Tutorial on the Hierarchical Equations of Motion (HEOM)

Xiaohan Dan and Victor S. Batista
Department of Chemistry, Yale University, New Haven, CT 06511, U.S.A.

Abstract

The hierarchical equations of motion (HEOM) provide a nonperturbative route to the reduced quantum dynamics of a system linearly coupled to a Gaussian harmonic environment. This tutorial is organized around one central idea: once the bath correlation function is represented as a sum of exponentials, each memory convolution can be promoted to a local first-order dynamical variable. We first expose this mechanism for a single exponential memory mode before introducing the full multi-index hierarchy. The derivation then shows explicitly how differentiation generates coherent propagation, memory damping, upward coupling, and downward feedback between auxiliary density operators (ADOs). Forward- and backward-path factors are mapped carefully to left and right operator multiplication so that the origin of every commutator and sign remains visible.

The same environmental memory is then viewed in the complementary QUAPI/TEMPO representation, where recent forward–backward histories are retained explicitly and compressed as a persistent matrix-product state. The accompanying notebook implements a two-stage convergence protocol for an Ohmic spin–boson qubit: the QUAPI memory length is first converged at fixed time step, then the Trotter step is refined at fixed physical memory time, and only afterward is the resulting trajectory compared with HEOM built from the same numerically evaluated bath correlation function. For the reported demonstration this selects $\tau_{\mathrm{mem}} = 4$, $(dt, K) = (0.05, 80)$; the remaining QUAPI–HEOM difference is interpreted relative to the independently measured memory, time-step, MPS-compression, correlation-fit, and HEOM-tier convergence scales. A compact code-to-equation appendix and the complete executable Python source are included. Unless stated otherwise, $\hbar = k_{\mathrm{B}} = 1$.

Contents

1 Overview and scope

HEOM was introduced by Tanimura and Kubo as a nonperturbative formulation of dissipative quantum dynamics and has since become a widely used framework for condensed-phase quantum dynamics and spectroscopy [1–4]. The purpose of this tutorial is not to survey every HEOM variant, but to derive the standard bosonic hierarchy for a single Hermitian system coupling operator interacting linearly with a Gaussian harmonic bath.

The logic is

$$\boxed{\text{harmonic bath}} \longrightarrow \boxed{\text{influence functional}} \longrightarrow \boxed{C(t) = \sum_k c_k e^{-\gamma_k t}} \longrightarrow \boxed{\text{memory variables}} \longrightarrow \boxed{\text{ADOs}} \longrightarrow \boxed{\text{local hierarchy}}. \tag{1.1}$$

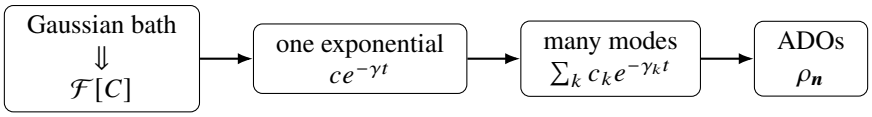
The physical reduced density operator is the zeroth member of this hierarchy. Higher ADOs are not additional physical states; they are auxiliary operators that store the information needed to represent environmental memory locally in time.

Central idea

The original reduced dynamics is nonlocal in time because the environment remembers the past. If that memory kernel is decomposed into exponentials, each exponential memory component obeys a first-order differential equation. HEOM replaces one integro-differential equation with memory by a larger set of coupled ordinary differential equations without explicit history dependence.

1.1 How to read the derivation

The HEOM derivation is easiest to follow if each section is viewed as justifying one arrow in equation (1.1). The bath trace first removes the microscopic oscillator coordinates and leaves only $C(t)$. A single-exponential warm-up then shows, without multi-index notation, why an exponential convolution becomes a local ordinary differential equation. Only after that simple case is clear do we introduce many exponential modes and the ADO hierarchy. Part V then returns to the same influence functional and introduces the complementary QUAPI/TEMPO time-history coordinates. The numerical comparison is performed only after the approximations internal to each representation have been varied separately.



Reader checkpoint

At every stage ask two questions: *what information about the bath is being stored?* and *what numerical approximation, if any, has been introduced?* These questions organize both the analytical derivation and the convergence study.

Exact transformations versus numerical approximations

A useful way to read the tutorial is to distinguish changes of representation from approximations. Tracing out a Gaussian harmonic bath and rewriting its influence in terms of $C(t)$ are exact for the model specified below. By contrast, replacing $C(t)$ by a finite exponential sum, truncating the HEOM hierarchy, imposing a finite QUAPI memory time, and compressing a TEMPO MPS are numerical approximations that must be converged independently. We will mark these transitions explicitly as they are introduced.

Part I

Foundations: eliminating the bath

Part I orientation

Input: a system Hamiltonian, a harmonic bath, and a linear system–bath coupling.
Goal: eliminate the microscopic bath coordinates exactly and identify the object that stores their dynamical influence.
Approximation introduced here: none beyond the stated Gaussian harmonic-bath model and the chosen initial-state convention.

2 Model Hamiltonian and conventions

We write the total Hamiltonian as

$$\hat{H}_{\mathrm{T}} = \hat{H}_{\mathrm{S}} + \hat{H}_{\mathrm{B}} + \hat{H}_{\mathrm{I}}, \tag{2.1}$$

with harmonic bath

$$\hat{H}_{\mathrm{B}} = \sum_j \left(\frac{\hat{p}_j^2}{2m_j} + \frac{1}{2} m_j \omega_j^2 \hat{x}_j^2 \right) \tag{2.2}$$

and bilinear system–bath coupling

$$\hat{H}_{\mathrm{I}} = \hat{V} \otimes \hat{B}, \qquad \hat{B} = \sum_j c_j \hat{x}_j. \tag{2.3}$$

For a one-dimensional system coordinate one may take $\hat{V} = f(\hat{q})$.

Table 1: Notation used throughout the tutorial.

Symbol	Meaning
$\hat{\rho}_{\mathrm{S}}(t)$	physical reduced density operator
$\hat{V}$	system coupling operator
$\hat{B}$	collective bath operator
$C(t)$	equilibrium bath correlation function
c_k, γ_k	amplitude and decay rate of exponential memory mode k
$\mathcal{B}_k(t)$	path-dependent memory functional for mode k
$\hat{\rho}_{\boldsymbol{n}}(t)$	ADO labeled by multi-index $\boldsymbol{n}$
$L(\boldsymbol{n}) = \sum_k n_k$	hierarchy tier

A Caldeira–Leggett Hamiltonian is sometimes written with the counterterm

$$\hat{H}_{\mathrm{ct}} = \hat{V}^2 \sum_j \frac{c_j^2}{2m_j\omega_j^2}. \tag{2.4}$$

This term removes the static shift of the bare system potential generated by completing the square in the bath coordinates. The HEOM structure is unchanged; one simply adopts a consistent renormalization convention and includes the counterterm in $\hat{H}_{\mathrm{S}}$ when appropriate.

We assume initially

$$\hat{\rho}_{\mathrm{T}}(0) = \hat{\rho}_{\mathrm{S}}(0) \otimes \rho_{\mathrm{B}}^{\mathrm{eq}}, \qquad \rho_{\mathrm{B}}^{\mathrm{eq}} = \frac{\mathrm{e}^{-\beta\hat{H}_{\mathrm{B}}}}{Z_{\mathrm{B}}}, \qquad Z_{\mathrm{B}} = \mathrm{Tr}_{\mathrm{B}}\, \mathrm{e}^{-\beta\hat{H}_{\mathrm{B}}}. \tag{2.5}$$

For correlated initial states, the hierarchy can still be used, but the initial ADOs generally require additional preparation.

3 Reduced dynamics as a double path integral

The total density operator evolves according to

$$\frac{d}{dt}\hat{\rho}_{\mathrm{T}}(t) = -\mathrm{i}\big[\hat{H}_{\mathrm{T}},\, \hat{\rho}_{\mathrm{T}}(t)\big]. \tag{3.1}$$

whereas the reduced state is

$$\hat{\rho}_{\mathrm{S}}(t) = \mathrm{Tr}_{\mathrm{B}}\, \hat{\rho}_{\mathrm{T}}(t). \tag{3.2}$$

After tracing out the bath, the evolution is generally non-Markovian.

In the system coordinate representation, the reduced density matrix can be written as a double path integral [5]. Readers who would like to derive this representation from the short-time propagator before proceeding may read Appendix A first; the main derivation below uses the path-integral form but does not otherwise depend on that appendix:

$$\begin{aligned}\rho(q_f^+, q_f^-, t) = \int \mathrm{d}q_0^+\, \mathrm{d}q_0^-\, \rho(q_0^+, q_0^-, 0) \int_{q^+(0)=q_0^+}^{q^+(t)=q_f^+} \mathcal{D}q^+ \int_{q^-(0)=q_0^-}^{q^-(t)=q_f^-} \mathcal{D}q^- \\ \times \exp\big\{\mathrm{i}S_{\mathrm{S}}[q^+] - \mathrm{i}S_{\mathrm{S}}[q^-]\big\}\, \mathcal{F}[q^+, q^-].\end{aligned} \tag{3.3}$$

The + path is the ket (forward) branch and the − path is the bra (backward) branch. For the real Hamiltonians considered here, the complex-conjugate propagator is represented by the minus sign multiplying the backward action.

The environment now appears only through the influence functional $\mathcal{F}[q^+, q^-]$.

4 Exact elimination of a Gaussian harmonic bath

Define the system coupling evaluated along the two branches,

$$V^+(\tau) = f(q^+(\tau)), \qquad V^-(\tau) = f(q^-(\tau)), \tag{4.1}$$

and the equilibrium bath correlation function

$$C(t) = \mathrm{Tr}_\mathrm{B}\left[\rho_\mathrm{B}^\mathrm{eq}\,\hat{B}(t)\hat{B}(0)\right], \qquad \hat{B}(t) = \mathrm{e}^{\mathrm{i}\hat{H}_\mathrm{B}t}\,\hat{B}\,\mathrm{e}^{-\mathrm{i}\hat{H}_\mathrm{B}t}. \tag{4.2}$$

Because the bath is harmonic and the coupling is linear in the bath coordinates, all bath path integrals are Gaussian. They can therefore be evaluated exactly. The result is the Feynman–Vernon influence functional [6]

$$\boxed{\mathcal{F}[V^+, V^-] = \exp\left[-\int_0^t \mathrm{d}\tau\,[V^+(\tau) - V^-(\tau)]\int_0^\tau \mathrm{d}s\,\left(C(\tau - s)V^+(s) - C^*(\tau - s)V^-(s)\right)\right].} \tag{4.3}$$

At this point the bath coordinates have disappeared completely. Their entire dynamical effect is encoded in the two-time kernel $C(\tau - s)$.

It is often useful to separate

$$C(t) = C_R(t) + \mathrm{i}C_I(t), \qquad \Delta V = V^+ - V^-, \qquad \Sigma V = V^+ + V^-. \tag{4.4}$$

Then

$$C(\tau - s)V^+(s) - C^*(\tau - s)V^-(s) = C_R(\tau - s)\Delta V(s) + \mathrm{i}C_I(\tau - s)\Sigma V(s). \tag{4.5}$$

The real and imaginary parts of the bath correlation function therefore enter the influence phase in distinct combinations.

4.1 Forward and backward endpoints: the operator dictionary

The two paths are not two independent physical systems. They are the ket and bra sides of the same density matrix. At the final time, a forward-branch factor multiplies the density operator from the left, whereas a backward-branch factor multiplies it from the right:

path factor	operator action on $\hat{X}$	branch
$V^+(t)$	$\hat{V}\hat{X}$	ket / forward
$V^-(t)$	$\hat{X}\hat{V}$	bra / backward

(4.6)

Consequently, $V^+ - V^-$ becomes $[\hat{V}, \cdot]$. This simple dictionary is the origin of the commutator appearing in the upward HEOM coupling and is worth keeping visible throughout the derivation.

What has been achieved?

The oscillator coordinates have disappeared. The environment now enters only through the stationary two-time correlation function $C(t)$. No Markov approximation and no perturbative expansion in the system–bath coupling has been made.

5 Bath correlation function and spectral density

For the discrete harmonic bath in equations (2.2) and (2.3),

$$C(t) = \sum_j \frac{c_j^2}{2m_j\omega_j}\left[\coth\!\left(\frac{\beta\omega_j}{2}\right)\cos(\omega_j t) - \mathrm{i}\sin(\omega_j t)\right]. \tag{5.1}$$

To see this directly, write

$$\hat{x}_j = \frac{1}{\sqrt{2m_j\omega_j}}(\hat{a}_j + \hat{a}_j^\dagger), \tag{5.2}$$

with

$$\langle \hat{a}_j^\dagger \hat{a}_j\rangle = n_j, \qquad \langle \hat{a}_j \hat{a}_j^\dagger\rangle = n_j + 1, \qquad n_j = \frac{1}{\mathrm{e}^{\beta\omega_j} - 1}. \tag{5.3}$$

This gives

$$C(t) = \sum_j \frac{c_j^2}{2m_j\omega_j}\left[(n_j+1)\mathrm{e}^{-\mathrm{i}\omega_j t} + n_j \mathrm{e}^{+\mathrm{i}\omega_j t}\right], \tag{5.4}$$

which is equivalent to equation (5.1).

With the convention

$$J(\omega) = \frac{\pi}{2}\sum_j \frac{c_j^2}{m_j\omega_j}\delta(\omega-\omega_j), \qquad \omega > 0, \tag{5.5}$$

one obtains

$$C(t) = \frac{1}{\pi}\int_0^\infty \mathrm{d}\omega\, J(\omega)\left[\coth\!\left(\frac{\beta\omega}{2}\right)\cos\omega t - \mathrm{i}\sin\omega t\right]. \tag{5.6}$$

It is useful to separate

$$C(t) = C_\mathrm{R}(t) + \mathrm{i}C_\mathrm{I}(t), \tag{5.7}$$

with

$$C_\mathrm{R}(t) = \frac{1}{\pi}\int_0^\infty \mathrm{d}\omega\, J(\omega)\coth\!\left(\frac{\beta\omega}{2}\right)\cos(\omega t), \tag{5.8}$$

$$C_\mathrm{I}(t) = -\frac{1}{\pi}\int_0^\infty \mathrm{d}\omega\, J(\omega)\sin(\omega t). \tag{5.9}$$

The real part contains thermal fluctuations, whereas the imaginary part encodes the dissipative response of the bath.

Using

$$n_\mathrm{B}(\omega) = \frac{1}{\mathrm{e}^{\beta\omega}-1}, \qquad \coth\!\left(\frac{\beta\omega}{2}\right) = 2n_\mathrm{B}(\omega) + 1, \tag{5.10}$$

the same correlation function can be written as

$$C(t) = \frac{1}{\pi}\int_0^\infty \mathrm{d}\omega\, J(\omega)\left[(n_\mathrm{B}(\omega)+1)\mathrm{e}^{-\mathrm{i}\omega t} + n_\mathrm{B}(\omega)\mathrm{e}^{+\mathrm{i}\omega t}\right]. \tag{5.11}$$

This form makes the thermal origin of the two frequency components explicit. If $J(-\omega) = -J(\omega)$, then

$$C(t) = \frac{1}{\pi}\int_{-\infty}^\infty \mathrm{d}\omega\, J(\omega)\frac{\mathrm{e}^{-\mathrm{i}\omega t}}{1-\mathrm{e}^{-\beta\omega}}. \tag{5.12}$$

Part II

From bath memory to HEOM

Part II orientation

Input: the exact Feynman–Vernon influence functional expressed through $C(t)$.
Goal: replace explicit time-history integrals by a time-local enlarged state.
Key step: an exponential memory convolution satisfies a first-order ODE; products of those memory variables generate the ADO hierarchy.

6 From exponential bath memory to an implementable HEOM

The hierarchical equations of motion can be derived directly from the Feynman–Vernon influence functional. The central idea is simple:

> If the bath correlation function is written as a sum of exponentials, each exponential memory kernel can be replaced by a first-order auxiliary dynamical variable. The collection of all powers of these memory variables forms the HEOM hierarchy.

This section first derives this result for one exponential mode and then shows explicitly how the resulting equations are organized for numerical implementation.

Throughout this section we set $\hbar = 1$.

6.1 The memory term that must be localized

The exact influence functional has already been derived in equation (4.3). For the HEOM construction, the only structure we now need to isolate is its inner history-dependent convolution,

$$\mathcal{A}(\tau) = \int_0^\tau \mathrm{d}s \, \left[C(\tau - s) V^+(s) - C^*(\tau - s) V^-(s) \right]. \tag{6.1}$$

The value of $\mathcal{A}(\tau)$ depends on the entire previous trajectory $0 \le s \le \tau$. A direct propagation would therefore require explicit access to the past. HEOM becomes possible when $C(t)$ is represented by exponentials, because each resulting convolution can then be advanced recursively from its current value.

What changes in the next step?

Nothing about the influence-functional physics is approximated by introducing a memory variable. The approximation enters only later if the exponential representation of $C(t)$ is truncated.

6.2 Warm-up: one exponential memory mode

Suppose first that

$$C(t) = c\, \mathrm{e}^{-\gamma t}, \qquad C^*(t) = \bar{c}\, \mathrm{e}^{-\gamma t}, \qquad t \ge 0, \tag{6.2}$$

with

$$\operatorname{Re}\gamma > 0. \tag{6.3}$$

Define

$$\boxed{\mathcal{B}(t) = -\mathrm{i}\int_0^t \mathrm{d}s\, \mathrm{e}^{-\gamma(t-s)}\left[cV^+(s) - \bar{c}V^-(s)\right].} \tag{6.4}$$

This quantity is simply the memory convolution already present in the influence functional. No additional approximation has been introduced.

For clarity, define

$$A(t) = \int_0^t \mathrm{d}s\, \mathrm{e}^{-\gamma(t-s)}\left[cV^+(s) - \bar{c}V^-(s)\right]. \tag{6.5}$$

Then

$$\mathcal{B}(t) = -\mathrm{i}A(t), \qquad A(t) = \mathrm{i}\mathcal{B}(t). \tag{6.6}$$

Now differentiate $\mathcal{B}(t)$. Using

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_0^t \mathrm{d}s\, f(t,s) = f(t,t) + \int_0^t \mathrm{d}s\, \frac{\partial f(t,s)}{\partial t}, \tag{6.7}$$

and

$$\frac{\partial}{\partial t}\mathrm{e}^{-\gamma(t-s)} = -\gamma \mathrm{e}^{-\gamma(t-s)}, \tag{6.8}$$

we obtain

$$\boxed{\dot{\mathcal{B}}(t) = -\gamma\mathcal{B}(t) - \mathrm{i}\left[cV^+(t) - \bar{c}V^-(t)\right].} \tag{6.9}$$

This equation is local in time.

Instead of recomputing an integral over the full trajectory at every time step, one only needs to propagate the current value of $\mathcal{B}(t)$.

The computational reason exponentials are useful

For

$$M(t) = \int_0^t \mathrm{e}^{-\gamma(t-s)} f(s)\, \mathrm{d}s,$$

one has

$$\dot{M}(t) = -\gamma M(t) + f(t).$$

Thus an exponential convolution can be propagated recursively without retaining the complete time history.

6.3 Introducing the auxiliary density operators

We now insert powers of the memory variable into the reduced-density path integral and define

$$\rho_n(t) \mathrel{\widehat{=}} \int \mathcal{D}q^+\mathcal{D}q^-\, \mathrm{e}^{\mathrm{i}S_{\mathrm{S}}[q^+]-\mathrm{i}S_{\mathrm{S}}[q^-]}\mathcal{B}(t)^n\mathcal{F}[q^+,q^-], \qquad n = 0,1,2,\ldots. \tag{6.10}$$

Endpoint and initial-state integrations have been suppressed for compactness.

The lowest member contains no inserted memory variable,

$$\boxed{\rho_0(t) = \rho_S(t),} \tag{6.11}$$

and is therefore the physical reduced density operator.

The quantities

$$\rho_1, \rho_2, \ldots \tag{6.12}$$

are called *auxiliary density operators* (ADOs). Despite the name, they are not generally physical density matrices. Their purpose is to store the information required to propagate $\rho_S(t)$ without explicitly storing its full past history.

