The results can be easily plotted using the standard "matplotlib" library[39,40] (Figure 5b).

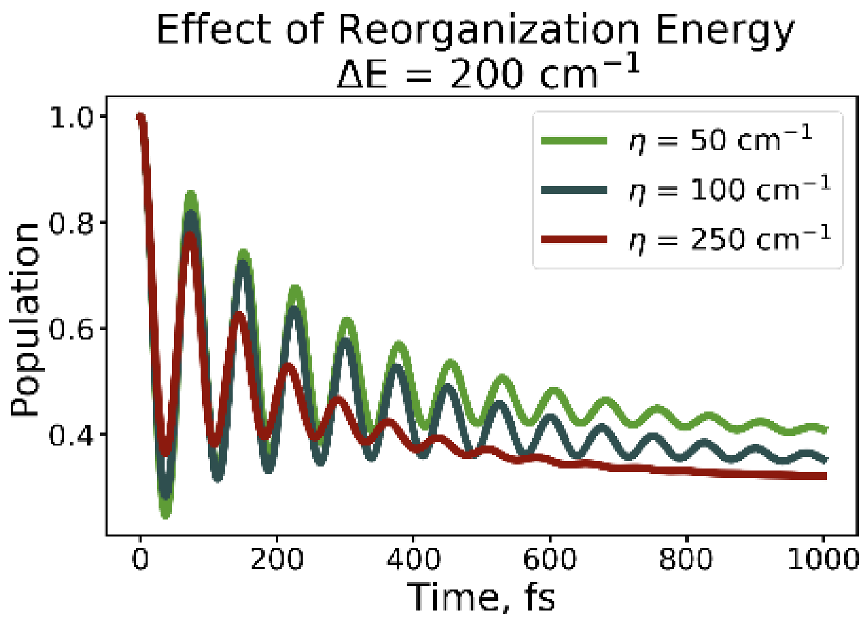




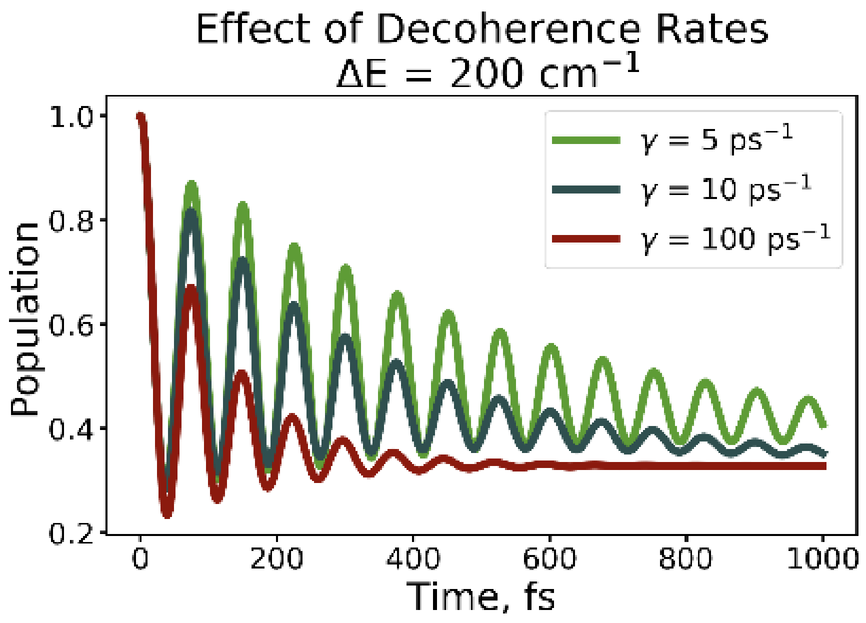


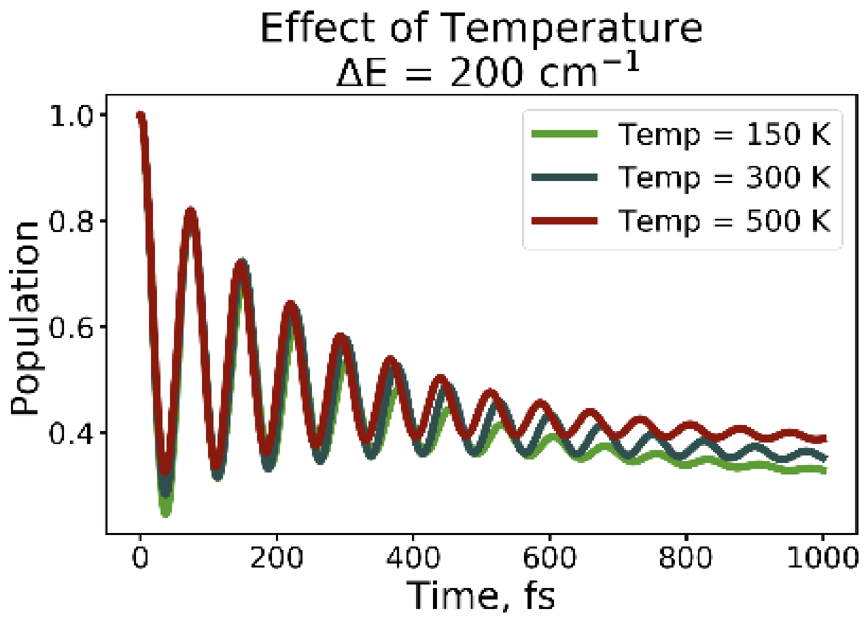
Effect of Temperature
ΔE = 100 cm⁻¹
Temp = 150 K
Temp = 300 K
Temp = 500 K
Population
Time, fs