6.4 Differentiating an ADO

Three contributions appear when ρ_n is differentiated.

1. System propagation. The system propagator gives

$$-\mathrm{i}\mathcal{L}_S\rho_n, \tag{6.13}$$

where

$$\mathcal{L}_S X = [\hat{H}_S, X]. \tag{6.14}$$

Thus, in an actual program,

$$-\mathrm{i}\mathcal{L}_S\rho_n = -\mathrm{i}\left(\hat{H}_S\rho_n - \rho_n\hat{H}_S\right). \tag{6.15}$$

This is just ordinary matrix multiplication.

2. Differentiation of $\mathcal{B}^n$. Using

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathcal{B}^n = n\mathcal{B}^{n-1}\dot{\mathcal{B}} \tag{6.16}$$

and Eq. (6.9),

$$\frac{\mathrm{d}}{\mathrm{d}t}\mathcal{B}^n = -n\gamma\mathcal{B}^n - \mathrm{i}n\mathcal{B}^{n-1}\left[cV^+(t) - \bar{c}V^-(t)\right]. \tag{6.17}$$

The first term produces

$$-n\gamma\rho_n, \tag{6.18}$$

while the second couples ρ_n to ρ_{n-1}.

3. Differentiation of the influence functional. From Eq. (4.3),

$$\frac{\dot{\mathcal{F}}}{\mathcal{F}} = -\left[V^+(t) - V^-(t)\right]A(t). \tag{6.19}$$

Since $A(t) = \mathrm{i}\mathcal{B}(t)$,

$$\boxed{\dot{\mathcal{F}} = -\mathrm{i}\left[V^+(t) - V^-(t)\right]\mathcal{B}(t)\mathcal{F}.} \tag{6.20}$$

This introduces one additional factor of $\mathcal{B}$, and hence couples ρ_n to ρ_{n+1}.

At the endpoint of the path integral,

$$V^{+}(t)X \longrightarrow \hat{V}X, \qquad V^{-}(t)X \longrightarrow X\hat{V}. \tag{6.21}$$

Consequently,

$$V^{+}(t) - V^{-}(t) \longrightarrow [\hat{V}, \cdot]. \tag{6.22}$$

Combining the three contributions gives

$$\boxed{\dot{\rho}_n = -\left(\mathrm{i}\mathcal{L}_{\mathrm{S}} + n\gamma\right)\rho_n - \mathrm{i}[\hat{V}, \rho_{n+1}] - \mathrm{i}n\left(c\,\hat{V}\rho_{n-1} - \bar{c}\,\rho_{n-1}\hat{V}\right).} \tag{6.23}$$

6.5 What equations would actually be integrated?

It is useful to write out the first few equations explicitly, when truncating the hierarchy at $n_{\max} = 2$.

For the physical density matrix,

$$\boxed{\dot{\rho}_0 = -\mathrm{i}[\hat{H}_{\mathrm{S}}, \rho_0] - \mathrm{i}[\hat{V}, \rho_1].} \tag{6.24}$$

For the first ADO,

$$\boxed{\dot{\rho}_1 = -\mathrm{i}[\hat{H}_{\mathrm{S}}, \rho_1] - \gamma\rho_1 - \mathrm{i}[\hat{V}, \rho_2] - \mathrm{i}\left(c\hat{V}\rho_0 - \bar{c}\rho_0\hat{V}\right).} \tag{6.25}$$

For the second ADO,

$$\boxed{\dot{\rho}_2 = -\mathrm{i}[\hat{H}_{\mathrm{S}}, \rho_2] - 2\gamma\rho_2 - \mathrm{i}[\hat{V}, \rho_3] - 2\mathrm{i}\left(c\hat{V}\rho_1 - \bar{c}\rho_1\hat{V}\right).} \tag{6.26}$$

Therefore a numerical implementation does not propagate a memory integral. It propagates the matrices

$$\{\rho_0, \rho_1, \rho_2, \ldots\} \tag{6.27}$$

simultaneously using ordinary first-order ODEs.

If the hierarchy is truncated, for example, at $n_{\max} = 2$, the simplest "hard cutoff" approximation sets

$$\rho_3 = 0. \tag{6.28}$$

The numerical state then consists only of

$$\rho_0, \qquad \rho_1, \qquad \rho_2. \tag{6.29}$$

For a d-dimensional system, each of these is a $d \times d$ complex matrix. Thus the complete ODE state contains

$$3d^2 \tag{6.30}$$

complex numbers in this example.

6.6 From one exponential to many memory modes

A realistic bath correlation function is represented as

$$\boxed{C(t) = \sum_{k=1}^{N_{\mathrm{exp}}} c_k \mathrm{e}^{-\gamma_k t}.} \tag{6.31}$$

The backward-branch function is represented in the corresponding basis as

$$\boxed{C^*(t) = \sum_{k=1}^{N_{\mathrm{exp}}} \bar{c}_k \mathrm{e}^{-\gamma_k t}.} \tag{6.32}$$

The quantities needed by the numerical HEOM calculation are therefore simply the three arrays

$$\boxed{\{c_k\}, \qquad \{\bar{c}_k\}, \qquad \{\gamma_k\}.} \tag{6.33}$$

These coefficients may come from, for example, a Matsubara decomposition, a Padé decomposition, or a direct exponential fit to $C(t)$, as discussed in Sec. 10.

The HEOM propagation itself does not depend on how these coefficients were obtained.

For each exponential define

$$\mathcal{B}_k(t) = -\mathrm{i} \int_0^t \mathrm{d}s\, \mathrm{e}^{-\gamma_k (t-s)} \left[c_k V^+(s) - \bar{c}_k V^-(s) \right], \tag{6.34}$$

which obeys

$$\boxed{\dot{\mathcal{B}}_k = -\gamma_k \mathcal{B}_k - \mathrm{i} \left[c_k V^+(t) - \bar{c}_k V^-(t) \right].} \tag{6.35}$$

Differentiating the influence functional gives

$$\boxed{\dot{\mathcal{F}} = -\mathrm{i} \left[V^+(t) - V^-(t) \right] \sum_k \mathcal{B}_k(t) \mathcal{F}.} \tag{6.36}$$

6.7 The multi-index used in an actual HEOM program

With N_{exp} modes, each ADO contains a product

$$\mathcal{B}_1^{n_1} \mathcal{B}_2^{n_2} \cdots \mathcal{B}_{N_{\mathrm{exp}}}^{n_{N_{\mathrm{exp}}}}. \tag{6.37}$$

We therefore label an ADO by the integer vector

$$\boxed{\boldsymbol{n} = (n_1, n_2, \ldots, n_{N_{\mathrm{exp}}}),} \tag{6.38}$$

where every

$$n_k = 0, 1, 2, \ldots. \tag{6.39}$$

The corresponding ADO is

$$\rho_{\boldsymbol{n}}(t) \mathrel{\widehat{=}} \int \mathcal{D}q^+ \mathcal{D}q^-\, \mathrm{e}^{\mathrm{i}S_{\mathrm{S}}[q^+] - \mathrm{i}S_{\mathrm{S}}[q^-]} \prod_k \mathcal{B}_k(t)^{n_k} \mathcal{F}[q^+, q^-]. \tag{6.40}$$

The physical state is

$$\boxed{\rho_{\mathbf{0}} = \rho_{\mathrm{S}}, \qquad \mathbf{0} = (0, 0, \ldots, 0).} \tag{6.41}$$

Let

$$\boldsymbol{e}_k = (0, \ldots, 0, 1, 0, \ldots, 0) \tag{6.42}$$

denote the unit vector in direction k.

For example, if $N_{\mathrm{exp}} = 3$ and

$$\boldsymbol{n} = (2, 0, 1), \tag{6.43}$$

then

$$\boldsymbol{n} + \boldsymbol{e}_2 = (2, 1, 1), \tag{6.44}$$

whereas

$$\boldsymbol{n} - \boldsymbol{e}_1 = (1, 0, 1). \tag{6.45}$$

These are precisely the neighboring ADOs required by the HEOM.

The full hierarchy is

$$\boxed{\begin{aligned} \dot{\rho}_{\boldsymbol{n}} = \; & -\mathrm{i}\left[\hat{H}_{\mathrm{S}}, \rho_{\boldsymbol{n}}\right] - \left(\sum_k n_k \gamma_k\right) \rho_{\boldsymbol{n}} \\ & -\mathrm{i} \sum_k \left[\hat{V}, \rho_{\boldsymbol{n}+\boldsymbol{e}_k}\right] \\ & -\mathrm{i} \sum_k n_k \left(c_k \hat{V} \rho_{\boldsymbol{n}-\boldsymbol{e}_k} - \bar{c}_k \rho_{\boldsymbol{n}-\boldsymbol{e}_k} \hat{V}\right). \end{aligned}} \tag{6.46}$$

This form is particularly useful for implementation because every term can be translated directly into matrix operations.

6.8 Truncating the infinite hierarchy

The exact hierarchy contains infinitely many ADOs, so it must be truncated for numerical work.

Define the hierarchy tier

$$\boxed{L(\boldsymbol{n}) = \sum_k n_k.} \tag{6.47}$$

Choose a maximum tier L_{max} and retain only ADOs satisfying

$$\boxed{\sum_k n_k \le L_{\mathrm{max}}.} \tag{6.48}$$

For example, if there are two exponential modes and $L_{\mathrm{max}} = 2$, the retained multi-indices are

$$\begin{aligned} L = 0: \quad & (0, 0), \\ L = 1: \quad & (1, 0),\ (0, 1), \\ L = 2: \quad & (2, 0),\ (1, 1),\ (0, 2). \end{aligned} \tag{6.49}$$

The number of ADOs retained for N_{exp} exponential modes is

$$\boxed{N_{\mathrm{ADO}} = \binom{N_{\mathrm{exp}} + L_{\mathrm{max}}}{L_{\mathrm{max}}}.} \tag{6.50}$$

This combinatorial growth is one of the principal computational costs of HEOM.

At the simplest hard boundary, if

$$L(\boldsymbol{n}) = L_{\mathrm{max}}, \tag{6.51}$$

then an upward neighbor $\rho_{\boldsymbol{n}+\boldsymbol{e}_k}$ is outside the hierarchy and is set to zero,

$$\rho_{\boldsymbol{n}+\boldsymbol{e}_k} \approx 0. \tag{6.52}$$

More sophisticated terminators can improve convergence, but the hard cutoff is particularly useful for understanding and testing a first implementation.

6.9 Initial conditions

For the usual factorized initial state

$$\rho_{\mathrm{tot}}(0) = \rho_{\mathrm{S}}(0) \otimes \rho_{\mathrm{B}}^{\mathrm{eq}}, \tag{6.53}$$

the memory integrals initially vanish,

$$\mathcal{B}_k(0) = 0. \tag{6.54}$$

Therefore

$$\boxed{\rho_{\mathbf{0}}(0) = \rho_{\mathrm{S}}(0), \qquad \rho_{\boldsymbol{n}\neq\mathbf{0}}(0) = 0.} \tag{6.55}$$

This is particularly simple computationally: initialize an array of ADOs to zero and place the initial physical density matrix in the element corresponding to $\boldsymbol{n} = \mathbf{0}$.

Correlated initial system–bath states require different initial ADOs and are not considered here.

Part III

Implementing HEOM

Part III orientation

Input: the generic arrays $\{c_k, \bar{c}_k, \gamma_k\}$ and a hierarchy cutoff L_{max}.
Output: a numerical trajectory for the physical zero-tier operator $\rho_{\mathbf{0}}(t) = \rho_{\mathrm{S}}(t)$.
Implementation task: generate the hierarchy graph once, then evaluate only matrix operations and neighbor lookups inside the ODE right-hand side.

7 From the HEOM equation to a numerical algorithm

For each stored ADO $\rho_{\boldsymbol{n}}$, the right-hand side of Eq. (6.46) is assembled as follows. The essential dictionary is

$$\boxed{\begin{array}{rcl} \boldsymbol{n} & \longleftrightarrow & \texttt{multiindices[a]}, \\ \rho_{\boldsymbol{n}} & \longleftrightarrow & \texttt{ados[a]}, \\ \boldsymbol{n}+\boldsymbol{e}_k & \longleftrightarrow & \texttt{index\_of.get(n\_up)}, \\ \boldsymbol{n}-\boldsymbol{e}_k & \longleftrightarrow & \texttt{index\_of[n\_down]}, \\ \dot{\rho}_{\boldsymbol{n}} & \longleftrightarrow & \texttt{dados[a]}. \end{array}} \tag{7.1}$$

1. Start with ordinary system evolution,

$$\dot{\rho}_{\boldsymbol{n}} \leftarrow -\mathrm{i}\left(\hat{H}_{\mathrm{S}}\rho_{\boldsymbol{n}} - \rho_{\boldsymbol{n}}\hat{H}_{\mathrm{S}}\right). \tag{7.2}$$

2. Add the memory-decay term,

$$\dot{\rho}_{\boldsymbol{n}} \leftarrow \dot{\rho}_{\boldsymbol{n}} - \left(\sum_k n_k\gamma_k\right)\rho_{\boldsymbol{n}}. \tag{7.3}$$

3. For every exponential mode k, look for the higher ADO

$$\rho_{\boldsymbol{n}+\boldsymbol{e}_k}. \tag{7.4}$$

   If it is included in the truncated hierarchy, add

$$-\mathrm{i}\left(\hat{V}\rho_{\boldsymbol{n}+\boldsymbol{e}_k} - \rho_{\boldsymbol{n}+\boldsymbol{e}_k}\hat{V}\right). \tag{7.5}$$

4. If $n_k > 0$, the lower ADO

$$\rho_{\boldsymbol{n}-\boldsymbol{e}_k} \tag{7.6}$$

   exists. Add

$$-\mathrm{i}n_k\left(c_k\hat{V}\rho_{\boldsymbol{n}-\boldsymbol{e}_k} - \bar{c}_k\rho_{\boldsymbol{n}-\boldsymbol{e}_k}\hat{V}\right). \tag{7.7}$$

5. Repeat these operations for every ADO in the hierarchy.

Thus the implementation requires only matrix products, matrix additions, and lookup of neighboring multi-indices.

What is actually stored in memory?

A numerical HEOM calculation does *not* store the previous trajectory $\rho_{\mathrm{S}}(t')$ for every $t' < t$. Instead, it stores the current collection

$$\{\rho_{\boldsymbol{n}}(t)\}.$$

The ADOs are a compressed dynamical representation of the bath history.

7.1 How the hierarchy is stored in code

Suppose the system Hilbert-space dimension is d. Every ADO is then a $d \times d$ complex matrix.

A convenient implementation is to construct a list

$$\{\boldsymbol{n}^{(0)}, \boldsymbol{n}^{(1)}, \ldots, \boldsymbol{n}^{(N_{\mathrm{ADO}}-1)}\} \tag{7.8}$$

containing all allowed multi-indices and a lookup table

$$\boldsymbol{n} \longrightarrow \text{array index}. \tag{7.9}$$

For example,

$$(0,0) \to 0, \qquad (1,0) \to 1, \qquad (0,1) \to 2, \qquad \ldots \tag{7.10}$$

The complete dynamical state may then be stored either as an array

$$\rho_{\mathrm{ADO}} \in \mathbb{C}^{N_{\mathrm{ADO}} \times d \times d}, \tag{7.11}$$

or flattened into a one-dimensional vector for a standard ODE solver,

$$y(t) = \mathrm{vec}\left(\rho_{\boldsymbol{n}^{(0)}}, \rho_{\boldsymbol{n}^{(1)}}, \ldots\right). \tag{7.12}$$

At every call to the ODE right-hand side:

1. reshape y into the collection of ADO matrices;
2. evaluate Eq. (6.46) for every allowed $\boldsymbol{n}$;
3. flatten the derivatives back into the form expected by the ODE solver.

7.2 Pseudocode for the HEOM right-hand side

The numerical structure of Eq. (6.46) can be summarized by the following pseudocode:

```
for each ADO index a:
    n   = multiindex[a]
    rho = ADO[a]

    drho = -i * (H @ rho - rho @ H)

    decay = sum_k n[k] * gamma[k]
    drho -= decay * rho

    for k in exponential_modes:

        n_up = n + e_k
        if n_up is retained:
            rho_up = ADO[index[n_up]]
            drho += -i * (V @ rho_up - rho_up @ V)

        if n[k] > 0:
            n_down = n - e_k
            rho_down = ADO[index[n_down]]

            drho += -i * n[k] * (
                       c[k]    * V @ rho_down
                     - cbar[k] * rho_down @ V
                     )

    dADO[a] = drho
```

Listing 1: Pseudocode for the HEOM right-hand side.

This pseudocode is essentially a direct transcription of Eq. (6.46).

7.3 A minimal two-level-system check

Before turning to the general implementation, it is useful to consider the smallest nontrivial hierarchy explicitly. For a qubit with

$$\hat{H}_{\mathrm{S}} = \frac{\epsilon}{2}\sigma_z + \frac{\Delta}{2}\sigma_x, \qquad \hat{V} = \sigma_z, \tag{7.13}$$

and a bath correlation function containing a single exponential,

$$C(t) = c, \mathrm{e}^{-\gamma t}, \tag{7.14}$$

each auxiliary density operator is labeled by a single nonnegative integer n. Truncating the hierarchy at

$$L_{\max} = 2 \tag{7.15}$$

therefore leaves only

$$\rho_0, \qquad \rho_1, \qquad \rho_2, \tag{7.16}$$

with the hard truncation condition

$$\rho_3 = 0. \tag{7.17}$$

The explicit equations of motion for these three matrices were given in equations (6.24) to (6.26). Thus, a direct implementation requires only three 2×2 complex matrices. At each time step, one evaluates their coherent system evolution, the damping of the auxiliary operators, and the nearest-neighbor hierarchy couplings

$$\rho_0 \longleftrightarrow \rho_1 \longleftrightarrow \rho_2, \tag{7.18}$$

while any coupling from ρ_2 to the next tier is discarded because $\rho_3 = 0$.

This three-node hierarchy provides a useful consistency check before introducing the general code. Every term can be written and inspected by hand, while the numerical implementation should reproduce exactly the same structure. The only conceptual step needed to pass to several exponential terms is to replace the single hierarchy index n by a multi-index $\mathbf{n} = (n_1, n_2, \ldots)$ and generate the corresponding neighboring auxiliary density operators automatically.

7.4 What is extracted after propagation?

Only the zero-tier ADO is the physical reduced density matrix,

$$\rho_{\mathrm{S}}(t) = \rho_{\mathbf{0}}(t). \tag{7.19}$$

For an observable $\hat{O}$,

$$\boxed{\langle O(t)\rangle = \mathrm{Tr}_{\mathrm{S}}\left[\hat{O}\rho_{\mathbf{0}}(t)\right].} \tag{7.20}$$

The higher-tier ADOs are propagated because they carry bath-memory information, not because they correspond to independently observable system states.

For a qubit, for example,

$$\langle \sigma_z(t)\rangle = \mathrm{Tr}\left[\sigma_z\rho_{\mathbf{0}}(t)\right]. \tag{7.21}$$

7.5 Practical convergence checks

A numerical HEOM calculation has two distinct convergence requirements.

First, the exponential representation must reproduce the bath correlation function accurately,

$$C(t) \simeq \sum_k c_k \mathrm{e}^{-\gamma_k t}. \tag{7.22}$$

This should be checked before performing the HEOM propagation.

Second, the hierarchy must be sufficiently deep. Calculations should therefore be repeated for increasing values of

$$L_{\max} \tag{7.23}$$

until the physical observables obtained from $\rho_{\mathbf{0}}(t)$ cease to change within the desired accuracy.

For example,

$$L_{\max} = 2,\ 3,\ 4, \ldots \tag{7.24}$$

may be compared until

$$\langle O(t) \rangle_{L_{\max}} \tag{7.25}$$

is converged.

The following numerical checks are also useful:

$$\operatorname{Tr} \rho_{\mathbf{0}}(t) \simeq 1, \tag{7.26}$$

$$\rho_{\mathbf{0}}(t) \simeq \rho_{\mathbf{0}}^{\dagger}(t). \tag{7.27}$$

Loss of trace or Hermiticity usually indicates an implementation error or insufficient numerical integration accuracy.