(a) (b) (c)



Effect of Reorganization Energy
ΔE = 200 cm⁻¹
η = 50 cm⁻¹
η = 100 cm⁻¹
η = 250 cm⁻¹
Population
Time, fs





(d) (e) (f)

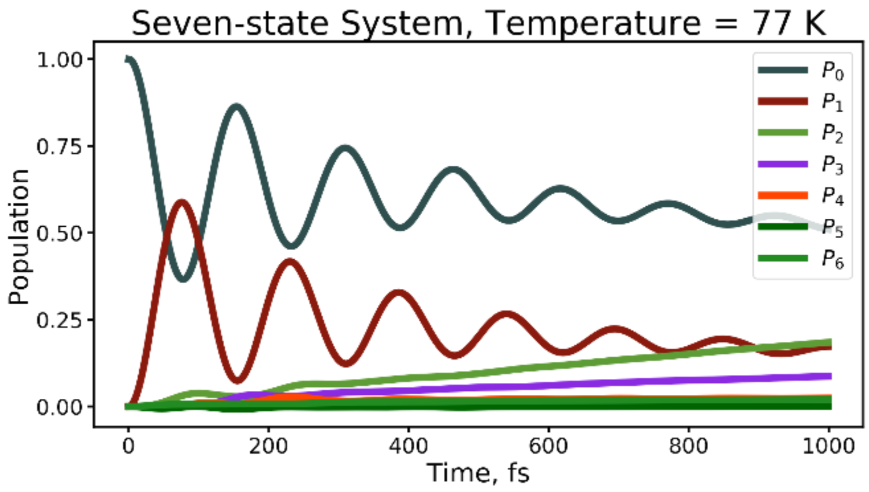
Figure 6. Qualitative trends in population dynamics of an open 2-level system as computed via HEOM. The variation of the dynamics with respect to: a, d) bath reorganization energy; b, e) decoherence rates; and c, f) temperature. All systems converged w.r.t Matsubara frequency and hierarchy level. The upper row corresponds to$\Delta E = 100\ cm^{-1}$ and the lower row corresponds to$\Delta E = 200\ cm^{-1}$.

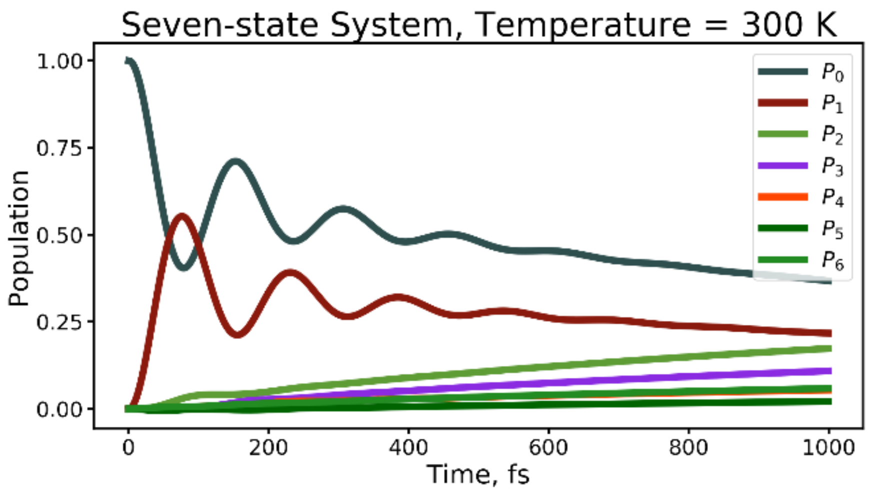
As another illustration, we explore the qualitative dependence of population dynamics in a molecular dimer system on the choice of system and bath parameters. The molecular dimer Hamiltonian is given by:

$H = \frac{1}{2}\Delta E * \sigma_z + V * \sigma_x$, (17)

where ΔE is the energy gap, $V = 200\ cm^{-1}$, is the electronic coupling between the two states, and σ_x andσ_z are the Pauli matrices. For this example, we start with the base model given by Strumpfer and Schulten[6] and vary the reorganization energy, η (Figure 6a, 6d), decoherence rates, γ (Figure 6b, 6e), and temperature (Figure 6c, 6f). For these examples, the other parameters are as follows: time step = 0.1 fs, number of steps = 10000, γ = 10 ps^{-1}, temperature = 300 K, and η = 100 cm^{-1}, unless otherwise specified. We observe that increasing η, γ, and/or temperature increases the rate of thermalization. These trends can be rationalized as follows. Reorganization energy quantifies how easy it is to perturb a system away from its equilibrium. A bath with a larger reorganization energy changes would tend to counteract the displacement away from equilibrium to a larger degree, and consequently would force faster thermalization. Small decoherence rates lead to a longer preservation of quantum coherences, which is in turn promotes persistent population transfer between involved states, which counteracts the idea of reaching thermal equilibrium. Note, this observation is also consistent with the behavior of decoherence-corrected surface hopping approaches.[41] Finally, a higher temperature promotes more frequent "collisions" of the quantum system with the bath, which increases energy transfer rates, and in turn allows the system to reach thermal equilibrium more quickly.