Because the rates γ_k may span widely different time scales, HEOM equations can become stiff. In that regime an ODE method suitable for stiff systems is often more efficient than a simple explicit integrator.

7.6 Complete implementation workflow

The derivation can therefore be translated into the following concrete workflow:

1. Specify the system Hamiltonian $\hat{H}_{\mathrm{S}}$ and coupling operator $\hat{V}$.
2. Determine the bath correlation function $C(t)$ from the chosen spectral density and temperature.
3. Represent it as
$$C(t) \simeq \sum_k c_k \mathrm{e}^{-\gamma_k t}, \qquad C^*(t) \simeq \sum_k \bar{c}_k \mathrm{e}^{-\gamma_k t}.$$
4. Choose a hierarchy depth $L_{\max}$.
5. Generate every non-negative multi-index satisfying
$$\sum_k n_k \leq L_{\max}.$$
6. Store a $d \times d$ complex matrix $\rho_{\boldsymbol{n}}$ for every such multi-index.
7. Initialize
$$\rho_{\mathbf{0}}(0) = \rho_{\mathrm{S}}(0), \qquad \rho_{\boldsymbol{n} \neq 0}(0) = 0.$$

8. At every ODE time step, evaluate Eq. (6.46) for all ADOs.
9. Propagate the coupled ODE system to the desired final time.
10. Extract the physical reduced state as
$$\rho_S(t) = \rho_{\mathbf{0}}(t).$$
11. Increase both the accuracy of the exponential decomposition and $L_{\max}$ until the desired observables are converged.

From the derivation to the program

The objects $\mathcal{B}_k(t)$ are essential for understanding the derivation, but they normally do not need to be propagated explicitly in an HEOM code.
After the hierarchy has been derived, the program propagates only

$$\boxed{\{\rho_{\boldsymbol{n}}(t)\}.}$$

Thus the practical transition is

$$\text{bath correlation function} \longrightarrow \{c_k, \bar{c}_k, \gamma_k\} \longrightarrow \{\boldsymbol{n}\} \longrightarrow \{\rho_{\boldsymbol{n}}\} \longrightarrow \text{coupled matrix ODEs.}$$

This is the computational content of the HEOM construction.

8 A minimal HEOM implementation in Python

The previous section exposed the algorithm independently of any programming language. We now write the same hierarchy as executable Python, keeping the correspondence with equation (7.1) visible throughout. We now translate the hierarchy derived in section 6 directly into a numerical solver. The purpose of this section is not yet to construct a particular spectral-density decomposition, but rather to show how a set of exponential coefficients

$$\{c_k, \bar{c}_k, \gamma_k\} \tag{8.1}$$

is converted into a working propagation of the reduced density matrix.

The central equation to implement is

$$\boxed{\begin{aligned}\dot{\rho}_{\boldsymbol{n}} = &-\mathrm{i}[\hat{H}_S, \rho_{\boldsymbol{n}}] - \left(\sum_k n_k \gamma_k\right)\rho_{\boldsymbol{n}} \\ &-\mathrm{i}\sum_k [\hat{V}, \rho_{\boldsymbol{n}+\boldsymbol{e}_k}] \\ &-\mathrm{i}\sum_k n_k \left(c_k \hat{V}\rho_{\boldsymbol{n}-\boldsymbol{e}_k} - \bar{c}_k \rho_{\boldsymbol{n}-\boldsymbol{e}_k}\hat{V}\right).\end{aligned}} \tag{8.2}$$

Every operation appearing in this equation is either a matrix multiplication, a matrix addition, or a lookup of another ADO in the hierarchy. Once the multi-indices have been generated, the numerical implementation is therefore quite direct.

8.1 Step 1: generate the hierarchy

For N_{exp} exponential modes and maximum hierarchy depth $L_{\max}$, we retain every non-negative multi-index

$$\boldsymbol{n} = (n_1, \ldots, n_{N_{\text{exp}}}) \tag{8.3}$$

satisfying

$$\sum_k n_k \leq L_{\max}. \tag{8.4}$$

For example, with two exponential modes and $L_{\max} = 2$,

$$\begin{aligned} L = 0 &: \quad (0,0), \\ L = 1 &: \quad (0,1),\ (1,0), \\ L = 2 &: \quad (0,2),\ (1,1),\ (2,0). \end{aligned} \tag{8.5}$$

The following Python function generates these indices automatically:

```python
import numpy as np
from itertools import product

def generate_multiindices(n_exp, L_max):
    """
    Generate every multi-index n = (n_1,...,n_nexp)
    satisfying sum(n_k) <= L_max.
    """
    multiindices = [
        n for n in product(range(L_max + 1), repeat=n_exp)
        if sum(n) <= L_max
    ]

    # Sort first by hierarchy tier and then lexicographically.
    multiindices.sort(key=lambda n: (sum(n), n))

    return multiindices
```

Listing 2: Code to generate the hierarchy.

For example,

```python
multiindices = generate_multiindices(n_exp=2, L_max=2)

for n in multiindices:
    print(n)
```

returns

```
(0, 0)
(0, 1)
(1, 0)
(0, 2)
(1, 1)
(2, 0)
```

The number of indices generated should agree with

$$N_{\text{ADO}} = \binom{N_{\text{exp}} + L_{\max}}{L_{\max}}. \tag{8.6}$$

8.2 Step 2: construct an index lookup table

During the propagation of a particular ADO $\rho_{\boldsymbol{n}}$, we repeatedly need its neighbors

$$\rho_{\boldsymbol{n}+\boldsymbol{e}_k} \qquad \text{and} \qquad \rho_{\boldsymbol{n}-\boldsymbol{e}_k}. \tag{8.7}$$

It is inefficient to search the complete list every time. We therefore create a dictionary mapping each multi-index to its position in the ADO array:

```
index_of = {
    n: i for i, n in enumerate(multiindices)
}
```

We can then obtain the array position of an ADO simply from

```
a = index_of[n]
```

or test whether an upward neighbor has survived the hierarchy truncation by

```
a_up = index_of.get(n_up)

if a_up is not None:
    ...
```

This lookup table is an important practical detail: the hierarchy is naturally labelled by multi-indices, whereas a numerical array is labelled by ordinary integer positions.

8.3 Step 3: storing all ADOs

For a system Hilbert space of dimension d, every ADO is a complex $d \times d$ matrix.

We store all ADOs in a single array,

$$\texttt{ados.shape} = (N_{\mathrm{ADO}}, d, d). \tag{8.8}$$

Thus

```
ados[a]
```

contains the matrix corresponding to the multi-index

```
multiindices[a].
```

Standard ODE solvers usually expect the dynamical state to be a one-dimensional vector. We therefore flatten the complete collection of ADO matrices before passing it to the integrator,

$$\{\rho_{\boldsymbol{n}}\} \longrightarrow y. \tag{8.9}$$

Inside the right-hand-side function, we reshape it back into matrices.

8.4 Step 4: implementing the HEOM right-hand side

We now implement Eq. (8.2) term by term.

The complete function is

```python
def heom_rhs(
    t,
    y,
    H,
    V,
    c,
    cbar,
    gamma,
    multiindices,
    index_of,
):
    """
    Right-hand side of the unscaled HEOM.

    Parameters
    ----------
    H : (d,d) complex array
        System Hamiltonian.

    V : (d,d) complex array
        System-bath coupling operator.

    c, cbar, gamma : arrays of length N_exp
        Exponential-decomposition coefficients.

    multiindices : list of tuples
        Retained hierarchy indices.

    index_of : dict
        Mapping multi-index -> array position.
    """

    d = H.shape[0]
    n_ado = len(multiindices)
    n_exp = len(gamma)

    # Recover the ADO matrices from the flattened ODE vector.
    ados = y.reshape((n_ado, d, d))

    # Array containing the time derivatives.
    dados = np.zeros_like(ados)

    for a, n in enumerate(multiindices):

        rho = ados[a]

        # ------------------------------------------------
        # 1. System Liouvillian:
        #
        #    -i [H, rho_n]
        # ------------------------------------------------
        drho = -1j * (H @ rho - rho @ H)

        # ------------------------------------------------
        # 2. Hierarchy damping:
        #
        #    -(sum_k n_k gamma_k) rho_n
        # ------------------------------------------------
        decay = sum(
            n[k] * gamma[k]
            for k in range(n_exp)
        )

        drho -= decay * rho

        # ------------------------------------------------
        # 3. Upward and downward hierarchy couplings.
```

```
        # ------------------------------------------------
        for k in range(n_exp):

            # ================================================
            # Upward coupling:
            #
            #   -i [V, rho_{n + e_k}]
            # ================================================

            n_up = list(n)
            n_up[k] += 1
            n_up = tuple(n_up)

            a_up = index_of.get(n_up)

            if a_up is not None:

                rho_up = ados[a_up]

                drho += -1j * (
                    V @ rho_up
                    - rho_up @ V
                )

            # ================================================
            # Downward coupling:
            #
            # -i n_k (
            #      c_k    V rho_{n-e_k}
            #    - cbar_k rho_{n-e_k} V
            # )
            # ================================================

            if n[k] > 0:

                n_down = list(n)
                n_down[k] -= 1
                n_down = tuple(n_down)

                a_down = index_of[n_down]
                rho_down = ados[a_down]

                drho += -1j * n[k] * (
                    c[k] * (V @ rho_down)
                    - cbar[k] * (rho_down @ V)
                )

        dados[a] = drho

    # Return the derivatives in the flattened form
    # expected by the ODE solver.
    return dados.reshape(-1)
```

This function is simply Eq. (8.2) written in Python.

The correspondence can be summarized as

$$
\begin{aligned}
-\mathrm{i}[\hat{H}_{\mathrm{S}}, \rho_{\boldsymbol{n}}] \quad &\longleftrightarrow \quad \texttt{-1j*(H@rho-rho@H)}, \\
-\sum_k n_k \gamma_k \rho_{\boldsymbol{n}} \quad &\longleftrightarrow \quad \texttt{-decay*rho}, \\
-\mathrm{i}[\hat{V}, \rho_{\boldsymbol{n}+\boldsymbol{e}_k}] \quad &\longleftrightarrow \quad \texttt{-1j*(V@rho\_up-rho\_up@V)}, \\
-\mathrm{i} n_k \left(c_k \hat{V} \rho_{\boldsymbol{n}-\boldsymbol{e}_k} - \bar{c}_k \rho_{\boldsymbol{n}-\boldsymbol{e}_k} \hat{V} \right) \quad &\longleftrightarrow \quad \texttt{downward coupling}.
\end{aligned}
\tag{8.10}
$$

This one-to-one correspondence is useful when debugging an implementation.

8.5 Step 5: a concrete qubit example

Consider the two-level Hamiltonian

$$\hat{H}_{\mathrm{S}} = \frac{\epsilon}{2}\sigma_z + \frac{\Delta}{2}\sigma_x \tag{8.11}$$

with system–bath coupling

$$\hat{V} = \sigma_z. \tag{8.12}$$

The Pauli matrices are

```
sx = np.array(
    [[0.0, 1.0],
     [1.0, 0.0]],
    dtype=complex,
)

sz = np.array(
    [[1.0,  0.0],
     [0.0, -1.0]],
    dtype=complex,
)
```

We choose, for illustration,

```
epsilon = 1.0
Delta   = 1.0

H = 0.5 * epsilon * sz + 0.5 * Delta * sx
V = sz
```

For the moment, suppose the bath correlation function contains only one exponential,

$$C(t) = c\,\mathrm{e}^{-\gamma t}. \tag{8.13}$$

We use illustrative parameters

```
c     = np.array([0.10 - 0.05j])
cbar  = np.array([0.10 + 0.05j])
gamma = np.array([1.0 + 0.0j])
```

These numbers are used only to test the HEOM machinery. In the next section they will be replaced by coefficients obtained systematically from a physical spectral density and temperature.

8.6 Step 6: initialize the hierarchy

Let us truncate the hierarchy at

$$L_{\max} = 4. \tag{8.14}$$

For one exponential mode, this retains

$$n = 0, 1, 2, 3, 4. \tag{8.15}$$

The hierarchy is constructed as

```
L_max = 4

multiindices = generate_multiindices(
    n_exp=len(gamma),
```

```
    L_max=L_max,
)

index_of = {
    n: i for i, n in enumerate(multiindices)
}

n_ado = len(multiindices)

print("Number of ADOs =", n_ado)
print("Multi-indices =", multiindices)
```

For this example the result is

$$N_{\mathrm{ADO}} = 5. \tag{8.16}$$

Suppose the qubit initially occupies the first basis state,

$$\rho_{\mathrm{S}}(0) = |0\rangle\langle 0| = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}. \tag{8.17}$$

For a factorized system–bath initial condition,

$$\rho_{\mathbf{0}}(0) = \rho_{\mathrm{S}}(0), \tag{8.18}$$

while every higher-tier ADO initially vanishes.

In Python,

```
rho0 = np.array(
    [[1.0, 0.0],
     [0.0, 0.0]],
    dtype=complex,
)

d = H.shape[0]

ados0 = np.zeros(
    (n_ado, d, d),
    dtype=complex,
)

physical_index = index_of[(0,) * len(gamma)]

ados0[physical_index] = rho0

y0 = ados0.reshape(-1)
```

Notice that we do not separately initialize or propagate the memory functions $\mathcal{B}_k(t)$. Their effect is already encoded in the hierarchy.

8.7 Step 7: propagate the hierarchy

The complete collection of ADOs can now be propagated using a standard ODE solver. For example,

```
from scipy.integrate import solve_ivp

t0 = 0.0
tf = 20.0
```

```
t_eval = np.linspace(t0, tf, 1001)

sol = solve_ivp(
    fun=lambda t, y: heom_rhs(
        t,
        y,
        H,
        V,
        c,
        cbar,
        gamma,
        multiindices,
        index_of,
    ),
    t_span=(t0, tf),
    y0=y0,
    t_eval=t_eval,
    method="DOP853",
    rtol=1e-9,
    atol=1e-11,
)
```

The ODE integrator sees one large vector $y(t)$, but physically that vector is just the concatenation of all ADO matrices.

For strongly separated decay rates γ_k, the hierarchy may become stiff. In that situation a stiff ODE integrator can be advantageous. The important point is that changing the integration method does not alter the HEOM itself.

8.8 Step 8: extract the physical reduced density matrix

The solver returns all hierarchy elements, but only

$$\rho_{\mathbf{0}}(t) \tag{8.19}$$

is the physical reduced density matrix.

We therefore extract the zero-tier ADO at each time:

```
rho_t = np.empty(
    (len(sol.t), d, d),
    dtype=complex,
)

for j in range(len(sol.t)):

    ados_t = sol.y[:, j].reshape(
        (n_ado, d, d)
    )

    rho_t[j] = ados_t[physical_index]
```

The diagonal populations are

$$P_0(t) = \rho_{00}(t), \qquad P_1(t) = \rho_{11}(t), \tag{8.20}$$

which can be obtained using

```
P0 = np.real(rho_t[:, 0, 0])
P1 = np.real(rho_t[:, 1, 1])
```

The coherence is

$$\rho_{01}(t), \tag{8.21}$$

so we may also monitor

```
coherence = rho_t[:, 0, 1]
```

For the population difference,

$$\langle \sigma_z(t) \rangle = \mathrm{Tr}[\sigma_z \rho_{\mathrm{S}}(t)], \tag{8.22}$$

we can use

```
sigma_z_t = np.array([
    np.trace(sz @ rho)
    for rho in rho_t
]).real
```

8.9 Step 9: plot the reduced dynamics

The populations can be visualized with

```
import matplotlib.pyplot as plt

plt.figure(figsize=(7, 4))

plt.plot(sol.t, P0, label=r"$\rho_{00}$")
plt.plot(sol.t, P1, label=r"$\rho_{11}$")

plt.xlabel("Time")
plt.ylabel("Population")
plt.legend()
plt.tight_layout()
plt.show()
```

Similarly,

```
plt.figure(figsize=(7, 4))

plt.plot(
    sol.t,
    sigma_z_t,
    label=r"$\langle\sigma_z(t)\rangle$",
)

plt.xlabel("Time")
plt.ylabel(r"$\langle\sigma_z\rangle$")
plt.legend()
plt.tight_layout()
plt.show()
```

The important point is that all physical quantities are calculated from $\rho_{\mathbf{0}}(t)$. The higher-tier ADOs participate in its dynamics but are not themselves interpreted as physical reduced states.

8.10 Step 10: basic numerical diagnostics

Before trusting a HEOM calculation, several simple tests should be performed.

8.10.1 Trace preservation

The physical density matrix should satisfy

$$\operatorname{Tr} \rho_{\mathrm{S}}(t) = 1. \tag{8.23}$$

In Python,

```
trace_t = np.array([
    np.trace(rho)
    for rho in rho_t
])

trace_error = np.max(
    np.abs(trace_t - 1.0)
)

print("Maximum trace error =", trace_error)
```

Because the equation for $\rho_{\mathbf{0}}$ contains only commutators,

$$\frac{\mathrm{d}}{\mathrm{d}t} \operatorname{Tr} \rho_{\mathbf{0}} = 0, \tag{8.24}$$

so a substantial trace error indicates a numerical or implementation problem.

8.10.2 Hermiticity

The physical reduced density matrix should remain Hermitian,

$$\rho_{\mathrm{S}}(t) = \rho_{\mathrm{S}}^{\dagger}(t). \tag{8.25}$$

A useful diagnostic is

```
hermiticity_error = max(
    np.linalg.norm(
        rho - rho.conj().T
    )
    for rho in rho_t
)

print(
    "Maximum Hermiticity error =",
    hermiticity_error,
)
```

8.10.3 Population conservation

For a two-level system,

$$P_0(t) + P_1(t) = 1. \tag{8.26}$$

Thus

```
population_error = np.max(
    np.abs(P0 + P1 - 1.0)
)

print(
    "Maximum population-sum error =",
    population_error,
)
```

provides another convenient check.

8.11 A useful structural unit test

The code can also be checked without knowing an exact open-system solution.

Set all bath coefficients to zero,

$$c_k = \bar{c}_k = 0. \tag{8.27}$$

Then all initially vanishing higher-tier ADOs remain zero. The zero-tier equation reduces to

$$\dot{\rho}_{\mathrm{S}} = -\mathrm{i}[\hat{H}_{\mathrm{S}}, \rho_{\mathrm{S}}], \tag{8.28}$$

which is simply the closed-system von Neumann equation.

The HEOM result can therefore be compared against the exact unitary solution

$$\rho_{\mathrm{S}}(t) = \mathrm{e}^{-\mathrm{i}\hat{H}_{\mathrm{S}}t}\rho_{\mathrm{S}}(0)\mathrm{e}^{+\mathrm{i}\hat{H}_{\mathrm{S}}t}. \tag{8.29}$$

This is an especially useful first test of the hierarchy indexing and array reshaping.

8.12 Checking hierarchy convergence

A finite hierarchy depth is an approximation. The physical result should therefore be recomputed for increasing values of $L_{\max}$.

For example,

$$L_{\max} = 2,\ 3,\ 4,\ 5, \ldots \tag{8.30}$$

should be tested until the relevant observable is unchanged within the desired numerical tolerance.

A simple convergence measure for an observable $O(t)$ is

$$\epsilon_L = \max_t \left| O_{L_{\max}}(t) - O_{L_{\max}-1}(t) \right|. \tag{8.31}$$

The calculation may be considered converged when

$$\epsilon_L < \varepsilon_{\mathrm{target}}. \tag{8.32}$$

It is important to distinguish this hierarchy convergence from convergence of the exponential representation of the bath correlation function. In a realistic calculation both must be checked independently:

$$\boxed{\begin{array}{c}\text{Does } \displaystyle\sum_k c_k \mathrm{e}^{-\gamma_k t} \text{ accurately reproduce } C(t)? \\ \text{Is the physical result converged with respect to } L_{\max}?\end{array}} \tag{8.33}$$

8.13 Complete executable example

For convenience, the essential pieces can be collected into a single minimal script:

```
"""
HEOM_minimal.py

Minimal, self-contained implementation of the unscaled HEOM for a system
whose bath correlation function is represented as

    C(t) = sum_k c_k exp(-gamma_k t).

This example uses a qubit and a single illustrative exponential mode.
The bath coefficients are intentionally not tied to a particular spectral
density; see HEOM_Drude_Lorentz.py for a physical Drude-Lorentz example.
"""

import numpy as np
import matplotlib.pyplot as plt
from itertools import product
from scipy.integrate import solve_ivp


def generate_multiindices(n_exp, L_max):
    """Return all non-negative multi-indices with total tier <= L_max."""
    indices = [
        n
        for n in product(range(L_max + 1), repeat=n_exp)
        if sum(n) <= L_max
    ]
    indices.sort(key=lambda n: (sum(n), n))
    return indices


def heom_rhs(t, y, H, V, c, cbar, gamma, multiindices, index_of):
    """
    Right-hand side of the unscaled HEOM

      dot(rho_n) =
        -i[H,rho_n]
        -(sum_k n_k gamma_k) rho_n
        -i sum_k [V,rho_{n+e_k}]
        -i sum_k n_k
             (c_k V rho_{n-e_k} - cbar_k rho_{n-e_k} V).

    Upward ADOs outside the retained hierarchy are omitted (hard cutoff).
    """
    del t  # Autonomous equations; retained for solve_ivp interface.

    d = H.shape[0]
    n_ado = len(multiindices)
    n_exp = len(gamma)

    ados = y.reshape((n_ado, d, d))
    dados = np.zeros_like(ados)

```

```
    for a, n in enumerate(multiindices):
        rho = ados[a]

        # 1. System Liouvillian: -i[H, rho_n]
        drho = -1j * (H @ rho - rho @ H)

        # 2. Hierarchy damping: -(sum_k n_k gamma_k) rho_n
        decay = sum(n[k] * gamma[k] for k in range(n_exp))
        drho -= decay * rho

        for k in range(n_exp):
            # 3. Upward coupling: -i[V, rho_{n+e_k}]
            n_up = list(n)
            n_up[k] += 1
            n_up = tuple(n_up)

            a_up = index_of.get(n_up)
            if a_up is not None:
                rho_up = ados[a_up]
                drho += -1j * (V @ rho_up - rho_up @ V)

            # 4. Downward coupling
            if n[k] > 0:
                n_down = list(n)
                n_down[k] -= 1
                n_down = tuple(n_down)

                rho_down = ados[index_of[n_down]]

                drho += -1j * n[k] * (
                    c[k] * (V @ rho_down)
                    - cbar[k] * (rho_down @ V)
                )

        dados[a] = drho

    return dados.reshape(-1)


def extract_physical_density(solution, n_ado, d, physical_index):
    """Extract rho_S(t)=rho_0(t) from a solve_ivp solution."""
    rho_t = np.empty((len(solution.t), d, d), dtype=complex)

    for j in range(len(solution.t)):
        hierarchy_t = solution.y[:, j].reshape((n_ado, d, d))
        rho_t[j] = hierarchy_t[physical_index]

    return rho_t


def main():
    # ------------------------------------------------------------
    # Qubit Hamiltonian and coupling operator
    # ------------------------------------------------------------
    sx = np.array(
        [[0.0, 1.0],
         [1.0, 0.0]],
        dtype=complex,
    )

    sz = np.array(
        [[1.0,  0.0],
         [0.0, -1.0]],
        dtype=complex,
    )

    epsilon = 1.0
    Delta = 1.0
```

```
121
122     H = 0.5 * epsilon * sz + 0.5 * Delta * sx
123     V = sz
124
125     # ------------------------------------------------------------
126     # Illustrative one-exponential bath representation
127     # ------------------------------------------------------------
128     c = np.array([0.10 - 0.05j], dtype=complex)
129     cbar = np.array([0.10 + 0.05j], dtype=complex)
130     gamma = np.array([1.0 + 0.0j], dtype=complex)
131
132     # ------------------------------------------------------------
133     # Construct hierarchy
134     # ------------------------------------------------------------
135     L_max = 4
136
137     multiindices = generate_multiindices(
138         n_exp=len(gamma),
139         L_max=L_max,
140     )
141
142     index_of = {n: i for i, n in enumerate(multiindices)}
143     n_ado = len(multiindices)
144
145     # ------------------------------------------------------------
146     # Factorized initial condition
147     # ------------------------------------------------------------
148     rho_initial = np.array(
149         [[1.0, 0.0],
150          [0.0, 0.0]],
151         dtype=complex,
152     )
153
154     d = H.shape[0]
155     ados_initial = np.zeros((n_ado, d, d), dtype=complex)
156
157     zero_index = (0,) * len(gamma)
158     physical_index = index_of[zero_index]
159     ados_initial[physical_index] = rho_initial
160
161     y0 = ados_initial.reshape(-1)
162
163     # ------------------------------------------------------------
164     # Time propagation
165     # ------------------------------------------------------------
166     times = np.linspace(0.0, 20.0, 1001)
167
168     solution = solve_ivp(
169         fun=lambda t, y: heom_rhs(
170             t,
171             y,
172             H,
173             V,
174             c,
175             cbar,
176             gamma,
177             multiindices,
178             index_of,
179         ),
180         t_span=(times[0], times[-1]),
181         y0=y0,
182         t_eval=times,
183         method="DOP853",
184         rtol=1e-9,
185         atol=1e-11,
186     )
187
188     if not solution.success:
```

```
        raise RuntimeError(solution.message)

    # ------------------------------------------------------------
    # Extract physical reduced density matrix and observables
    # ------------------------------------------------------------
    rho_t = extract_physical_density(
        solution,
        n_ado=n_ado,
        d=d,
        physical_index=physical_index,
    )

    P0 = rho_t[:, 0, 0].real
    P1 = rho_t[:, 1, 1].real

    sigma_z_t = np.array(
        [np.trace(sz @ rho).real for rho in rho_t]
    )

    # ------------------------------------------------------------
    # Diagnostics
    # ------------------------------------------------------------
    trace_error = max(
        abs(np.trace(rho) - 1.0)
        for rho in rho_t
    )

    hermiticity_error = max(
        np.linalg.norm(rho - rho.conj().T)
        for rho in rho_t
    )

    population_error = np.max(np.abs(P0 + P1 - 1.0))

    print("Number of exponential modes:", len(gamma))
    print("Hierarchy depth L_max:", L_max)
    print("Number of ADOs:", n_ado)
    print("Maximum trace error:", trace_error)
    print("Maximum Hermiticity error:", hermiticity_error)
    print("Maximum population-sum error:", population_error)

    # ------------------------------------------------------------
    # Population plot
    # ------------------------------------------------------------
    plt.figure(figsize=(7, 4))
    plt.plot(solution.t, P0, label=r"$\rho_{00}$")
    plt.plot(solution.t, P1, label=r"$\rho_{11}$")
    plt.xlabel("Time")
    plt.ylabel("Population")
    plt.legend()
    plt.tight_layout()
    plt.show()

    # ------------------------------------------------------------
    # <sigma_z> plot
    # ------------------------------------------------------------
    plt.figure(figsize=(7, 4))
    plt.plot(solution.t, sigma_z_t, label=r"$\langle\sigma_z(t)\rangle$")
    plt.xlabel("Time")
    plt.ylabel(r"$\langle\sigma_z\rangle$")
    plt.legend()
    plt.tight_layout()
    plt.show()


if __name__ == "__main__":
    main()
```

Listing 3: Complete Python implementation of the generic HEOM solver for a qubit with an exponentially decomposed bath correlation function. The script constructs the hierarchy of auxiliary density operators, propagates the coupled HEOM, extracts the physical reduced density matrix, and performs basic trace, Hermiticity, and population-conservation checks.

8.14 What this code has accomplished

The implementation above makes explicit the sequence

$$\boxed{C(t) \longrightarrow \{c_k, \bar{c}_k, \gamma_k\} \longrightarrow \{\boldsymbol{n}\} \longrightarrow \{\rho_{\boldsymbol{n}}(t)\} \longrightarrow \rho_{\mathrm{S}}(t).} \tag{8.34}$$

The path integral itself no longer appears in the numerical calculation. Its role was to derive the structure of the hierarchy. Once Eq. (8.2) has been obtained, HEOM propagation is an ordinary initial-value problem for a finite set of coupled matrix ODEs.

What remains to make the example physical

The HEOM solver itself is now complete. The illustrative coefficients

$$c_k, \qquad \bar{c}_k, \qquad \gamma_k$$

must next be replaced by coefficients derived from a physically specified bath.
For a harmonic bath this requires the sequence

$$J(\omega),\ T \longrightarrow C(t) \longrightarrow \sum_k c_k \mathrm{e}^{-\gamma_k t}.$$

The next step is therefore to start from a concrete spectral density—for example an Ohmic or Drude–Lorentz bath—derive its correlation function, construct its exponential decomposition, and insert those coefficients directly into the solver developed here.

Part IV

From a physical bath to HEOM coefficients

Part IV orientation

Input: a spectral density $J(\omega)$ and temperature T.
Output: the coefficient arrays $\{c_k, \bar{c}_k, \gamma_k\}$ required by the generic HEOM solver.
Independent convergence variable: the number and type of exponentials used to represent $C(t)$; this is distinct from the hierarchy depth $L_{\max}$.

9 From $J(\omega)$ and T to exponential memory modes

The HEOM solver developed in section 8 assumes that the bath correlation function has already been written in the form

$$C(t) \simeq \sum_{k=0}^{K} c_k \mathrm{e}^{-\gamma_k t}. \tag{9.1}$$

The numerical propagator therefore requires the arrays

$$\boxed{\{c_k\}, \qquad \{\bar{c}_k\}, \qquad \{\gamma_k\}.} \tag{9.2}$$

In a physical calculation, however, one normally starts from a spectral density $J(\omega)$ and a temperature T, not from these exponential coefficients directly.

The purpose of this section is to construct the missing bridge,

$$\boxed{J(\omega), T \longrightarrow C(t) \longrightarrow \{c_k, \bar{c}_k, \gamma_k\} \longrightarrow \text{HEOM propagation.}} \tag{9.3}$$

The general relation between $J(\omega)$ and $C(t)$ was derived in Part I. We now specialize to the Drude–Lorentz spectral density and ask how its correlation function is converted into the finite exponential representation required by HEOM.

Throughout this section we continue to use $\hbar = 1$ and therefore write

$$\beta = \frac{1}{k_{\mathrm{B}} T}. \tag{9.4}$$

All energies, frequencies, inverse times, and temperatures must consequently be expressed in mutually consistent units.

9.1 Why the spectral density alone is not yet an HEOM input

Equation (5.6) is a continuous integral over frequencies. The HEOM derived previously instead requires a discrete exponential representation,

$$C(t) \simeq \sum_{k} c_k \mathrm{e}^{-\gamma_k t}. \tag{9.5}$$

Thus a numerical HEOM calculation contains two distinct representations of the bath:

$$J(\omega) \quad \text{in frequency space,} \tag{9.6}$$

and

$$\{c_k, \gamma_k\} \quad \text{in exponential time-domain form.} \tag{9.7}$$

The latter is what is actually passed to the hierarchy.

For some spectral densities this decomposition must be obtained numerically. For the Drude–Lorentz form, however, it follows analytically from the pole structure of the integrand in Eq. (5.6).

9.2 The Drude–Lorentz spectral density

We choose

$$\boxed{J_{\mathrm{D}}(\omega) = \frac{2\lambda\gamma_{\mathrm{D}}\omega}{\omega^2 + \gamma_{\mathrm{D}}^2}.} \tag{9.8}$$

Here

- λ is the reorganization energy;
- γ_{D} is the Drude relaxation rate;
- $\tau_{\mathrm{B}} \sim \gamma_{\mathrm{D}}^{-1}$ is the characteristic bath-memory time.

With the convention of Eq. (5.6), the reorganization energy is indeed

$$\boxed{\lambda = \frac{1}{\pi}\int_0^\infty \frac{J_{\mathrm{D}}(\omega)}{\omega}\,\mathrm{d}\omega.} \tag{9.9}$$

At low frequency,

$$J_{\mathrm{D}}(\omega) \sim \frac{2\lambda}{\gamma_{\mathrm{D}}}\omega, \qquad \omega \to 0, \tag{9.10}$$

so the Drude–Lorentz bath is Ohmic at low frequency.

The parameter γ_{D} additionally provides a characteristic frequency cutoff and therefore sets the principal environmental correlation time.

10 Matsubara decomposition: contour poles and residues

We now show explicitly why the Drude–Lorentz correlation function becomes a sum of decaying exponentials. The key mathematical ingredients are

1. rewrite the correlation function as a Fourier-type integral containing $\mathrm{e}^{-\mathrm{i}\omega t}$;
2. extend the frequency ω into the complex plane;
3. evaluate the integral using the residue theorem;
4. observe that evaluating $\mathrm{e}^{-\mathrm{i}\omega t}$ at a pole $\omega_p = -\mathrm{i}\gamma_p$ produces the factor $\mathrm{e}^{-\gamma_p t}$.

This last step is the direct reason that poles in the complex-frequency plane generate exponential memory modes in the time domain.

10.1 Writing the correlation function as a contour integral

We start from the harmonic-bath correlation function,

$$C(t) = \frac{1}{\pi}\int_0^\infty \mathrm{d}\omega\, J(\omega)\left[\coth\left(\frac{\beta\omega}{2}\right)\cos(\omega t) - \mathrm{i}\sin(\omega t)\right]. \tag{10.1}$$

To use contour integration, it is convenient to convert the integral over positive frequencies into an integral over the entire real axis. Extend the spectral density as an odd function,

$$J(-\omega) = -J(\omega). \tag{10.2}$$

Using also

$$\coth(-x) = -\coth(x), \tag{10.3}$$

one finds that Eq. (10.1) is equivalently

$$\boxed{C(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \mathrm{d}\omega\, J(\omega)\left[\coth\left(\frac{\beta\omega}{2}\right) + 1\right]\mathrm{e}^{-\mathrm{i}\omega t}.} \tag{10.4}$$

This form is especially useful because all of the time dependence is now contained in the single Fourier factor

$$\mathrm{e}^{-\mathrm{i}\omega t}. \tag{10.5}$$

For compactness, define

$$F(\omega) = J(\omega)\left[\coth\left(\frac{\beta\omega}{2}\right) + 1\right]. \tag{10.6}$$

Then

$$C(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \mathrm{d}\omega\, F(\omega)\mathrm{e}^{-\mathrm{i}\omega t}. \tag{10.7}$$

The problem has therefore become a standard Fourier integral that can be evaluated by extending ω into the complex plane.

10.2 Which half-plane should be used?

Let the complex frequency be

$$\omega = x + \mathrm{i}y. \tag{10.8}$$

The Fourier factor becomes

$$\mathrm{e}^{-\mathrm{i}\omega t} = \mathrm{e}^{-\mathrm{i}(x+\mathrm{i}y)t} = \mathrm{e}^{-\mathrm{i}xt}\mathrm{e}^{yt}. \tag{10.9}$$

For $t > 0$, this factor decays exponentially when

$$y = \operatorname{Im}\omega < 0. \tag{10.10}$$

We therefore close the integration contour with a large semicircle in the *lower* half of the complex ω-plane.

On that semicircle,

$$\mathrm{e}^{yt} \to 0 \qquad (y < 0), \tag{10.11}$$

so, under the usual conditions on the remaining part of the integrand, the contribution from the large semicircle vanishes as its radius tends to infinity.

The original real-frequency integral is then determined entirely by the poles enclosed inside the lower-half-plane contour.

10.3 Using the residue theorem

The residue theorem states that the integral of a meromorphic function around a closed contour is determined by the residues of the poles enclosed by that contour. Because the lower-half-plane contour is traversed clockwise, one has

$$\oint_{C_-} \mathrm{d}\omega\, F(\omega)\mathrm{e}^{-\mathrm{i}\omega t} = -2\pi\mathrm{i}\sum_{\omega_p \in C_-} \operatorname{Res}_{\omega=\omega_p}\left[F(\omega)\mathrm{e}^{-\mathrm{i}\omega t}\right]. \tag{10.12}$$

Since the large semicircle does not contribute for $t > 0$, the contour integral reduces to the original real-axis integral. Consequently,

$$C(t) = -\mathrm{i} \sum_{\omega_p \in C_-} \mathrm{Res}_{\omega=\omega_p} \left[F(\omega) \mathrm{e}^{-\mathrm{i}\omega t} \right]. \tag{10.13}$$

This equation is the crucial step: the continuous frequency integral has been replaced by a discrete sum over the poles of the integrand.

Now suppose that ω_p is a simple pole of $F(\omega)$. The exponential $\mathrm{e}^{-\mathrm{i}\omega t}$ is analytic at ω_p, so it can simply be evaluated at the pole and pulled out of the residue:

$$\mathrm{Res}_{\omega=\omega_p} \left[F(\omega) \mathrm{e}^{-\mathrm{i}\omega t} \right] = \mathrm{e}^{-\mathrm{i}\omega_p t} \, \mathrm{Res}_{\omega=\omega_p} F(\omega). \tag{10.14}$$

Therefore the contribution of one pole has the generic form

$$\boxed{C_p(t) = c_p \, \mathrm{e}^{-\mathrm{i}\omega_p t},} \tag{10.15}$$

where

$$c_p = -\mathrm{i} \, \mathrm{Res}_{\omega=\omega_p} F(\omega) \tag{10.16}$$

contains the residue and hence determines the amplitude of that contribution.

This immediately explains the separation between the two quantities that appear in an HEOM exponential decomposition:

$$\boxed{\begin{array}{ccc} \text{position of the pole} & \longrightarrow & \text{time dependence,} \\ \text{residue at the pole} & \longrightarrow & \text{coefficient.} \end{array}} \tag{10.17}$$

10.4 Why a pole on the imaginary axis gives a decaying exponential

Suppose an enclosed pole lies on the negative imaginary axis,

$$\omega_p = -\mathrm{i}\gamma_p, \qquad \gamma_p > 0. \tag{10.18}$$

Its Fourier factor is

$$\begin{aligned} \mathrm{e}^{-\mathrm{i}\omega_p t} &= \mathrm{e}^{-\mathrm{i}(-\mathrm{i}\gamma_p)t} \\ &= \boxed{\mathrm{e}^{-\gamma_p t}.} \end{aligned} \tag{10.19}$$

Thus a pole located a distance γ_p below the real axis generates a memory component that decays with rate γ_p.

This is precisely the exponential form required by HEOM.

More generally, if a pole is located at

$$\omega_p = \Omega_p - \mathrm{i}\gamma_p, \tag{10.20}$$

then

$$\mathrm{e}^{-\mathrm{i}\omega_p t} = \mathrm{e}^{-\gamma_p t} \mathrm{e}^{-\mathrm{i}\Omega_p t}. \tag{10.21}$$

The imaginary part of the pole therefore determines the decay rate, whereas its real part determines an oscillation frequency. The Drude–Matsubara decomposition considered below is particularly simple because all relevant poles lie on the imaginary axis.

10.5 Poles of the Drude–Lorentz spectral density

For the Drude–Lorentz spectral density,

$$J_{\mathrm{D}}(\omega) = \frac{2\lambda\gamma_{\mathrm{D}}\omega}{\omega^2 + \gamma_{\mathrm{D}}^2}, \tag{10.22}$$

the denominator factorizes as

$$\omega^2 + \gamma_{\mathrm{D}}^2 = (\omega - \mathrm{i}\gamma_{\mathrm{D}})(\omega + \mathrm{i}\gamma_{\mathrm{D}}). \tag{10.23}$$

The spectral density therefore has simple poles at

$$\omega = \pm\mathrm{i}\gamma_{\mathrm{D}}. \tag{10.24}$$

Because we close the contour in the lower half-plane for $t > 0$, the enclosed Drude pole is

$$\boxed{\omega_0 = -\mathrm{i}\gamma_{\mathrm{D}}.} \tag{10.25}$$

Its contribution to the correlation function contains

$$\mathrm{e}^{-\mathrm{i}\omega_0 t} = \mathrm{e}^{-\gamma_{\mathrm{D}} t}. \tag{10.26}$$

Hence the Drude pole generates the exponential memory mode

$$\boxed{C_0(t) = c_0 \mathrm{e}^{-\gamma_{\mathrm{D}} t}.} \tag{10.27}$$

The coefficient c_0 is obtained by calculating the residue at $\omega = -\mathrm{i}\gamma_{\mathrm{D}}$; its explicit value will be derived below.