In addition, we vary the energy gap magnitude, ΔE to be either100 cm^{-1} or 200 cm^{-1}, for each of the bath parameters set. As expected, the equilibrium populations depend on the chosenΔE. However, we also observe two notable trends. First, the rate of thermalization decreases with the increase of ΔE, which is consistent with the slowing down of the population transfer for larger energy gaps, expected from the simple Rabi oscillation consideration. This observation is also consistent with the recently reported surface hopping calculations on quantum systems embedded in effective baths.[41] Second, we observe that the rates of thermalization become much more sensitive to the bath reorganization energy and decoherence rates for larger energy gap system (Figure 6, panels d-f).





(a) (b)

Figure 7. Evolution of the state populations of a seven-state system, modeling one unit of the FMO pigment–protein complex. Simulations are performed at two temperatures: a) 77 K, chosen to model a cryogenic temperature and limit decoherence, and b) 300 K, chosen to model physiological temperature.

We also illustrate the use of our HEOM implementation in modeling of exciton transfer in the FMO system, a pigment–protein complex found in green sulfur bacteria.[15] The complex consists of several identical units that each contain seven bacteriochlorophyll molecules. Each of these molecules can be treated as an individual state (site). The cite energies and couplings (in cm^{-1}) are given by a 7 x 7 Hamiltonian:[15]

$$H =$$

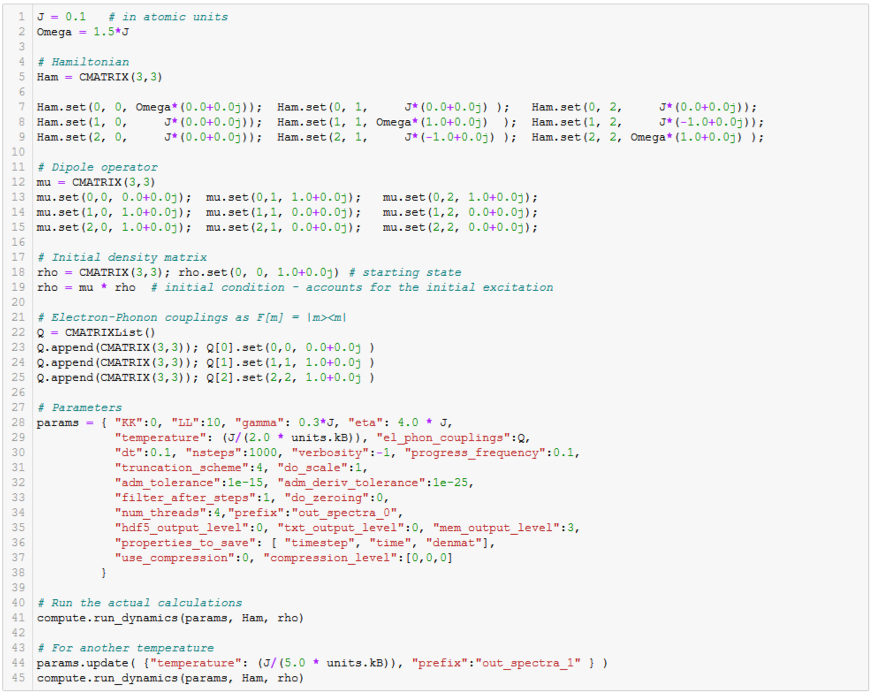
$$\begin{pmatrix} 410 & amp; -87.7 & amp; 5.5 & amp; -5.9 & amp; 6.7 & amp; -13.7 & amp; -9.9 \\ & amp; 530 & amp; 30.8 & amp; 8.2 & amp; 0.7 & amp; 11..8 & amp; 4.3 \\ & amp; & amp; 210 & amp; -53.2 & amp; -2.2 & amp; -9.6 & amp; 6.0 \\ & amp; & amp; & amp; 320 & amp; -70.7 & amp; -17.0 & amp; -63.3 \\ & amp; & amp; & amp; & amp; 480 & amp; 81.1 & amp; -1.3 \\ & amp; & amp; & amp; & amp; & amp; 630 & amp; 39.7 \\ & amp; & amp; & amp; & amp; & amp; & amp; 440 \end{pmatrix} \quad (18)$$

The calculations are conducted with the parameters: KK = 1 (two Matsubara frequencies), LL = 5 (maximal hierarchy level), $\gamma = 0.02$ fs^{-1} (decoherence rate), and $\eta = 35$ cm^{-1} (reorganization energy). The 1 ps trajectory with the integration timestep of 1 fs requires an order of thirteen minutes running with 1 thread, when run on Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz. The resultant population dynamics (Figure 7) for both the cryogenic temperature (panel a) and physiological temperature (panel b) matches those reported earlier by Ishizaki and Fleming.[15]

3.2.2. Calculation of the absorption line shapes.

An example snippet to run the absorption line shape calculations is shown in Figure 8. It is structured similarly to the one for the density matrix evolution (Figure 4), but has a number of exceptions. First of all, the Hamiltonian is defined differently – the ground electronic state is not coupled to any of the excited states, but the two states are coupled to each other via $H_{12} = H_{21} = -J$. Such a Hamiltonian corresponds to the excitonic dimer, as for instance defined by Shi et al.[25]

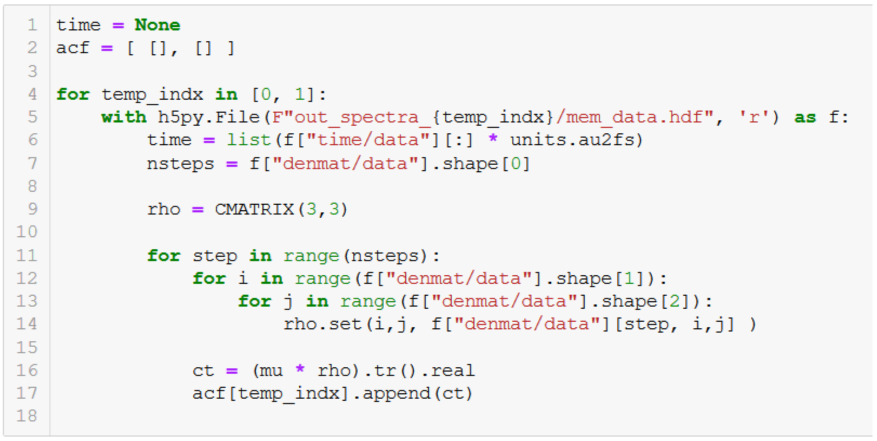
In addition, we define the transition dipole moment operator, which couples the ground and excited states and is given by$\mu = \sum_{n=1}^{2} (\ |n\rangle \langle 0| \ + \ |0\rangle \langle n| \ )$. The matrix representation of this operator is given in lines 12-15 of the snippet in Figure 8. The initial conditions in the absorption line shapes calculations are also different from those in the bare population dynamics. In this case, although the system starts in its electronic ground state,$\rho_{\text{gs}} = \ |0\rangle \langle 0|$, the transition dipole moment operator evolves this initial density matrix to a different state, in which coherences between the ground and each of the excited states are initialized,$\rho_{\mathbf{0}}\left(t\right) = \mu\rho_{\text{gs}}$. This setup is done in lines 18-19 of the snippet.



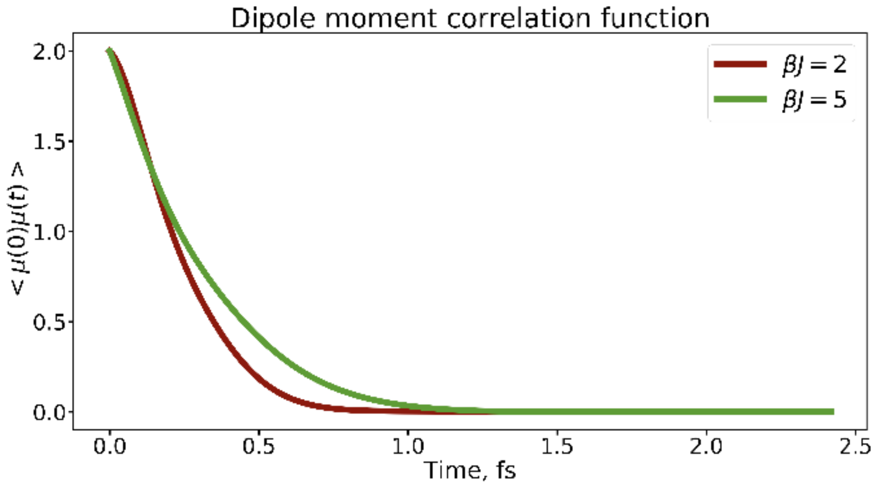
```
 1 J = 0.1   # in atomic units
 2 Omega = 1.5*J
 3 
 4 # Hamiltonian
 5 Ham = CMATRIX(3,3)
 6 
 7 Ham.set(0, 0, Omega*(0.0+0.0j));  Ham.set(0, 1,     J*(0.0+0.0j) );  Ham.set(0, 2,     J*(0.0+0.0j));
 8 Ham.set(1, 0,     J*(0.0+0.0j));  Ham.set(1, 1, Omega*(1.0+0.0j)  );  Ham.set(1, 2,     J*(-1.0+0.0j));
 9 Ham.set(2, 0,     J*(0.0+0.0j));  Ham.set(2, 1,     J*(-1.0+0.0j) );  Ham.set(2, 2, Omega*(1.0+0.0j) );
10 
11 # Dipole operator
12 mu = CMATRIX(3,3)
13 mu.set(0,0, 0.0+0.0j);  mu.set(0,1, 1.0+0.0j);   mu.set(0,2, 1.0+0.0j);
14 mu.set(1,0, 1.0+0.0j);  mu.set(1,1, 0.0+0.0j);   mu.set(1,2, 0.0+0.0j);
15 mu.set(2,0, 1.0+0.0j);  mu.set(2,1, 0.0+0.0j);   mu.set(2,2, 0.0+0.0j);
16 
17 # Initial density matrix
18 rho = CMATRIX(3,3); rho.set(0, 0, 1.0+0.0j) # starting state
19 rho = mu * rho  # initial condition - accounts for the initial excitation
20 
21 # Electron-Phonon couplings as F[m] = |m><m|
22 Q = CMATRIXList()
23 Q.append(CMATRIX(3,3)); Q[0].set(0,0, 0.0+0.0j )
24 Q.append(CMATRIX(3,3)); Q[1].set(1,1, 1.0+0.0j )
25 Q.append(CMATRIX(3,3)); Q[2].set(2,2, 1.0+0.0j )
26 
27 # Parameters
28 params = { "KK":0, "LL":10, "gamma": 0.3*J, "eta": 4.0 * J,
29            "temperature": (J/(2.0 * units.kB)), "el_phon_couplings":Q,
30            "dt":0.1, "nsteps":1000, "verbosity":-1, "progress_frequency":0.1,
31            "truncation_scheme":4, "do_scale":1,
32            "adm_tolerance":1e-15, "adm_deriv_tolerance":1e-25,
33            "filter_after_steps":1, "do_zeroing":0,
34            "num_threads":4,"prefix":"out_spectra_0",
35            "hdf5_output_level":0, "txt_output_level":0, "mem_output_level":3,
36            "properties_to_save": [ "timestep", "time", "denmat"],
37            "use_compression":0, "compression_level":[0,0,0]
38          }
39 
40 # Run the actual calculations
41 compute.run_dynamics(params, Ham, rho)
42 
43 # For another temperature
44 params.update( {"temperature": (J/(5.0 * units.kB)), "prefix":"out_spectra_1" } )
45 compute.run_dynamics(params, Ham, rho)
```