10.6 Poles of the thermal factor: Matsubara frequencies

The second family of poles comes from the thermal factor

$$\coth\left(\frac{\beta\omega}{2}\right). \tag{10.28}$$

The function $\coth z$ has simple poles whenever

$$\sinh z = 0. \tag{10.29}$$

Since

$$\sinh z = 0 \qquad \Longleftrightarrow \qquad z = \mathrm{i}\pi k \qquad (k = 0, \pm 1, \pm 2, \dots), \qquad i.e., \qquad k \in \mathbb{Z}, \tag{10.30}$$

we obtain

$$\frac{\beta\omega}{2} = \mathrm{i}\pi k, \tag{10.31}$$

or

$$\omega = \mathrm{i}\frac{2\pi k}{\beta}. \tag{10.32}$$

It is conventional to define the positive bosonic Matsubara frequencies

$$\boxed{\nu_k = \frac{2\pi k}{\beta}, \qquad k = 1, 2, \ldots.} \tag{10.33}$$

The corresponding poles occur in pairs at

$$\omega = \pm \mathrm{i}\nu_k. \tag{10.34}$$

Again, because the contour is closed in the lower half-plane, the poles that contribute for $t > 0$ are

$$\boxed{\omega_k = -\mathrm{i}\nu_k, \qquad k = 1, 2, \ldots.} \tag{10.35}$$

Each such pole produces the Fourier factor

$$\mathrm{e}^{-\mathrm{i}\omega_k t} = \mathrm{e}^{-\mathrm{i}(-\mathrm{i}\nu_k)t} = \mathrm{e}^{-\nu_k t}. \tag{10.36}$$

Therefore the kth thermal pole contributes

$$\boxed{C_k(t) = c_k \mathrm{e}^{-\nu_k t},} \tag{10.37}$$

where c_k is determined by the residue at $\omega = -\mathrm{i}\nu_k$.

10.7 Collecting all residues

The lower-half-plane contour therefore contains

$$-\mathrm{i}\gamma_{\mathrm{D}}, \qquad -\mathrm{i}\nu_1, \qquad -\mathrm{i}\nu_2, \qquad \ldots \tag{10.38}$$

and each pole contributes one decaying exponential.

Summing the residues gives

$$\boxed{C(t) = c_0 \mathrm{e}^{-\gamma_{\mathrm{D}} t} + \sum_{k=1}^{\infty} c_k \mathrm{e}^{-\nu_k t}, \qquad t > 0.} \tag{10.39}$$

The origin of this expression can therefore be summarized as

$$\boxed{\begin{aligned} \omega_0 = -\mathrm{i}\gamma_{\mathrm{D}} \quad &\longrightarrow \quad c_0 \mathrm{e}^{-\gamma_{\mathrm{D}} t}, \\ \omega_k = -\mathrm{i}\nu_k \quad &\longrightarrow \quad c_k \mathrm{e}^{-\nu_k t}. \end{aligned}} \tag{10.40}$$

The pole locations determine the decay constants

$$\gamma_{\mathrm{D}}, \qquad \nu_k, \tag{10.41}$$

whereas the corresponding residues determine the amplitudes

$$c_0, \qquad c_k. \tag{10.42}$$

Why contour integration naturally produces HEOM modes

The exponential decomposition used by HEOM is not introduced merely as a convenient fitting ansatz for the Drude bath. It follows directly from the analytic structure of its correlation function.
The correlation function contains the Fourier factor

$$e^{-i\omega t}.$$

The residue theorem converts the frequency integral into a sum over poles, and at each pole this factor becomes

$$e^{-i\omega_p t}.$$

For a pole on the negative imaginary axis,

$$\omega_p = -i\gamma_p,$$

this is

$$\boxed{e^{-i\omega_p t} = e^{-\gamma_p t}.}$$

Therefore

$$\boxed{\text{pole in complex frequency } \longrightarrow \text{ exponential memory mode in time.}}$$

The residue determines how strongly that mode contributes, while the location of the pole determines how rapidly it decays. This is precisely the mathematical structure required to construct the HEOM hierarchy.

10.8 The Drude pole: deriving the coefficient c_0

We now calculate explicitly the coefficient multiplying the exponential generated by the Drude pole. From the contour-integral representation derived above,

$$C(t) = -i \sum_{\omega_p \in C_-} \mathrm{Res}_{\omega=\omega_p} \left[F(\omega) e^{-i\omega t} \right], \tag{10.43}$$

where

$$F(\omega) = J_D(\omega) \left[\coth\left(\frac{\beta\omega}{2}\right) + 1 \right]. \tag{10.44}$$

Because $e^{-i\omega t}$ is analytic at every pole, each contribution can be written as

$$C_p(t) = c_p e^{-i\omega_p t}, \tag{10.45}$$

with

$$\boxed{c_p = -i\ \mathrm{Res}_{\omega=\omega_p} F(\omega).} \tag{10.46}$$

Thus the task is simply to evaluate the appropriate residue.

For the Drude–Lorentz spectral density,

$$J_D(\omega) = \frac{2\lambda\gamma_D\omega}{\omega^2 + \gamma_D^2} = \frac{2\lambda\gamma_D\omega}{(\omega - i\gamma_D)(\omega + i\gamma_D)}, \tag{10.47}$$

the pole enclosed by the lower-half-plane contour is

$$\omega_0 = -i\gamma_D. \tag{10.48}$$

The corresponding time dependence is immediately

$$e^{-i\omega_0 t} = e^{-\gamma_D t}, \tag{10.49}$$

so that

$$\boxed{\gamma_0 = \gamma_D.} \tag{10.50}$$

It remains to calculate the amplitude c_0.

Since the thermal factor is regular at $\omega = -i\gamma_D$, the residue comes entirely from the pole of $J_D(\omega)$:

$$\begin{aligned} \mathrm{Res}_{\omega=-i\gamma_D} F(\omega) = {} & \mathrm{Res}_{\omega=-i\gamma_D} J_D(\omega) \\ & \times \left[\coth\left(-\frac{i\beta\gamma_D}{2} \right) + 1 \right]. \end{aligned} \tag{10.51}$$

Let us evaluate these two factors separately.

Residue of the Drude spectral density. For a simple pole,

$$\begin{aligned} \mathrm{Res}_{\omega=-i\gamma_D} J_D(\omega) &= \lim_{\omega\to-i\gamma_D} (\omega + i\gamma_D) J_D(\omega) \\ &= \lim_{\omega\to-i\gamma_D} \frac{2\lambda\gamma_D\omega}{\omega - i\gamma_D}. \end{aligned} \tag{10.52}$$

Substituting $\omega = -i\gamma_D$ gives

$$\begin{aligned} \mathrm{Res}_{\omega=-i\gamma_D} J_D(\omega) &= \frac{2\lambda\gamma_D(-i\gamma_D)}{-2i\gamma_D} \\ &= \boxed{\lambda\gamma_D.} \end{aligned} \tag{10.53}$$

Thermal factor evaluated at the Drude pole. We next use the identity

$$\coth(-ix) = i\cot x. \tag{10.54}$$

Therefore

$$\coth\left(-\frac{i\beta\gamma_D}{2} \right) = i\cot\left(\frac{\beta\gamma_D}{2} \right). \tag{10.55}$$

The full residue is consequently

$$\mathrm{Res}_{\omega=-i\gamma_D} F(\omega) = \lambda\gamma_D \left[1 + i\cot\left(\frac{\beta\gamma_D}{2} \right) \right]. \tag{10.56}$$

Finally, using Eq. (10.46),

$$\begin{aligned} c_0 &= -i\lambda\gamma_D \left[1 + i\cot\left(\frac{\beta\gamma_D}{2} \right) \right] \\ &= \boxed{\lambda\gamma_D \left[\cot\left(\frac{\beta\gamma_D}{2} \right) - i \right].} \end{aligned} \tag{10.57}$$

Thus the contribution from the Drude pole is

$$\boxed{C_0(t) = \lambda\gamma_{\mathrm{D}}\left[\cot\left(\frac{\beta\gamma_{\mathrm{D}}}{2}\right) - \mathrm{i}\right]\mathrm{e}^{-\gamma_{\mathrm{D}}t}.} \tag{10.58}$$

The coefficient contains both real and imaginary parts,

$$\mathrm{Re}\, c_0 = \lambda\gamma_{\mathrm{D}}\cot\left(\frac{\beta\gamma_{\mathrm{D}}}{2}\right), \tag{10.59}$$

$$\mathrm{Im}\, c_0 = -\lambda\gamma_{\mathrm{D}}. \tag{10.60}$$

The real part is temperature dependent and contributes to the fluctuation part of the bath correlation function, whereas the imaginary part is temperature independent for this spectral density and is associated with the dissipative response.

10.9 The Matsubara poles: deriving the coefficients c_k

We now repeat the same calculation for the thermal poles. The important difference is that these poles do not originate from the spectral density. They originate from

$$\coth\left(\frac{\beta\omega}{2}\right). \tag{10.61}$$

The nonzero thermal poles occur at

$$\omega = \pm\mathrm{i}\nu_k, \qquad \nu_k = \frac{2\pi k}{\beta}, \qquad k = 1, 2, \ldots. \tag{10.62}$$

For $t > 0$, the lower-half-plane contour encloses

$$\omega_k = -\mathrm{i}\nu_k. \tag{10.63}$$

Evaluating the Fourier factor at this pole gives

$$\mathrm{e}^{-\mathrm{i}\omega_k t} = \mathrm{e}^{-\nu_k t}, \tag{10.64}$$

and hence

$$\boxed{\gamma_k = \nu_k = \frac{2\pi k}{\beta}, \qquad k \geq 1.} \tag{10.65}$$

We must now calculate the corresponding residue.

At $\omega = -\mathrm{i}\nu_k$, the Drude spectral density is regular. The pole comes entirely from the hyperbolic cotangent. Therefore

$$\mathrm{Res}_{\omega=-\mathrm{i}\nu_k} F(\omega) = J_{\mathrm{D}}(-\mathrm{i}\nu_k)\,\mathrm{Res}_{\omega=-\mathrm{i}\nu_k}\coth\left(\frac{\beta\omega}{2}\right). \tag{10.66}$$

The constant $+1$ appearing in Eq. (10.44) is analytic and therefore contributes no residue at a Matsubara pole.

Residue of the thermal factor. The function $\coth z$ has residue unity at each of its simple poles:

$$\mathrm{Res}_{z=\mathrm{i}\pi k} \coth z = 1. \tag{10.67}$$

However, our variable is ω, not z. With

$$z = \frac{\beta\omega}{2}, \tag{10.68}$$

we have

$$z - z_k = \frac{\beta}{2}(\omega - \omega_k). \tag{10.69}$$

Near the pole,

$$\coth z \sim \frac{1}{z - z_k}, \tag{10.70}$$

and therefore

$$\coth\left(\frac{\beta\omega}{2}\right) \sim \frac{2}{\beta}\frac{1}{\omega - \omega_k}. \tag{10.71}$$

Hence

$$\boxed{\mathrm{Res}_{\omega=-\mathrm{i}\nu_k} \coth\left(\frac{\beta\omega}{2}\right) = \frac{2}{\beta}.} \tag{10.72}$$

This factor of $2/\beta$ is essential and comes from the change of variables between z and ω.

Spectral density evaluated at the Matsubara pole. We next evaluate

$$J_{\mathrm{D}}(-\mathrm{i}\nu_k) = \frac{2\lambda\gamma_{\mathrm{D}}(-\mathrm{i}\nu_k)}{(-\mathrm{i}\nu_k)^2 + \gamma_{\mathrm{D}}^2}. \tag{10.73}$$

Since

$$(-\mathrm{i}\nu_k)^2 = -\nu_k^2, \tag{10.74}$$

we obtain

$$\begin{aligned} J_{\mathrm{D}}(-\mathrm{i}\nu_k) &= \frac{-2\mathrm{i}\lambda\gamma_{\mathrm{D}}\nu_k}{\gamma_{\mathrm{D}}^2 - \nu_k^2} \\ &= \boxed{\frac{2\mathrm{i}\lambda\gamma_{\mathrm{D}}\nu_k}{\nu_k^2 - \gamma_{\mathrm{D}}^2}.} \end{aligned} \tag{10.75}$$

Combining Eqs. (10.72) and (10.75),

$$\begin{aligned} \mathrm{Res}_{\omega=-\mathrm{i}\nu_k} F(\omega) &= \frac{2\mathrm{i}\lambda\gamma_{\mathrm{D}}\nu_k}{\nu_k^2 - \gamma_{\mathrm{D}}^2}\frac{2}{\beta} \\ &= \boxed{\frac{4\mathrm{i}\lambda\gamma_{\mathrm{D}}\nu_k}{\beta\left(\nu_k^2 - \gamma_{\mathrm{D}}^2\right)}.} \end{aligned} \tag{10.76}$$

Finally, Eq. (10.46) gives

$$c_k = -\mathrm{i}\,\mathrm{Res}_{\omega=-\mathrm{i}\nu_k} F(\omega)$$

$$= -\mathrm{i}\frac{4\mathrm{i}\lambda\gamma_{\mathrm{D}}\nu_k}{\beta\left(\nu_k^2-\gamma_{\mathrm{D}}^2\right)}$$

$$= \boxed{\frac{4\lambda\gamma_{\mathrm{D}}}{\beta}\frac{\nu_k}{\nu_k^2-\gamma_{\mathrm{D}}^2}, \qquad k\geq 1.} \tag{10.77}$$

For real λ, γ_{D}, and β, this expression is real. Thus the Matsubara exponentials contribute only to the real part of the Drude correlation function.

How each Matsubara coefficient is obtained

For the kth thermal pole,

$$\boxed{\begin{aligned} \omega_k = -\mathrm{i}\nu_k \quad &\Rightarrow \quad \mathrm{e}^{-\nu_k t}, \\ \mathrm{Res}_{\omega_k}\coth\left(\frac{\beta\omega}{2}\right) &= \frac{2}{\beta}, \\ J_{\mathrm{D}}(\omega_k) &= \frac{2\mathrm{i}\lambda\gamma_{\mathrm{D}}\nu_k}{\nu_k^2-\gamma_{\mathrm{D}}^2}, \\ c_k &= -\mathrm{i}\,J_{\mathrm{D}}(\omega_k)\frac{2}{\beta}. \end{aligned}}$$

The final line immediately produces

$$c_k = \frac{4\lambda\gamma_{\mathrm{D}}}{\beta}\frac{\nu_k}{\nu_k^2-\gamma_{\mathrm{D}}^2}.$$

10.10 Collecting the Drude and Matsubara contributions

We have now derived both families of terms.

The pole of the Drude spectral density,

$$\omega_0 = -\mathrm{i}\gamma_{\mathrm{D}}, \tag{10.78}$$

produces

$$c_0\mathrm{e}^{-\gamma_{\mathrm{D}}t}, \tag{10.79}$$

with

$$c_0 = \lambda\gamma_{\mathrm{D}}\left[\cot\left(\frac{\beta\gamma_{\mathrm{D}}}{2}\right)-\mathrm{i}\right]. \tag{10.80}$$

The thermal poles,

$$\omega_k = -\mathrm{i}\nu_k, \qquad \nu_k = \frac{2\pi k}{\beta}, \qquad k\geq 1, \tag{10.81}$$

produce

$$c_k\mathrm{e}^{-\nu_k t}, \tag{10.82}$$

with

$$c_k = \frac{4\lambda\gamma_{\mathrm{D}}}{\beta}\frac{\nu_k}{\nu_k^2-\gamma_{\mathrm{D}}^2}. \tag{10.83}$$

Adding all residues therefore gives

$$\boxed{\begin{aligned} C(t) = \lambda\gamma_{\mathrm{D}} \left[\cot\left(\frac{\beta\gamma_{\mathrm{D}}}{2}\right) - \mathrm{i}\right] \mathrm{e}^{-\gamma_{\mathrm{D}} t} \\ + \frac{4\lambda\gamma_{\mathrm{D}}}{\beta} \sum_{k=1}^{\infty} \frac{\nu_k}{\nu_k^2 - \gamma_{\mathrm{D}}^2} \mathrm{e}^{-\nu_k t}, \qquad t > 0. \end{aligned}} \tag{10.84}$$

It is useful to see explicitly how every factor in this expression arose:

$$\boxed{\begin{aligned} -\mathrm{i}\gamma_{\mathrm{D}} \quad &\longrightarrow \quad \lambda\gamma_{\mathrm{D}} \left[\cot\left(\frac{\beta\gamma_{\mathrm{D}}}{2}\right) - \mathrm{i}\right] \mathrm{e}^{-\gamma_{\mathrm{D}} t}, \\ -\mathrm{i}\nu_k \quad &\longrightarrow \quad \frac{4\lambda\gamma_{\mathrm{D}}}{\beta} \frac{\nu_k}{\nu_k^2 - \gamma_{\mathrm{D}}^2} \mathrm{e}^{-\nu_k t}. \end{aligned}} \tag{10.85}$$

Thus the exponential decomposition used by HEOM is obtained directly from the residue theorem:

$$\boxed{\text{pole position} \longrightarrow \gamma_k, \qquad \text{residue} \longrightarrow c_k.} \tag{10.86}$$

For the Drude–Lorentz bath, all relevant poles lie on the negative imaginary axis, so all of the time-dependent factors are decaying exponentials. The Drude pole gives the mode with decay constant γ_{D}, while the thermal poles give the Matsubara modes with decay constants ν_k.

The result is therefore exactly of the form required by the hierarchy,

$$\boxed{C(t) = \sum_{k=0}^{\infty} c_k \mathrm{e}^{-\gamma_k t}.} \tag{10.87}$$

From residues to HEOM coefficients

There is no separate prescription for guessing the HEOM coefficients in the Drude–Lorentz case. They are obtained systematically from the complex-plane poles of the correlation-function integrand:

$$\boxed{c_p = -\mathrm{i}\ \mathrm{Res}_{\omega=\omega_p} \left\{ J_{\mathrm{D}}(\omega) \left[\coth\left(\frac{\beta\omega}{2}\right) + 1\right] \right\}.}$$

Once a pole is written as

$$\omega_p = -\mathrm{i}\gamma_p,$$

its contribution is automatically

$$c_p \mathrm{e}^{-\gamma_p t}.$$

This provides the direct mathematical bridge

$$J(\omega), T \longrightarrow \{\omega_p, \mathrm{Res}_p\} \longrightarrow \{\gamma_p, c_p\} \longrightarrow \text{HEOM}.$$

Technical remark. The formulas above assume that a Matsubara pole does not coincide with the Drude pole, i.e.,

$$\nu_k \neq \gamma_{\mathrm{D}}. \tag{10.88}$$

If these frequencies coincide exactly, the two simple poles merge and the separate formulas above become singular; the corresponding limiting expression must then be evaluated by treating the combined higher-order pole directly.

10.11 Truncating the Matsubara expansion

In a numerical calculation the infinite Matsubara sum must be truncated. Keeping K Matsubara terms gives

$$\boxed{C_K(t) = c_0 \mathrm{e}^{-\gamma_0 t} + \sum_{k=1}^{K} c_k \mathrm{e}^{-\gamma_k t}.} \tag{10.89}$$

The number of exponentials entering the hierarchy is therefore

$$\boxed{N_{\mathrm{exp}} = K + 1.} \tag{10.90}$$

The first term is the Drude mode and the remaining K terms are Matsubara modes.

This has an important computational consequence. Increasing K improves the representation of the bath correlation function but also increases the number of hierarchy directions.

For hierarchy depth L_{max},

$$N_{\mathrm{ADO}} = \binom{K + 1 + L_{\mathrm{max}}}{L_{\mathrm{max}}}. \tag{10.91}$$

Thus convergence of the correlation function and growth of the hierarchy are directly connected.

10.12 The backward-branch coefficients

For the Drude–Matsubara decomposition above, all decay rates

$$\gamma_k \tag{10.92}$$

are real. Therefore complex conjugation does not change the exponential basis,

$$\left(\mathrm{e}^{-\gamma_k t}\right)^* = \mathrm{e}^{-\gamma_k t}. \tag{10.93}$$

Consequently,

$$\boxed{\bar{c}_k = c_k^*} \tag{10.94}$$

for this particular decomposition.

Thus

$$\bar{c}_0 = \lambda\gamma_{\mathrm{D}} \left[\cot\left(\frac{\beta\gamma_{\mathrm{D}}}{2}\right) + \mathrm{i}\right], \tag{10.95}$$

$$\bar{c}_k = c_k, \qquad k \geq 1. \tag{10.96}$$

This is why the Python implementation can simply construct

```
cbar = np.conjugate(c)
```

for the present Drude–Matsubara representation.

This equality should not be assumed for an arbitrary complex exponential basis.

10.13 Constructing the coefficients in Python

The complete coefficient generation can be implemented as

```
def drude_matsubara_coefficients(
    lam,
    gamma_D,
    beta,
    K,
):
    """
    Drude-Lorentz correlation-function decomposition

        C(t) ~= sum_k c[k] exp(-gamma[k] t)

    using K Matsubara terms in addition to the
    Drude pole.

    Parameters
    ----------
    lam : float
        Reorganization energy.

    gamma_D : float
        Drude relaxation rate.

    beta : float
        Inverse temperature, beta = 1/(k_B T),
        in units consistent with H and gamma_D.

    K : int
        Number of Matsubara terms.

    Returns
    -------
    c : complex ndarray, shape (K+1,)
    cbar : complex ndarray, shape (K+1,)
    gamma : complex ndarray, shape (K+1,)
    """

    n_exp = K + 1

    c = np.zeros(
        n_exp,
        dtype=complex,
    )

    gamma = np.zeros(
        n_exp,
        dtype=complex,
    )

    # ------------------------------------------------
    # Drude pole: k = 0
    # ------------------------------------------------

    gamma[0] = gamma_D

    c[0] = (
        lam
        * gamma_D
        * (
            1.0
            / np.tan(
                0.5
                * beta
                * gamma_D
```

```
            )
            - 1j
        )
    )

    # ------------------------------------------------
    # Matsubara poles: k = 1,...,K
    # ------------------------------------------------

    for k in range(1, K + 1):

        nu_k = (
            2.0
            * np.pi
            * k
            / beta
        )

        gamma[k] = nu_k

        c[k] = (
            4.0
            * lam
            * gamma_D
            / beta
            * nu_k
            / (
                nu_k**2
                - gamma_D**2
            )
        )

    cbar = np.conjugate(c)

    return c, cbar, gamma
```

Listing 4: Complete coefficient generation.

The output of this function can be passed directly to the HEOM right-hand side constructed in the previous section.

10.14 Inspecting the exponential decomposition

Before performing the hierarchy propagation, it is useful to reconstruct the correlation function represented by the coefficients.

For

$$C_K(t) = \sum_{k=0}^{K} c_k \mathrm{e}^{-\gamma_k t}, \tag{10.97}$$

we can write

```
def correlation_from_exponentials(
    t,
    c,
    gamma,
):
    """
    Reconstruct C(t) from the exponential decomposition.
    """

    t = np.atleast_1d(t)

```

```
    C = np.zeros(
        len(t),
        dtype=complex,
    )

    for ck, gk in zip(c, gamma):

        C += (
            ck
            * np.exp(
                -gk * t
            )
        )

    return C
```

For example,

```
lam = 0.10
gamma_D = 1.0
beta = 1.0
K = 4

c, cbar, gamma = (
    drude_matsubara_coefficients(
        lam=lam,
        gamma_D=gamma_D,
        beta=beta,
        K=K,
    )
)

print("c =", c)
print("cbar =", cbar)
print("gamma =", gamma)
```

produces the complete bath information required by the HEOM solver.

10.15 Visualizing the bath correlation function

It is informative to plot the real and imaginary components separately.

```
times_corr = np.linspace(
    0.01,
    10.0,
    1000,
)

C_t = correlation_from_exponentials(
    times_corr,
    c,
    gamma,
)

plt.figure(figsize=(7, 4))

plt.plot(
    times_corr,
    C_t.real,
    label=r"$\mathrm{Re}\,C(t)$",
)

plt.plot(
    times_corr,
```

```
    C_t.imag,
    label=r"$\mathrm{Im}\,C(t)$",
)

plt.xlabel("Time")
plt.ylabel(r"$C(t)$")

plt.legend()
plt.tight_layout()
plt.show()
```

The real part generally contains a relatively rapid thermal contribution from the Matsubara modes, while the imaginary part of the present Drude decomposition comes entirely from the Drude pole,

$$\boxed{C_{\mathrm{I}}(t) = -\lambda\gamma_{\mathrm{D}} \mathrm{e}^{-\gamma_{\mathrm{D}} t}.} \tag{10.98}$$

The characteristic decay time of this part of the bath memory is therefore

$$\tau_{\mathrm{B}} = \gamma_{\mathrm{D}}^{-1}. \tag{10.99}$$

10.16 Checking convergence with the number of Matsubara terms

The Matsubara truncation K is an approximation and should be converged independently of the hierarchy depth $L_{\max}$.

For example, one can compare

$$K = 1,\ 2,\ 4,\ 8, \ldots \tag{10.100}$$

by reconstructing the corresponding correlation functions.

A simple implementation is

```
times_corr = np.linspace(
    0.01,
    5.0,
    1000,
)

plt.figure(figsize=(7, 4))

for K_test in [1, 2, 4, 8]:

    c_test, _, gamma_test = (
        drude_matsubara_coefficients(
            lam=lam,
            gamma_D=gamma_D,
            beta=beta,
            K=K_test,
        )
    )

    C_test = (
        correlation_from_exponentials(
            times_corr,
            c_test,
            gamma_test,
        )
    )

    plt.plot(
```

```
        times_corr,
        C_test.real,
        label=fr"$K={K_test}$",
    )

plt.xlabel("Time")
plt.ylabel(r"$\mathrm{Re}\,C_K(t)$")

plt.legend()
plt.tight_layout()
plt.show()
```

Listing 5: Complete setup for checking convergence with the number of Matsubara terms.

The short-time part of the real correlation function is generally the most sensitive to the Matsubara truncation.

One should therefore not simply choose K and assume the decomposition is adequate. The correlation function itself should first be converged over the time window relevant to the system dynamics.

Two independent convergence parameters

A Drude–Matsubara HEOM calculation has at least two conceptually different truncations:

$$\boxed{K \quad \text{and} \quad L_{\max}.}$$

Increasing K improves the representation

$$C(t) \simeq \sum_{k=0}^{K} c_k \mathrm{e}^{-\gamma_k t},$$

whereas increasing $L_{\max}$ improves the representation of the resulting hierarchy.
Convergence with respect to one does not guarantee convergence with respect to the other.

10.17 A subtlety at very short times

For the ideal Drude–Lorentz spectral density,

$$J_{\mathrm{D}}(\omega) \sim \frac{1}{\omega} \qquad (\omega \to \infty). \tag{10.101}$$

Consequently, the real part of the formal correlation function has particularly strong ultraviolet structure at $t = 0$, and convergence of the raw Matsubara representation exactly at $t = 0$ is slow.

For this reason, plots used to assess the Matsubara representation are often more informative for

$$t > 0 \tag{10.102}$$

rather than by focusing exclusively on the value at $t = 0$.

In practical HEOM calculations, Padé decompositions and residual corrections are commonly used to represent the thermal part of the correlation function more efficiently than a long Matsubara sum. These refinements do not change the hierarchy derived previously; they only provide a more economical set of exponential coefficients.

11 Using the physical bath in the HEOM solver

We can now replace the illustrative coefficients used in section 8 by coefficients derived from the physical bath.

Suppose that

```
lam = 0.10
gamma_D = 1.0
beta = 1.0
K = 4
```

in the same energy units used for $\hat{H}_S$.

Then

```
c, cbar, gamma = (
    drude_matsubara_coefficients(
        lam=lam,
        gamma_D=gamma_D,
        beta=beta,
        K=K,
    )
)
```

replaces the earlier manually chosen arrays.

The number of exponential modes is now

```
n_exp = len(gamma)
```

and the hierarchy is regenerated accordingly:

```
L_max = 4

multiindices = generate_multiindices(
    n_exp=n_exp,
    L_max=L_max,
)

index_of = {
    n: i
    for i, n in enumerate(multiindices)
}

n_ado = len(multiindices)

print(
    "Number of exponentials:",
    n_exp,
)

print(
    "Number of ADOs:",
    n_ado,
)
```

For $K = 4$, there are

$$N_{\mathrm{exp}} = 5 \tag{11.1}$$

hierarchy directions.

If $L_{\max} = 4$, the number of ADOs is

$$N_{\mathrm{ADO}} = \binom{5+4}{4} = 126. \tag{11.2}$$

For a qubit, each ADO is only a 2×2 matrix, so the total state propagated by the ODE solver contains

$$126 \times 4 = 504 \tag{11.3}$$

complex matrix elements.

This example already illustrates why the number of exponential terms must be kept as small as possible while maintaining an accurate representation of $C(t)$.

11.1 Complete physical qubit example

The earlier qubit model,

$$\hat{H}_{\mathrm{S}} = \frac{\epsilon}{2}\sigma_z + \frac{\Delta}{2}\sigma_x, \qquad \hat{V} = \sigma_z, \tag{11.4}$$

can now be coupled to an actual Drude bath.

The complete setup becomes

```
# ------------------------------------------------------------
# Qubit
# ------------------------------------------------------------

epsilon = 1.0
Delta = 1.0

H = (
    0.5 * epsilon * sz
    + 0.5 * Delta * sx
)

V = sz


# ------------------------------------------------------------
# Drude-Lorentz bath
# ------------------------------------------------------------

lam = 0.10
gamma_D = 1.0
beta = 1.0

K = 4

c, cbar, gamma = (
    drude_matsubara_coefficients(
        lam=lam,
        gamma_D=gamma_D,
        beta=beta,
        K=K,
    )
)


# ------------------------------------------------------------
# Hierarchy
# ------------------------------------------------------------

L_max = 4

multiindices = generate_multiindices(
    n_exp=len(gamma),
    L_max=L_max,
)
```

```

index_of = {
    n: i
    for i, n in enumerate(multiindices)
}

n_ado = len(multiindices)


# ------------------------------------------------------------
# Initial condition
# ------------------------------------------------------------

rho_initial = np.array(
    [[1.0, 0.0],
     [0.0, 0.0]],
    dtype=complex,
)

d = H.shape[0]

ados_initial = np.zeros(
    (n_ado, d, d),
    dtype=complex,
)

zero_index = (
    0,
) * len(gamma)

physical_index = index_of[
    zero_index
]

ados_initial[
    physical_index
] = rho_initial

y0 = ados_initial.reshape(-1)


# ------------------------------------------------------------
# Propagation
# ------------------------------------------------------------

times = np.linspace(
    0.0,
    20.0,
    1001,
)

solution = solve_ivp(
    fun=lambda t, y: heom_rhs(
        t,
        y,
        H,
        V,
        c,
        cbar,
        gamma,
        multiindices,
        index_of,
    ),
    t_span=(
        times[0],
        times[-1],
    ),
    y0=y0,
```

```
    t_eval=times,
    method="DOP853",
    rtol=1e-9,
    atol=1e-11,
)
```

Listing 6: Complete setup and propagation of the qubit HEOM for a Drude–Lorentz bath using a truncated Matsubara decomposition.

Nothing in the HEOM right-hand side had to be changed. Only the arrays

$$c, \qquad \bar{c}, \qquad \gamma \tag{11.5}$$

were replaced by physically derived values.

This separation is one of the useful features of the HEOM formulation: the hierarchy propagation machinery is independent of the particular method used to construct the exponential representation of the bath.

11.2 What the different bath parameters do

The Drude parameters have transparent physical effects.

Increasing λ. The parameter λ scales every coefficient c_k. Increasing λ therefore increases the strength of the system–bath coupling and generally makes environmental relaxation and decoherence stronger.

Increasing γ_{D}. Since

$$\tau_{\mathrm{B}} \sim \gamma_{\mathrm{D}}^{-1}, \tag{11.6}$$

a large γ_{D} corresponds to a rapidly relaxing bath with a short memory time.

A smaller γ_{D} produces a more slowly decaying bath correlation function and therefore stronger non-Markovian memory.

Changing the temperature. Temperature enters through

$$\beta = \frac{1}{k_{\mathrm{B}} T} \tag{11.7}$$

and affects both the Drude coefficient

$$\cot\left(\frac{\beta\gamma_{\mathrm{D}}}{2}\right) \tag{11.8}$$

and the Matsubara frequencies

$$\nu_k = \frac{2\pi k}{\beta}. \tag{11.9}$$

At lower temperature, β becomes larger and the Matsubara frequencies become more closely spaced. More thermal exponential terms are therefore typically required.

This explains an important computational fact:

> Low-temperature HEOM calculations are generally more demanding because the thermal bath correlation function requires a richer exponential representation.

11.3 The full convergence workflow

A practical HEOM calculation should now be performed in the following order.

1. Choose the physical parameters
$$\hat{H}_{\mathrm{S}}, \qquad \hat{V}, \qquad J(\omega), \qquad T.$$
2. Construct the bath correlation function.
3. Obtain an exponential representation
$$C(t) \simeq \sum_k c_k \mathrm{e}^{-\gamma_k t}.$$
4. Increase the number of exponential terms until the representation of $C(t)$ is sufficiently accurate.
5. For that decomposition, construct the HEOM hierarchy.
6. Increase $L_{\max}$ until the physical reduced dynamics is converged.
7. Tighten the ODE solver tolerance until numerical-integration errors are smaller than the desired physical accuracy.

Schematically,

$$\boxed{\begin{aligned} J(\omega), T &\longrightarrow C(t) \\ &\xrightarrow{K} \{c_k, \bar{c}_k, \gamma_k\} \\ &\xrightarrow{L_{\max}} \{\rho_{\boldsymbol{n}}(t)\} \\ &\xrightarrow{\text{ODE tolerance}} \rho_{\mathrm{S}}(t). \end{aligned}} \tag{11.10}$$

These are three logically distinct numerical convergence problems.

12 Padé spectrum decomposition

The Matsubara expansion derived above is particularly useful pedagogically because its origin is completely transparent. Padé spectrum decomposition provides a more compact alternative when many Matsubara modes would otherwise be required. The thermal factor

$$\coth\left(\frac{\beta\omega}{2}\right) \tag{12.1}$$

has poles at

$$\omega = \pm \mathrm{i}\nu_k, \qquad \nu_k = \frac{2\pi k}{\beta}, \qquad k = 1, 2, \ldots, \tag{12.2}$$

and retaining the first K poles gives the truncated Matsubara representation derived above.

The difficulty is that the Matsubara frequencies are fixed:

$$\nu_1, \nu_2, \nu_3, \ldots = \frac{2\pi}{\beta}, \frac{4\pi}{\beta}, \frac{6\pi}{\beta}, \ldots. \tag{12.3}$$

Truncating after K terms simply discards all poles with $k > K$. At low temperature, β is large and the Matsubara frequencies become closely spaced. Many terms may then be required to represent the short-time thermal correlation accurately.

This is especially expensive in HEOM because every retained exponential introduces another hierarchy direction. With N_{exp} exponential terms and hierarchy depth L_{max},

$$N_{\mathrm{ADO}} = \binom{N_{\mathrm{exp}} + L_{\mathrm{max}}}{L_{\mathrm{max}}}. \tag{12.4}$$

Reducing the number of exponentials can therefore produce a very large reduction in computational cost.

Padé spectrum decomposition addresses this problem differently. Instead of retaining the first poles of the exact Matsubara series, it approximates the thermal distribution by a rational function whose poles and residues are chosen collectively. The resulting poles are therefore *not* simply the first Matsubara frequencies. Both their positions and their weights depend on the chosen Padé order.

The advantage is that a relatively small number of Padé poles can reproduce the thermal function over a substantially larger useful frequency range than the same number of Matsubara poles [7–9].

12.1 What is actually being approximated?

Introduce the dimensionless variable

$$x = \beta\omega. \tag{12.5}$$

The Bose thermal function may be written as

$$f_{\mathrm{B}}(x) = \frac{1}{1 - \mathrm{e}^{-x}} = \frac{1}{2} + \frac{1}{2}\coth\left(\frac{x}{2}\right). \tag{12.6}$$

A finite Padé spectrum decomposition replaces this transcendental function by a rational sum-over-poles representation. In a commonly used bosonic Padé construction,

$$\boxed{f_{\mathrm{B}}(x) \simeq \frac{1}{2} + \frac{1}{x} + \sum_{j=1}^{N_{\mathrm{P}}} \frac{2\kappa_j x}{x^2 + \epsilon_j^2}.} \tag{12.7}$$

Equivalently,

$$\boxed{\coth\left(\frac{x}{2}\right) \simeq \frac{2}{x} + 4\sum_{j=1}^{N_{\mathrm{P}}} \frac{\kappa_j x}{x^2 + \epsilon_j^2}.} \tag{12.8}$$

Here

$$\epsilon_j > 0 \tag{12.9}$$

are the dimensionless Padé pole positions, and

$$\kappa_j \tag{12.10}$$

are their corresponding weights.

This equation makes the connection with the Matsubara expansion particularly clear. In the Matsubara decomposition the pole positions are fixed at

$$x = \pm 2\pi \mathrm{i} k. \tag{12.11}$$

In the Padé decomposition they are instead

$$\boxed{x = \pm \mathrm{i}\epsilon_j.} \tag{12.12}$$

Since $x = \beta\omega$, the poles in the physical frequency plane are

$$\boxed{\omega_j = -\mathrm{i}\frac{\epsilon_j}{\beta}} \tag{12.13}$$

for the lower-half-plane contour.

The corresponding Fourier factor is therefore

$$\mathrm{e}^{-\mathrm{i}\omega_j t} = \exp\left(-\frac{\epsilon_j}{\beta}t\right). \tag{12.14}$$

Thus each Padé pole generates an HEOM exponential with decay rate

$$\boxed{\gamma_j^{(\mathrm{P})} = \frac{\epsilon_j}{\beta}.} \tag{12.15}$$

This is the direct Padé analogue of

$$\gamma_k^{(\mathrm{M})} = \nu_k = \frac{2\pi k}{\beta} \tag{12.16}$$

in the Matsubara expansion.

Matsubara versus Padé poles

The two decompositions differ primarily in how the thermal poles are chosen:

Matsubara:	$\gamma_k = \dfrac{2\pi k}{\beta}$,	fixed equally spaced poles,
Padé:	$\gamma_j = \dfrac{\epsilon_j}{\beta}$,	order-dependent optimized poles.

Both ultimately produce the same object required by HEOM:

$$C(t) \simeq \sum_j c_j \mathrm{e}^{-\gamma_j t}.$$

12.2 From a Padé pole to a Drude correlation-function coefficient

We now determine the coefficients that must be passed to the HEOM solver.

Recall the Drude–Lorentz spectral density,

$$J_{\mathrm{D}}(\omega) = \frac{2\lambda\gamma_{\mathrm{D}}\omega}{\omega^2 + \gamma_{\mathrm{D}}^2}. \tag{12.17}$$

It is convenient to retain the contribution from the Drude pole itself exactly. Thus the first exponential remains

$$\gamma_0 = \gamma_{\mathrm{D}}, \tag{12.18}$$

with

$$\boxed{c_0 = \lambda\gamma_{\mathrm{D}}\left[\cot\left(\frac{\beta\gamma_{\mathrm{D}}}{2}\right) - \mathrm{i}\right].} \tag{12.19}$$

The Padé approximation is then used to obtain a compact representation of the remaining thermal contribution. Consider the jth Padé pole,

$$\omega_j = -\mathrm{i}\frac{\epsilon_j}{\beta} = -\mathrm{i}\gamma_j^{(\mathrm{P})}. \tag{12.20}$$

From Eq. (12.8), the term responsible for this pole is

$$4\kappa_j \frac{x}{x^2+\epsilon_j^2}, \qquad x = \beta\omega. \tag{12.21}$$

Near the lower pole

$$x_j = -\mathrm{i}\epsilon_j, \tag{12.22}$$

we have

$$\mathrm{Res}_{x=-\mathrm{i}\epsilon_j}\left[4\kappa_j \frac{x}{x^2+\epsilon_j^2}\right] = 4\kappa_j \frac{-\mathrm{i}\epsilon_j}{-2\mathrm{i}\epsilon_j} = 2\kappa_j. \tag{12.23}$$

Because $x = \beta\omega$,

$$x - x_j = \beta(\omega - \omega_j), \tag{12.24}$$

so the residue with respect to ω acquires an additional factor $1/\beta$:

$$\boxed{\mathrm{Res}_{\omega=\omega_j} \coth\left(\frac{\beta\omega}{2}\right) \simeq \frac{2\kappa_j}{\beta}.} \tag{12.25}$$

Compare this with the exact Matsubara residue,

$$\frac{2}{\beta}. \tag{12.26}$$

The Padé weight κ_j therefore modifies the strength carried by each effective thermal pole.