Figure 8. An example snippet to run the HEOM-based absorption line shapes calculations within a Jupyter notebook.

First, the HEOM dynamics is computed to propagate the reduced density matrix of the system using the familiar function "compute.run_dynamics". In this example, we execute the calculations twice – for two different temperatures. Note how the second call of the function takes the parameters dictionary with only two relevant parameters changed – "temperature" and "prefix". The latter would ensure that we store the results of these two calculations in separate directories. Once the RDM evolution is computed and the results are stored in the output HDF5 file, the data can be read and used to compute the transition dipole moment ACF, Eq. 15. This is illustrated in the snippet in Figure 9a. In particular, note how Eq. 15 is implemented in a single line, thanks to object-oriented design and suitable operator overloads of the CMATRIX data class within the core of Libra package. The ACFs for two considered temperatures are illustrated in Figure 9b.



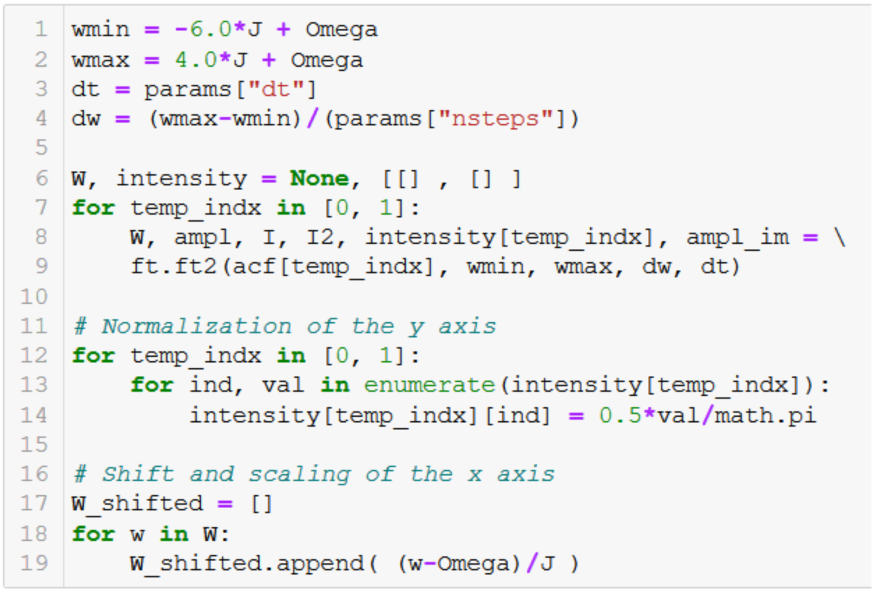
```python
time = None
acf = [ [], [] ]

for temp_indx in [0, 1]:
    with h5py.File(F"out_spectra_{temp_indx}/mem_data.hdf", 'r') as f:
        time = list(f["time/data"][:] * units.au2fs)
        nsteps = f["denmat/data"].shape[0]

        rho = CMATRIX(3,3)

        for step in range(nsteps):
            for i in range(f["denmat/data"].shape[1]):
                for j in range(f["denmat/data"].shape[2]):
                    rho.set(i,j, f["denmat/data"][step, i,j] )

            ct = (mu * rho).tr().real
            acf[temp_indx].append(ct)

```



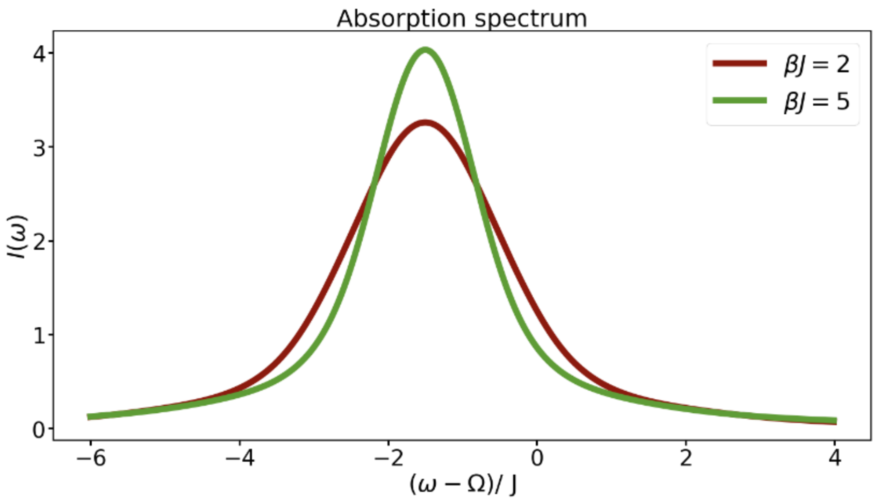
(a) (b)

Figure 9. *Computing the transition dipole moment ACF: a) a code snippet detailing how to read the propagated RDM and calculate the transition dipole moment ACF; b) the computed ACFs.*

Finally, the computed ACFs can be Fourier-transformed (FT) to yield the spectra according to Eq. 14. The FT of this type is implemented in the "libra_py.ft" module. The underlying "ft2" function takes the initial time-series of data to be computed, the range of the frequencies one is interested in, as well as the corresponding resolution, which is related to the number of time-steps (the number of data points in the time-series). The function yields a number of properties, including the sine and cosine FTs of the data ("ampl_im" and "ampl_re", in line 8, Figure 10a, "ampl_re" is assigned to variable "intensity[]"), the absolute value of the complex FT amplitude (variable "I") and the square of its magnitude (variable "I2"). The constructed frequency-domain grid (variable "W") is also reported, primarily for plotting purposes. The computed absorption line shapes (in the shifted and scaled axes) for two considered temperatures are illustrated in Figure 10b.



```
wmin = -6.0*J + Omega
wmax = 4.0*J + Omega
dt = params["dt"]
dw = (wmax-wmin)/(params["nsteps"])

W, intensity = None, [[] , [] ]
for temp_indx in [0, 1]:
    W, ampl, I, I2, intensity[temp_indx], ampl_im = \
    ft.ft2(acf[temp_indx], wmin, wmax, dw, dt)

# Normalization of the y axis
for temp_indx in [0, 1]:
    for ind, val in enumerate(intensity[temp_indx]):
        intensity[temp_indx][ind] = 0.5*val/math.pi

# Shift and scaling of the x axis
W_shifted = []
for w in W:
    W_shifted.append( (w-Omega)/J )
```



(a) (b)

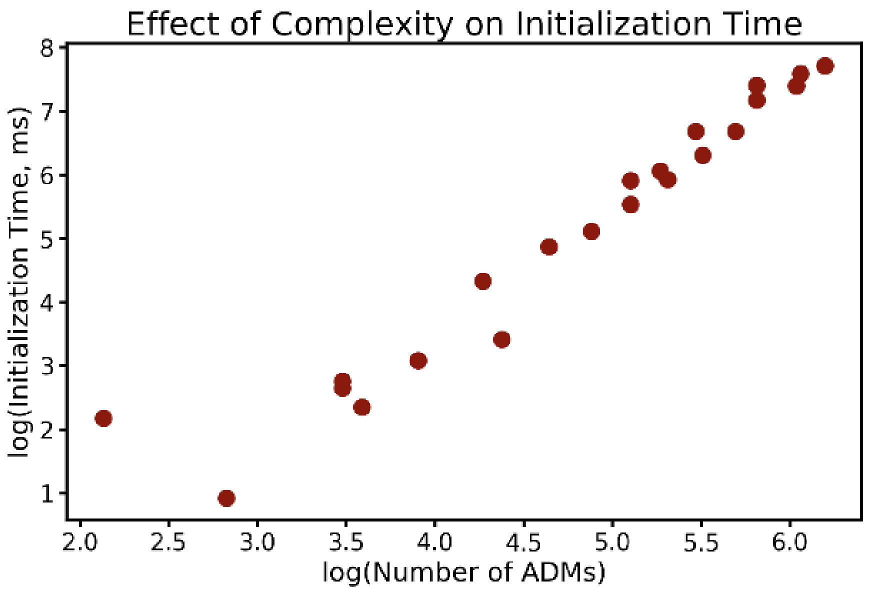
Figure 10. *Computing the absorption spectra: a) a code snippet illustrating a Fourier transform of the ACF computed above; b) the computed absorption line shapes for the two temperatures considered.*