Next evaluate the Drude spectral density at

$$\omega_j = -\mathrm{i}\gamma_j^{(\mathrm{P})}. \tag{12.27}$$

Exactly as in the Matsubara derivation,

$$J_{\mathrm{D}}\left(-\mathrm{i}\gamma_j^{(\mathrm{P})}\right) = \frac{2\mathrm{i}\lambda\gamma_{\mathrm{D}}\gamma_j^{(\mathrm{P})}}{\left(\gamma_j^{(\mathrm{P})}\right)^2 - \gamma_{\mathrm{D}}^2}. \tag{12.28}$$

Using the residue theorem, the coefficient of the jth Padé exponential is therefore

$$\begin{aligned} c_j^{(\mathrm{P})} &= -\mathrm{i}J_{\mathrm{D}}\left(-\mathrm{i}\gamma_j^{(\mathrm{P})}\right)\frac{2\kappa_j}{\beta} \\ &= \boxed{\frac{4\lambda\gamma_{\mathrm{D}}\kappa_j}{\beta}\frac{\gamma_j^{(\mathrm{P})}}{\left(\gamma_j^{(\mathrm{P})}\right)^2 - \gamma_{\mathrm{D}}^2}.} \end{aligned} \tag{12.29}$$

Since

$$\gamma_j^{(\mathrm{P})} = \frac{\epsilon_j}{\beta}, \tag{12.30}$$

this may also be written as

$$\boxed{c_j^{(\mathrm{P})} = \frac{4\lambda\gamma_{\mathrm{D}}\kappa_j}{\beta}\frac{\epsilon_j/\beta}{(\epsilon_j/\beta)^2 - \gamma_{\mathrm{D}}^2}.} \tag{12.31}$$

For real bath parameters, these thermal Padé coefficients are real.

The resulting finite correlation-function representation is therefore

$$\boxed{C_{N_{\mathrm{P}}}^{(\mathrm{P})}(t) = c_0 \mathrm{e}^{-\gamma_{\mathrm{D}} t} + \sum_{j=1}^{N_{\mathrm{P}}} c_j^{(\mathrm{P})} \mathrm{e}^{-(\epsilon_j/\beta)t}.} \tag{12.32}$$

The only remaining question is how to calculate

$$\epsilon_j \qquad \text{and} \qquad \kappa_j. \tag{12.33}$$

12.3 How the Padé poles ϵ_j are actually calculated

A particularly convenient feature of Padé spectrum decomposition is that its poles can be generated numerically from the eigenvalues of a small real symmetric tridiagonal matrix [8].

For a Padé order $N_{\mathrm{P}} = N$, construct a $2N \times 2N$ symmetric matrix $\mathbf{A}$ with zero diagonal and nearest-neighbor elements

$$\boxed{A_{m,m+1} = A_{m+1,m} = \frac{1}{\sqrt{(2m+1)(2m+3)}}, \qquad m = 1, \ldots, 2N-1.} \tag{12.34}$$

Thus

$$\mathbf{A} = \begin{pmatrix} 0 & a_1 & 0 & \cdots \\ a_1 & 0 & a_2 & \cdots \\ 0 & a_2 & 0 & \ddots \\ \vdots & \vdots & \ddots & \ddots \end{pmatrix}, \tag{12.35}$$

where

$$a_m = \frac{1}{\sqrt{(2m+1)(2m+3)}}. \tag{12.36}$$

Its eigenvalues occur in positive–negative pairs. Let the N negative eigenvalues be

$$\alpha_1, \ldots, \alpha_N < 0. \tag{12.37}$$

The Padé poles are then

$$\boxed{\epsilon_j = -\frac{2}{\alpha_j}, \qquad j = 1, \ldots, N.} \tag{12.38}$$

Because $\alpha_j < 0$, every ϵ_j is positive.

This converts the abstract problem of locating the roots of a high-order Padé denominator into a stable symmetric-eigenvalue problem.

12.4 A second matrix needed for the Padé weights

The residues κ_j also depend on the zeros of the numerator of the rational approximation.

Construct a second symmetric matrix $\mathbf{B}$ of dimension $(2N-1)\times(2N-1)$ with

$$\boxed{B_{m,m+1} = B_{m+1,m} = \frac{1}{\sqrt{(2m+3)(2m+5)}}, \qquad m = 1, \ldots, 2N-2.} \tag{12.39}$$

Let its $N-1$ negative eigenvalues be

$$\beta_1, \ldots, \beta_{N-1} < 0. \tag{12.40}$$

Define

$$\boxed{\chi_m = -\frac{2}{\beta_m}, \qquad m = 1, \ldots, N-1.} \tag{12.41}$$

The ϵ_j are associated with the poles of the rational approximation, whereas the χ_m are associated with its numerator zeros.

12.5 Calculating the Padé weights κ_j

Once the two sets

$$\{\epsilon_j\}_{j=1}^{N} \qquad \text{and} \qquad \{\chi_m\}_{m=1}^{N-1} \tag{12.42}$$

have been obtained, the Padé weights follow from

$$\boxed{\kappa_j = \frac{N(2N+3)}{2} \frac{\displaystyle\prod_{m=1}^{N-1}\left(\chi_m^2 - \epsilon_j^2\right)}{\displaystyle\prod_{\substack{m=1 \\ m\neq j}}^{N}\left(\epsilon_m^2 - \epsilon_j^2\right)}, \qquad j = 1, \ldots, N.} \tag{12.43}$$

This expression is simply the partial-fraction residue of the Padé rational function written in terms of its poles and zeros.

The complete numerical construction is therefore

$$\boxed{N \longrightarrow \mathbf{A}, \mathbf{B} \longrightarrow \{\epsilon_j, \chi_j\} \longrightarrow \{\kappa_j\} \longrightarrow \{\gamma_j, c_j\}.} \tag{12.44}$$

Notice that none of these quantities has to be determined by nonlinear fitting.

12.6 Implementing the Padé poles and weights in Python

The preceding equations translate directly into a short numerical routine. The use of a symmetric eigensolver is preferable because both matrices are real and symmetric.

```python
import numpy as np
from scipy.linalg import eigvalsh


def pade_poles_weights(N):
    """
    Return the N dimensionless bosonic Pade poles epsilon_j
    and their weights kappa_j.
    """

    if N < 1:
        raise ValueError("N must be at least 1.")

    # ------------------------------------------------------------
    # Denominator matrix A: dimension 2N
    # ------------------------------------------------------------

    A = np.zeros((2 * N, 2 * N), dtype=float)

    for m in range(1, 2 * N):

        value = 1.0 / np.sqrt(
            (2 * m + 1) * (2 * m + 3)
        )

        A[m - 1, m] = value
        A[m, m - 1] = value

    eigenvalues_A = eigvalsh(A)

    # The spectrum is symmetric.  Take the N negative eigenvalues.
    negative_A = eigenvalues_A[:N]

    epsilon = -2.0 / negative_A


    # ------------------------------------------------------------
    # Numerator matrix B: dimension 2N - 1
    # ------------------------------------------------------------

    if N == 1:

        chi = np.empty(0, dtype=float)

    else:

        B = np.zeros(
            (2 * N - 1, 2 * N - 1),
            dtype=float,
        )

        for m in range(1, 2 * N - 1):

            value = 1.0 / np.sqrt(
                (2 * m + 3) * (2 * m + 5)
            )

            B[m - 1, m] = value
            B[m, m - 1] = value

        eigenvalues_B = eigvalsh(B)

        # B has N-1 negative eigenvalues.
        negative_B = eigenvalues_B[:N - 1]

        chi = -2.0 / negative_B

```

```
    # ------------------------------------------------------------
    # Pade weights kappa_j
    # ------------------------------------------------------------

    kappa = np.empty(N, dtype=float)

    prefactor = 0.5 * N * (2 * N + 3)

    for j in range(N):

        numerator = np.prod(
            chi**2 - epsilon[j]**2
        )

        denominator = np.prod([
            epsilon[m]**2 - epsilon[j]**2
            for m in range(N)
            if m != j
        ])

        kappa[j] = (
            prefactor
            * numerator
            / denominator
        )

    return epsilon, kappa
```

Listing 7: Construction of the dimensionless Padé poles ϵ_j and weights κ_j from symmetric eigenvalue problems.

For example,

```
epsilon_pade, kappa_pade = pade_poles_weights(N=4)

print("epsilon =", epsilon_pade)
print("kappa   =", kappa_pade)
```

returns the dimensionless pole locations and their weights.

The physical decay rates have not yet been constructed. Temperature enters only when we convert

$$\epsilon_j \longrightarrow \gamma_j^{(\mathrm{P})} = \frac{\epsilon_j}{\beta}. \tag{12.45}$$

12.7 Constructing the HEOM coefficients for a Drude–Lorentz bath

We can now combine the Padé pole calculation with Eq. (12.29).

The following routine returns arrays in exactly the same format as the Matsubara routine used previously:

$$\boxed{\texttt{c}, \quad \texttt{cbar}, \quad \texttt{gamma}.} \tag{12.46}$$

```
def drude_pade_coefficients(
    lam,
    gamma_D,
    beta,
    N,
):
    """
    Drude-Lorentz Pade decomposition
```

```

        C(t) ~= sum_k c[k] exp(-gamma[k] t)

    using N thermal Pade poles plus the exact Drude pole.
    """

    epsilon, kappa = pade_poles_weights(N)

    n_exp = N + 1

    c = np.zeros(
        n_exp,
        dtype=complex,
    )

    gamma = np.zeros(
        n_exp,
        dtype=complex,
    )


    # ------------------------------------------------------------
    # Exact Drude pole
    # ------------------------------------------------------------

    gamma[0] = gamma_D

    c[0] = (
        lam
        * gamma_D
        * (
            1.0
            / np.tan(
                0.5
                * beta
                * gamma_D
            )
            - 1j
        )
    )


    # ------------------------------------------------------------
    # Thermal Pade poles
    # ------------------------------------------------------------

    for j in range(N):

        gamma_j = (
            epsilon[j]
            / beta
        )

        gamma[j + 1] = gamma_j

        c[j + 1] = (
            4.0
            * lam
            * gamma_D
            * kappa[j]
            / beta
            * gamma_j
            / (
                gamma_j**2
                - gamma_D**2
            )
        )

```

```
77
78      # All decay rates are real in this representation,
79      # so complex conjugation gives the backward coefficients.
80
81      cbar = np.conjugate(c)
82
83      return c, cbar, gamma
```

Listing 8: Construction of the Drude–Lorentz Padé exponential coefficients used directly by the HEOM propagator.

The important point is that the output has exactly the same structure as the Matsubara coefficient generator. Therefore nothing in the HEOM right-hand side needs to be changed.

For example,

```
lam = 0.10
gamma_D = 1.0
beta = 1.0

N_pade = 4

c, cbar, gamma = drude_pade_coefficients(
    lam=lam,
    gamma_D=gamma_D,
    beta=beta,
    N=N_pade,
)
```

can replace

```
c, cbar, gamma = drude_matsubara_coefficients(
    lam=lam,
    gamma_D=gamma_D,
    beta=beta,
    K=K,
)
```

in the HEOM code developed previously.

Everything downstream remains unchanged:

$$\boxed{\{c_k, \bar{c}_k, \gamma_k\} \longrightarrow \texttt{generate\_multiindices} \longrightarrow \texttt{heom\_rhs} \longrightarrow \rho_{\mathrm{S}}(t).} \tag{12.47}$$

12.8 What has changed relative to Matsubara?

The distinction is now very concrete.

For Matsubara,

$$\gamma_k^{(\mathrm{M})} = \frac{2\pi k}{\beta} \tag{12.48}$$

and every thermal pole has unit residue weight in the thermal factor.

For Padé,

$$\gamma_j^{(\mathrm{P})} = \frac{\epsilon_j}{\beta}, \tag{12.49}$$

and every pole carries its own weight

$$\kappa_j. \tag{12.50}$$

Thus Padé does *not* merely replace

$$2\pi k \longrightarrow \epsilon_k. \tag{12.51}$$

Both the positions and the residues of the thermal poles are changed:

$$\boxed{\begin{array}{ccc} 2\pi k & \longrightarrow & \epsilon_j, \\ 1 & \longrightarrow & \kappa_j. \end{array}} \tag{12.52}$$

This additional freedom is what allows a small number of Padé poles to represent the thermal function much more efficiently.

12.9 Comparing Padé and Matsubara in practice

A useful numerical test is to construct two correlation functions with the same number of thermal modes,

$$C_N^{(\mathrm{M})}(t) \qquad \text{and} \qquad C_N^{(\mathrm{P})}(t), \tag{12.53}$$

and compare both with a high-accuracy reference.

For example,

```
N = 4

# Matsubara
c_M, _, gamma_M = drude_matsubara_coefficients(
    lam=lam,
    gamma_D=gamma_D,
    beta=beta,
    K=N,
)

# Pade
c_P, _, gamma_P = drude_pade_coefficients(
    lam=lam,
    gamma_D=gamma_D,
    beta=beta,
    N=N,
)

C_M = correlation_from_exponentials(
    times_corr,
    c_M,
    gamma_M,
)

C_P = correlation_from_exponentials(
    times_corr,
    c_P,
    gamma_P,
)
```

One may then plot

$$\mathrm{Re}\, C_N^{(\mathrm{M})}(t) \qquad \text{and} \qquad \mathrm{Re}\, C_N^{(\mathrm{P})}(t) \tag{12.54}$$

over the time interval relevant to the system dynamics.

The important comparison is not simply whether Padé and Matsubara agree with each other. Both are finite approximations. Instead, each should be compared against either

1. a much more highly converged exponential decomposition, or
2. a direct numerical evaluation of the defining frequency integral.

12.10 Padé convergence

The Padé order N is therefore a convergence parameter, just as the number of Matsubara terms K was previously.

A practical sequence is

$$N = 1,\ 2,\ 3,\ 4, \ldots \tag{12.55}$$

until the reconstructed correlation function and the resulting reduced dynamics cease to change within the desired tolerance.

There are consequently still two independent convergence problems:

N	$\longrightarrow$	accuracy of the Padé bath representation,
$L_{\max}$	$\longrightarrow$	accuracy of the HEOM hierarchy.

(12.56)

Padé decomposition does not remove the need for convergence testing. Its advantage is that the required value of N is often substantially smaller than the number of Matsubara terms required to achieve comparable accuracy.

12.11 Why this matters computationally

Suppose that a Matsubara calculation requires eight thermal modes while a Padé representation achieves comparable accuracy with four.

Including the Drude pole gives

$$N_{\mathrm{exp}}^{(\mathrm{M})} = 9, \qquad N_{\mathrm{exp}}^{(\mathrm{P})} = 5. \tag{12.57}$$

At hierarchy depth $L_{\max} = 4$, the corresponding numbers of ADOs would be

$$N_{\mathrm{ADO}}^{(\mathrm{M})} = \binom{9+4}{4} = 715, \tag{12.58}$$

$$N_{\mathrm{ADO}}^{(\mathrm{P})} = \binom{5+4}{4} = 126. \tag{12.59}$$

Thus reducing the number of bath exponentials by only four has reduced the hierarchy in this example by more than a factor of five.

This combinatorial amplification is the main reason that improving the correlation-function decomposition matters so much in HEOM.

12.12 Optional correction for the neglected fast modes

A finite Padé expansion still leaves a residual difference

$$\delta C(t) = C(t) - C_N^{(\mathrm{P})}(t). \tag{12.60}$$

When the omitted modes decay much faster than the relevant system time scale, their residual contribution can sometimes be approximated as effectively local in time,

$$\delta C(t) \simeq 2\Delta_N\, \delta(t). \tag{12.61}$$

This produces a local "terminator" correction to the system Liouvillian. Such corrections can improve a low-order Padé calculation, but they are not a substitute for convergence testing. For a first implementation it is pedagogically cleaner to omit the terminator, increase N, and verify convergence explicitly. A terminator can then be introduced as a numerical optimization once the uncorrected implementation is understood.

How Padé is actually implemented

The Padé decomposition does not require fitting the bath correlation function.
For a chosen order N:

$$\boxed{N \to \mathbf{A}, \mathbf{B} \to \epsilon_j, \chi_j \to \kappa_j \to \gamma_j = \frac{\epsilon_j}{\beta} \to c_j.}$$

The small symmetric matrices $\mathbf{A}$ and $\mathbf{B}$ determine the dimensionless Padé poles and weights. Temperature then rescales the poles into physical decay rates, and the spectral density determines their coefficients. For the Drude–Lorentz bath the final result is

$$\boxed{C_N^{(\mathrm{P})}(t) = c_0 \mathrm{e}^{-\gamma_\mathrm{D} t} + \sum_{j=1}^{N} c_j^{(\mathrm{P})} \mathrm{e}^{-(\epsilon_j/\beta)t},}$$

with

$$c_0 = \lambda \gamma_\mathrm{D} \left[\cot\left(\frac{\beta\gamma_\mathrm{D}}{2}\right) - \mathrm{i} \right]$$

and

$$c_j^{(\mathrm{P})} = \frac{4\lambda\gamma_\mathrm{D}\kappa_j}{\beta} \frac{\epsilon_j/\beta}{(\epsilon_j/\beta)^2 - \gamma_\mathrm{D}^2}.$$

Once these arrays have been constructed, the HEOM solver does not need to know whether the exponentials originated from a Matsubara or a Padé decomposition.

Technical remark. As in the Matsubara case, the simple expression for $c_j^{(\mathrm{P})}$ assumes

$$\frac{\epsilon_j}{\beta} \neq \gamma_\mathrm{D}. \tag{12.62}$$

If a Padé pole coincides with the Drude pole, the separate simple-pole expressions must be replaced by the appropriate limiting expression for the merged pole.

Crucially, nothing downstream changes:

$$\boxed{\text{Matsubara or Padé} \longrightarrow \{c_k, \bar{c}_k, \gamma_k\} \longrightarrow \text{the same HEOM solver.}} \tag{12.63}$$

This is why it is useful to design the numerical implementation around the generic coefficient arrays rather than hard-coding a particular bath decomposition.

The complete bridge from a physical bath to HEOM

A physical HEOM simulation does not begin by choosing arbitrary hierarchy coefficients. It begins with a spectral density and temperature:

$$J(\omega), T.$$

For the Drude–Lorentz bath,

$$J_{\mathrm{D}}(\omega) = \frac{2\lambda\gamma_{\mathrm{D}}\omega}{\omega^2 + \gamma_{\mathrm{D}}^2},$$

the thermal correlation function can be decomposed as

$$C(t) = \sum_k c_k \mathrm{e}^{-\gamma_k t}.$$

These coefficients are exactly the objects required by the hierarchy derived from the Feynman–Vernon influence functional.
The complete computational chain is therefore

$$\boxed{J(\omega), T \to C(t) \to \{c_k, \bar{c}_k, \gamma_k\} \to \{\rho_{\boldsymbol{n}}\} \to \rho_{\mathrm{S}}(t).}$$

At this point every element of a basic HEOM calculation has been constructed explicitly.