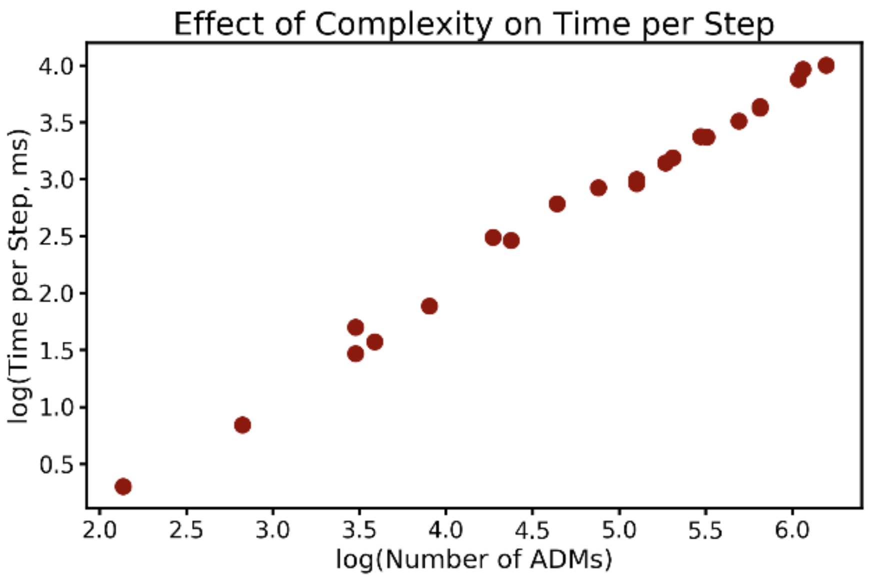
3.2.3. Some complexity considerations

In this section, we discuss the computational requirements for running HEOM calculations and we assess the runtimes expected for problems of various sizes. The main parameters that affect the complexity and runtime of HEOM simulations are the maximal hierarchy level (L, defined by the parameter "LL"), the number of Matsubara frequencies (K, defined by the parameter "KK"), and the number of quantum states (M, defined by the dimensionality of the input Hamiltonian matrix). The latter two define the length of the multi-dimensional index vectors enumerating ADMs, $d = M * (K + 1)$. The total number of distinct ADMs is given by:

$N_{\text{ADM}} = \frac{(L+d)!}{L!d!} = \prod_{i=1}^{d} \frac{L+i}{i}$. (19)

This number is returned by the "compute_nn_tot" function of our module. The factorial growth of the number of ADMs w.r.t. any of the M, K or L parameters is the main reason for the method's complexity. Considering that the convergence w.r.t. to both K and L numbers should be achieved, the number of ADMs can be quite large. Here, we considered a number of calculations with the systematically-varying "KK" and "LL" parameters for a 2-level system, Eq. 17. The calculations are executed on Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz processors using only one thread. We separately measure the time to initialize the calculations (Figure 11a) and integrate HEOM (Figure 11b).





(a) (b)

__Figure 11.__ The benchmarking of the run times for a) initialization of HEOM hierarchy; b) dynamical calculations vs. the number of ADMs.

It is interesting to see that the initialization of the internal variables and the construction of the hierarchy could be the quite more demanding than the actual integration, especially as the problem's complexity to grow to the order of 10^6 ADMs. This is likely do to fact that it is much more demanding to allocate large chunks of memory (to accommodate all ADMs) than operate on the already allocated chunks. The absolute values of the runtimes are also reasonable – for the problem with 10^6 ADMs, it takes only about 10 seconds to advance 1 step forward. In many situations, the solutions converge much earlier than one hits this level of complexity, so most of the time, the calculations are very fast. Although our present implementation could be further optimized, especially for large problems, it may already be a viable tool for many projects.

4. Conclusions

The HEOM approach is now available in the modular open-source Libra software, starting from the version v4.8.1. This version is available at the Zenodo server[42] as well as from the Quantum Dynamics Hub GitHub Libra software repository https://github.com/Quantum-Dynamics-Hub/libra-code. The current work provides a comprehensive account on the underlying theoretical foundations and terminology of the method, the key algorithms used, and the important implementation details and use guidance. The detailed examples to run the calculations presented in this work are available at the Zenodo server[43] as well as from the GitHub data repository https://github.com/AkimovLab/Project_HEOM.

The present implementation features a user-friendly design of the Python-level modules for HEOM calculation, which shall facilitate the use of this method in Jupyter- or Python-based calculations. Our current implementation considers a number of acceleration features, such as using the scaled HEOM method, filtering the HEOM, using the concept of active equation lists, which are updated periodically to remove the propagation of ADMs when the ADM's time derivatives are negligible. A nearly trivial OpenMP parallelization is incorporated in the step of computing ADM's time-derivatives and leads to expected acceleration of the computations. We have provided examples of using the code for propagating reduced density matrices to describe quantum dynamics of open systems. We have illustrated the capabilities of the present HEOM implementation in computing spectral line shapes. We have provided a number of examples on the expected qualitative changes in the dynamics of quantum systems in response to variation of various properties of the bath. We have provided some computational scalability and execution time benchmarks to guide the potential users of this software in feasibility of various types of calculations.

Acknowledgements

Computational support is provided by the Center for Computational Research at the University at Buffalo. AJ and AVA acknowledge useful discussions with Prof. Joseph Subotnik that stimulated this work. AJ also acknowledges the computational clusters of Subotnik group, and department of chemistry IITB on which part of the codes were written and benchmarked.

Funding Information

AVA acknowledges the financial support of the National Science Foundation (Grant OAC-NSF-1931366).

References

1. Y. Tanimura and R. Kubo, *J. Phys. Soc. Jpn.* , **1989** , 58, 101–114.

2. A. Ishizaki and Y. Tanimura, *J Phys Soc Jpn* , **2005** , 74, 3131–3134.

3. Y. Tanimura and R. Kubo, *J Phys Soc Jpn* , **1989** , 58, 1199–1206.

4. Y. Tanimura, *J. Phys. Soc. Jpn.* , **2006** , 75, 082001.

5. J. Strümpfer and K. Schulten, *J Chem Phys* , **2009** , 131, 225101.

6. J. Strümpfer and K. Schulten, *J Chem Theory Comput* ,**2012** , 8, 2808–2816.

7. A. Jain, A. S. Petit, J. M. Anna and J. E. Subotnik, *J. Phys. Chem. B* , **2019** , 123, 1602–1617.

8. Q. Shi, L. Chen, G. Nan, R. Xu and Y. Yan, *J. Chem. Phys.* ,**2009** , 130, 164518.

9. C. Kreisbeck and T. Kramer, *J. Phys. Chem. Lett.* ,**2012** , 3, 2828–2833.

10. D. E. Chandler, J. Strümpfer, M. Sener, S. Scheuring and K. Schulten, *Biophys. J.* , **2014** , 106, 2503–2510.

11. J. Strümpfer, M. Şener and K. Schulten, *J. Phys. Chem. Lett.* ,**2012** , 3, 536–542.

12. M. Yang and G. R. Fleming, *Chem. Phys.* , **2002** , 18.

13. C. Kreisbeck, T. Kramer, M. Rodríguez and B. Hein, *J. Chem. Theory Comput.* , **2011** , 7, 2166–2174.

14. T. Forster, *Ann. Phys.* , **1948** , 437, 55–75.

15. A. Ishizaki and G. R. Fleming, *Proc. Natl. Acad. Sci.* ,**2009** , 106, 17255–17260.

16. J. Zhu, S. Kais, P. Rebentrost and A. Aspuru-Guzik, *J. Phys. Chem. B* , **2011** , 115, 1531–1537.

17. J. Wu, F. Liu, J. Ma, R. J. Silbey and J. Cao, *J. Chem. Phys.* , **2012** , 137, 174111.

18. D. M. Wilkins and N. S. Dattani, *J. Chem. Theory Comput.* ,**2015** , 11, 3411–3419.

19. S.-H. Yeh and S. Kais, *J. Chem. Phys.* , **2014** , 141, 12B645_1.

20. X. Leng, Y.-M. Yan, R.-D. Zhu, K. Song, Y.-X. Weng and Q. Shi,*J. Phys. Chem. B* , **2018** , 122, 4642–4652.

21. Y. Jing, L. Chen, S. Bai and Q. Shi, *J. Chem. Phys.* ,**2013** , 138, 01B615.

22. S. Bai, K. Song and Q. Shi, *J. Phys. Chem. Lett.* ,**2015** , 6, 1954–1960.

23. A. Jain and J. E. Subotnik, *JCP* , **2015** , 143, 134107.

24. S. J. Cotton and W. H. Miller, *J. Chem. Theory Comput.* ,**2016** , 12, 983–991.

25. Q. Shi, L. Chen, G. Nan, R.-X. Xu and Y. Yan, *J. Chem. Phys.* ,**2009** , 130, 084105.

26. *Tanimura HEOM codes* , .

27. M. Tsuchimoto and Y. Tanimura, *J. Chem. Theory Comput.* ,**2015** , 11, 3859–3865.

28. C. Kreisbeck and T. Kramer, *Exciton Dynamics Lab for Light-Harvesting Complexes (GPU-HEOM)* , https://nanohub.org/resources/gpuheompop, **2017** .

29. J. R. Johansson, P. D. Nation and F. Nori, *Comput. Phys. Commun.* , **2013** , 184, 1234.

30. A. V. Akimov, *J. Comput. Chem.* , **2016** , 37, 1626–1649.

31. L. Chen, R. Zheng, Q. Shi and Y. Yan, *J. Chem. Phys.* ,**2009** , 131, 094502.

32. J. Strümpfer and K. Schulten, *J. Chem. Theory Comput.* ,**2012** , 8, 2808–2816.

33. M. Tanaka and Y. Tanimura, *J Chem Phys* , **2010** , 132.

34. L. Zhu, H. Liu, W. Xie and Q. Shi, *J Chem Phys* ,**2012** , 137.

35. Y. Tanimura, *J. Chem. Phys.* , **2012** , 137, 22A550.

36. H. Liu, L. Zhu, S. Bai and Q. Shi, *J. Chem. Phys.* ,**2014** , 140, 134106.

37. D. Abrahams and R. W. Grosse-Kunstleve, *CC Users J.* ,**2003** .

38. *HDF5 for Python* , https://www.h5py.org/.

39. J. D. Hunter, *Comput. Sci. Eng.* , **2007** , 9, 90–95.

40. Thomas A Caswell, Michael Droettboom, Antony Lee, John Hunter, Eric Firing, David Stansby, Jody Klymak, Tim Hoffmann, Elliott Sales de Andrade, Nelle Varoquaux, Jens Hedegaard Nielsen, Benjamin Root, Phil Elson, Ryan May, Darren Dale, Jae-Joon Lee, Jouni K. Seppänen, Damon McDougall, Andrew Straw, Paul Hobson, Christoph Gohlke, Tony S Yu, Eric Ma, Adrien F. Vincent, Steven Silvester, Charlie Moad, Nikita Kniazev, Paul Ivanov, Elan Ernest and Jan Katins, *matplotlib/matplotlib: REL: v3.2.1* , Zenodo, **2020** .

41. B. Smith A. and A. V. Akimov, *J Chem Phys* , **2019** , 151, 124107.

42. alexvakimov, Brendan Smith, Kosuke Sato, Wei Li, Xiang Sun and Matthew Chan, *Quantum-Dynamics-Hub/libra-code: HEOM in Libra* , Zenodo, **2020** , https://zenodo.org/record/3753197#.Xpdpepl7lPY

43. Story Temen, *AkimovLab/Project_HEOM: Data for the HEOM module demonstration* , Zenodo, **2020** , https://zenodo.org/record/3753507#.Xpdu8pl7lPY