13 Complete Drude–Lorentz HEOM implementation

The following standalone program collects the physical-bath construction and HEOM propagation in one place. It is included for reproducibility after the analytical and implementation layers have been developed separately.

```
"""
HEOM_Drude_Lorentz.py

Complete HEOM example for a qubit coupled through sigma_z to a
Drude-Lorentz harmonic bath,

    J_D(omega) = 2 lambda gamma_D omega / (omega^2 + gamma_D^2),

using a truncated Matsubara decomposition

    C(t) ~= sum_k c_k exp(-gamma_k t).

Units: hbar = 1.  H, lambda, gamma_D, beta^{-1}, and all frequencies must
be expressed in mutually consistent units.
"""

import numpy as np
import matplotlib.pyplot as plt
from itertools import product
from scipy.integrate import solve_ivp


def generate_multiindices(n_exp, L_max):
    """Return all non-negative multi-indices with total tier <= L_max."""
    indices = [
        n
        for n in product(range(L_max + 1), repeat=n_exp)
        if sum(n) <= L_max
    ]
```

```
    indices.sort(key=lambda n: (sum(n), n))
    return indices


def drude_matsubara_coefficients(lam, gamma_D, beta, K):
    r"""
    Construct the Drude-Matsubara decomposition

        C(t) ~= sum_{k=0}^K c_k exp(-gamma_k t)

    for the convention

        C(t) = (1/pi) int_0^infty d omega J(omega)
               [coth(beta omega / 2) cos(omega t)
                - i sin(omega t)]

    with

        J(omega) = 2 lam gamma_D omega / (omega^2 + gamma_D^2).

    The coefficients are

        gamma_0 = gamma_D
        c_0 = lam gamma_D [cot(beta gamma_D / 2) - i]

    and, for k >= 1,

        nu_k = 2 pi k / beta
        gamma_k = nu_k
        c_k = (4 lam gamma_D / beta)
              nu_k / (nu_k^2 - gamma_D^2).

    Because all gamma_k are real in this representation,
    cbar_k = c_k^*.
    """
    if lam < 0:
        raise ValueError("lam must be non-negative.")
    if gamma_D <= 0:
        raise ValueError("gamma_D must be positive.")
    if beta <= 0:
        raise ValueError("beta must be positive.")
    if K < 0:
        raise ValueError("K must be a non-negative integer.")

    n_exp = K + 1

    c = np.zeros(n_exp, dtype=complex)
    gamma = np.zeros(n_exp, dtype=complex)

    x = 0.5 * beta * gamma_D
    if np.isclose(np.sin(x), 0.0, atol=1e-12):
        raise ValueError(
            "beta*gamma_D/2 is too close to a pole of cot(x); "
            "choose parameters away from this singular representation."
        )

    # Drude pole
    gamma[0] = gamma_D
    c[0] = lam * gamma_D * (1.0 / np.tan(x) - 1j)

    # Matsubara poles
    for k in range(1, K + 1):
        nu_k = 2.0 * np.pi * k / beta

        if np.isclose(nu_k**2, gamma_D**2, rtol=1e-12, atol=1e-14):
            raise ValueError(
                f"Matsubara frequency nu_{k} is too close to gamma_D; "
                "the simple coefficient formula becomes singular."
```

```
            )

        gamma[k] = nu_k
        c[k] = (
            4.0
            * lam
            * gamma_D
            / beta
            * nu_k
            / (nu_k**2 - gamma_D**2)
        )

    cbar = np.conjugate(c)
    return c, cbar, gamma


def correlation_from_exponentials(t, c, gamma):
    """Reconstruct C(t) from its finite exponential representation."""
    t = np.atleast_1d(t).astype(float)

    return np.sum(
        c[:, None] * np.exp(-gamma[:, None] * t[None, :]),
        axis=0,
    )


def heom_rhs(t, y, H, V, c, cbar, gamma, multiindices, index_of):
    """Right-hand side of the unscaled HEOM with a hard tier cutoff."""
    del t

    d = H.shape[0]
    n_ado = len(multiindices)
    n_exp = len(gamma)

    ados = y.reshape((n_ado, d, d))
    dados = np.zeros_like(ados)

    for a, n in enumerate(multiindices):
        rho = ados[a]

        # System Liouvillian
        drho = -1j * (H @ rho - rho @ H)

        # Hierarchy damping
        decay = sum(n[k] * gamma[k] for k in range(n_exp))
        drho -= decay * rho

        for k in range(n_exp):
            # Upward coupling
            n_up = list(n)
            n_up[k] += 1
            n_up = tuple(n_up)

            a_up = index_of.get(n_up)
            if a_up is not None:
                rho_up = ados[a_up]
                drho += -1j * (V @ rho_up - rho_up @ V)

            # Downward coupling
            if n[k] > 0:
                n_down = list(n)
                n_down[k] -= 1
                n_down = tuple(n_down)

                rho_down = ados[index_of[n_down]]

                drho += -1j * n[k] * (
                    c[k] * (V @ rho_down)
```

```
                    - cbar[k] * (rho_down @ V)
                )

            dados[a] = drho

    return dados.reshape(-1)


def extract_physical_density(solution, n_ado, d, physical_index):
    """Extract rho_S(t)=rho_0(t) from the hierarchy solution."""
    rho_t = np.empty((len(solution.t), d, d), dtype=complex)

    for j in range(len(solution.t)):
        hierarchy_t = solution.y[:, j].reshape((n_ado, d, d))
        rho_t[j] = hierarchy_t[physical_index]

    return rho_t


def main():
    # ------------------------------------------------------------
    # Qubit
    # ------------------------------------------------------------
    sx = np.array(
        [[0.0, 1.0],
         [1.0, 0.0]],
        dtype=complex,
    )

    sz = np.array(
        [[1.0,  0.0],
         [0.0, -1.0]],
        dtype=complex,
    )

    epsilon = 1.0
    Delta = 1.0

    H = 0.5 * epsilon * sz + 0.5 * Delta * sx
    V = sz

    # ------------------------------------------------------------
    # Drude-Lorentz bath
    # ------------------------------------------------------------
    lam = 0.10
    gamma_D = 1.0
    beta = 1.0

    # Number of Matsubara terms retained in the actual HEOM propagation.
    # Increase this value as part of the bath-decomposition convergence test.
    K = 2

    c, cbar, gamma = drude_matsubara_coefficients(
        lam=lam,
        gamma_D=gamma_D,
        beta=beta,
        K=K,
    )

    print("c =", c)
    print("cbar =", cbar)
    print("gamma =", gamma)

    # ------------------------------------------------------------
    # Inspect finite exponential representation of C(t)
    # ------------------------------------------------------------
    times_corr = np.linspace(0.01, 10.0, 1000)

```

```
    C_t = correlation_from_exponentials(
        times_corr,
        c,
        gamma,
    )

    plt.figure(figsize=(7, 4))
    plt.plot(times_corr, C_t.real, label=r"$\mathrm{Re}\,C(t)$")
    plt.plot(times_corr, C_t.imag, label=r"$\mathrm{Im}\,C(t)$")
    plt.xlabel("Time")
    plt.ylabel(r"$C(t)$")
    plt.legend()
    plt.tight_layout()
    plt.show()

    # ------------------------------------------------------------
    # Demonstrate Matsubara convergence of Re C(t)
    # ------------------------------------------------------------
    plt.figure(figsize=(7, 4))

    for K_test in [1, 2, 4, 8]:
        c_test, _, gamma_test = drude_matsubara_coefficients(
            lam=lam,
            gamma_D=gamma_D,
            beta=beta,
            K=K_test,
        )

        C_test = correlation_from_exponentials(
            times_corr,
            c_test,
            gamma_test,
        )

        plt.plot(
            times_corr,
            C_test.real,
            label=fr"$K={K_test}$",
        )

    plt.xlabel("Time")
    plt.ylabel(r"$\mathrm{Re}\,C_K(t)$")
    plt.legend()
    plt.tight_layout()
    plt.show()

    # ------------------------------------------------------------
    # Construct hierarchy
    # ------------------------------------------------------------
    L_max = 3

    multiindices = generate_multiindices(
        n_exp=len(gamma),
        L_max=L_max,
    )

    index_of = {n: i for i, n in enumerate(multiindices)}
    n_ado = len(multiindices)

    print("Number of exponential modes:", len(gamma))
    print("Hierarchy depth L_max:", L_max)
    print("Number of ADOs:", n_ado)

    # ------------------------------------------------------------
    # Factorized initial condition
    # ------------------------------------------------------------
    rho_initial = np.array(
        [[1.0, 0.0],
```

```
         [0.0, 0.0]],
        dtype=complex,
    )

    d = H.shape[0]

    ados_initial = np.zeros(
        (n_ado, d, d),
        dtype=complex,
    )

    zero_index = (0,) * len(gamma)
    physical_index = index_of[zero_index]

    ados_initial[physical_index] = rho_initial
    y0 = ados_initial.reshape(-1)

    # ------------------------------------------------------------
    # HEOM propagation
    # ------------------------------------------------------------
    times = np.linspace(0.0, 20.0, 501)

    solution = solve_ivp(
        fun=lambda t, y: heom_rhs(
            t,
            y,
            H,
            V,
            c,
            cbar,
            gamma,
            multiindices,
            index_of,
        ),
        t_span=(times[0], times[-1]),
        y0=y0,
        t_eval=times,
        method="DOP853",
        rtol=1e-8,
        atol=1e-10,
    )

    if not solution.success:
        raise RuntimeError(solution.message)

    # ------------------------------------------------------------
    # Physical reduced density matrix
    # ------------------------------------------------------------
    rho_t = extract_physical_density(
        solution,
        n_ado=n_ado,
        d=d,
        physical_index=physical_index,
    )

    P0 = rho_t[:, 0, 0].real
    P1 = rho_t[:, 1, 1].real

    sigma_z_t = np.array(
        [np.trace(sz @ rho).real for rho in rho_t]
    )

    # ------------------------------------------------------------
    # Diagnostics
    # ------------------------------------------------------------
    trace_error = max(
        abs(np.trace(rho) - 1.0)
        for rho in rho_t
```

```
    )

    hermiticity_error = max(
        np.linalg.norm(rho - rho.conj().T)
        for rho in rho_t
    )

    population_error = np.max(np.abs(P0 + P1 - 1.0))

    print("Maximum trace error:", trace_error)
    print("Maximum Hermiticity error:", hermiticity_error)
    print("Maximum population-sum error:", population_error)

    # ------------------------------------------------------------
    # Reduced dynamics
    # ------------------------------------------------------------
    plt.figure(figsize=(7, 4))
    plt.plot(solution.t, P0, label=r"$\rho_{00}$")
    plt.plot(solution.t, P1, label=r"$\rho_{11}$")
    plt.xlabel("Time")
    plt.ylabel("Population")
    plt.legend()
    plt.tight_layout()
    plt.show()

    plt.figure(figsize=(7, 4))
    plt.plot(
        solution.t,
        sigma_z_t,
        label=r"$\langle\sigma_z(t)\rangle$",
    )
    plt.xlabel("Time")
    plt.ylabel(r"$\langle\sigma_z\rangle$")
    plt.legend()
    plt.tight_layout()
    plt.show()


if __name__ == "__main__":
    main()
```

Listing 9: Complete Python implementation of the HEOM for a qubit coupled to a Drude–Lorentz harmonic bath. The script constructs the Drude–Matsubara exponential decomposition of the bath correlation function, examines convergence of the correlation function with the number of Matsubara terms, generates the hierarchy of auxiliary density operators, propagates the HEOM, and extracts the reduced-system dynamics.

Part V

QUAPI/TEMPO and an independent numerical benchmark

Part V orientation

Common starting point: the same Feynman–Vernon bath memory used by HEOM.
Different coordinates: HEOM stores exponential-mode amplitudes in ADOs, whereas QUAPI retains a finite sequence of discrete forward–backward path pairs and TEMPO compresses that history as an MPS.
Convergence principle: converge each method internally before comparing their reduced trajectories.

14 HEOM, QUAPI/TEMPO: two representations of environmental memory

It should now be clear that after the bath has been traced out, the physical reduced state alone is not sufficient to advance the dynamics: information about the past must also be retained. HEOM and QUAPI solve this same problem by enlarging the propagated state in different coordinates.

For HEOM one propagates

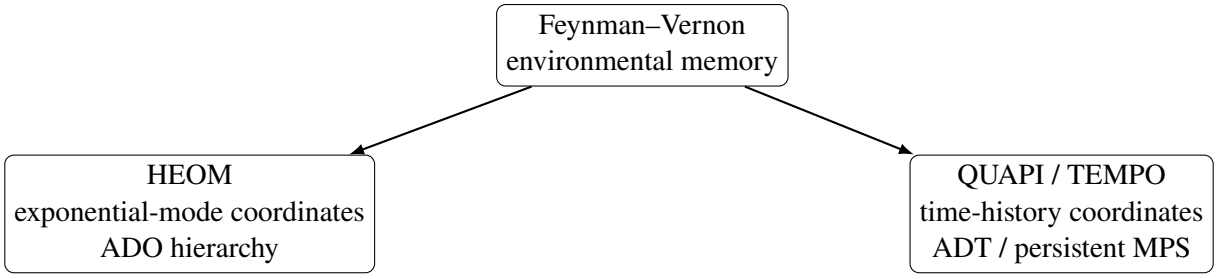
$$\Xi_{\mathrm{HEOM}}(t) = \{\rho_{\mathbf{0}}(t), \rho_{\boldsymbol{e}_1}(t), \rho_{\boldsymbol{e}_2}(t), \ldots\}, \tag{14.1}$$

where the auxiliary coordinates are weighted integrals of the past associated with exponential modes of $C(t)$. The enlarged state obeys a local ODE.

QUAPI instead retains an explicit recent forward–backward history,

$$\Xi^{(n)}_{\mathrm{QUAPI}} = A_n(a_n, a_{n-1}, \ldots, a_{n-K}), \qquad a_j = (s^+_j, s^-_j), \tag{14.2}$$

so that the newest time slice can interact with the previous K slices through the discretized influence functional. TEMPO changes the representation, not the influence-functional physics: it factorizes the augmented history tensor as an MPS and truncates weak temporal correlations by SVD.



There is generally no one-to-one correspondence between a particular ADO and a particular QUAPI time slice. A first-tier ADO contains a weighted integral over all earlier times for one exponential mode, whereas a QUAPI index refers to one definite discrete time. The two methods are therefore best viewed as mode-domain and time-domain coordinate systems for the same bath-induced memory. Their agreement after independent convergence is a stringent cross-check precisely because their numerical approximations are so different.

Meaning of "time-local embedding"

The enlarged HEOM or QUAPI/TEMPO state can be advanced from its current value without storing anything else. This does *not* mean that $\hat{\rho}_{\mathrm{S}}(t)$ itself has become Markovian; information can flow into the auxiliary memory coordinates and later return to the system.

15 From the continuous influence functional to discrete QUAPI memory

QUAPI discretizes the forward and backward system paths at times $t_k = k\,dt$ and combines the two branch values into one path-pair index

$$a_k = (s^+_k, s^-_k). \tag{15.1}$$

For uniform time cells, the continuum bath correlation function enters through integrated influence coefficients

$$\eta_0 = \int_0^{dt} \mathrm{d}x\,(dt - x)C(x), \tag{15.2}$$

$$\eta_j = \int_{-dt}^{dt} \mathrm{d}x\, (dt - |x|) C(j\,dt + x), \qquad j \geq 1. \tag{15.3}$$

The index $j = k - k'$ is therefore a *time separation*: η_j couples the newest path-pair variable to the variable j slices in the past. The discretized influence functional has a lower-triangular pair structure because a time slice interacts only with itself and earlier slices.

A finite-memory approximation retains only

$$0 \leq j \leq K, \qquad \tau_{\mathrm{mem}} = K\,dt, \tag{15.4}$$

so the retained influence factors form a diagonal band of width K in the (k, k') plane. This is the central QUAPI approximation: interactions with older slices are discarded after their effect has decayed sufficiently.

Why Kdt, not K, is the physical memory scale

If dt is halved while K is kept fixed, the physical memory time is also halved. A clean Trotter-step convergence test must therefore increase K as dt decreases so that $\tau_{\mathrm{mem}} = Kdt$ remains fixed.

16 Numerical case study: an Ohmic spin–boson qubit

The following numerical example is deliberately organized around questions rather than around code order. The notebook never uses HEOM to choose QUAPI parameters. QUAPI is first converged internally; only then is HEOM introduced as an external benchmark. This prevents a method-to-method difference from being mistaken for a convergence criterion.

16.1 Question 1: what model and numerical controls are being tested?

The system Hamiltonian and coupling operator are

$$\hat{H}_{\mathrm{S}} = \frac{\epsilon}{2}\sigma_z + \frac{\Delta}{2}\sigma_x, \qquad \hat{V} = \sigma_z, \tag{16.1}$$

with factorized initial state

$$\hat{\rho}_{\mathrm{S}}(0) = |{+x}\rangle\langle{+x}| = \frac{1}{2}\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}. \tag{16.2}$$

The exponentially cut off Ohmic spectral density is

$$J(\omega) = 2\pi\alpha_{\mathrm{SB}}\,\omega \mathrm{e}^{-\omega/\omega_c}. \tag{16.3}$$

For this qubit, $\hat{V}^2 = \mathbb{I}$, so the usual quadratic counterterm is proportional to the identity and its forward and backward phases cancel from the reduced dynamics.

16.2 Question 2: how long does the bath remember?

The notebook evaluates the same correlation function as equation (5.6) and constructs the cell-integrated coefficients η_j defined in equation (15.3). Before any propagation, the first diagnostic is therefore the decay of $C(t)$ itself, shown in figure 1. The code evaluates $C(t)$ on a fine frequency grid and reuses one fine time grid when constructing all η_j for a given (dt, K).

Table 2: Parameters used by `QUAPI_HEOM_convergence(1).ipynb`.

Quantity	Symbol / variable	Value
Qubit bias	ϵ	1.0
Tunneling	Δ	1.0
Ohmic coupling	α_{SB}	0.08
Cutoff frequency	ω_c	5.0
Temperature	T	0.5
Final time	t_{final}	8.0
Memory-test step	dt_{mem}	0.10
Memory candidates	K	10, 20, 30, 40
Memory tolerance	ε_K	10^{-3}
Time-step candidates	dt	0.20, 0.10, 0.05
Time-step tolerance	ε_{dt}	2×10^{-2}
MPS relative cutoff	$\varepsilon_{\mathrm{SVD}}$	3×10^{-6}
Maximum MPS bond	$\chi_{\max}$	24
HEOM exponentials	$N_{\exp}$	12
HEOM tiers	$L_{\max}$	3, 4

16.3 Question 3: how is the QUAPI history propagated without a dense ADT?

As described in section 15, each temporal physical index is the four-state pair $a_k = (s_k^+, s_k^-)$ with $s_k^\pm = \pm 1$. When a new slice is added, the complete retained "star" of pairwise bath couplings contributes

$$\prod_{j=0}^{\min(K,k)} \exp\left[-(s_k^+ - s_k^-)\left(\eta_j s_{k-j}^+ - \eta_j^* s_{k-j}^-\right)\right]. \tag{16.4}$$

The persistent implementation stores the augmented density tensor only as an MPS. The new four-state label is carried through the temporary virtual bonds while the entire "star" of influence factors is applied. A right-canonical QR sweep is followed by left-to-right SVD truncation. If the history already contains the newest site plus K older sites, the oldest physical index is summed directly on the final MPS core and absorbed into its neighbor. The dense history tensor is never reconstructed during the convergence sweeps.

Why this matters for the dt test

At fixed physical memory time $\tau_{\mathrm{mem}} = Kdt$, halving dt doubles the required K. A dense augmented tensor would therefore become exponentially more expensive exactly when the Trotter grid is refined. The persistent MPS representation makes the fixed-τ_{mem} test practical.

16.4 Question 4: is the finite memory converged?

At fixed $dt = 0.1$ the notebook increases K and uses the purely internal QUAPI metric

$$\varepsilon_K = \max_{t,i,j} |\rho_{ij}^{(K)}(t) - \rho_{ij}^{(K_{\mathrm{prev}})}(t)|. \tag{16.5}$$

HEOM does not enter this test. The first K for which $\varepsilon_K < 10^{-3}$ is selected.

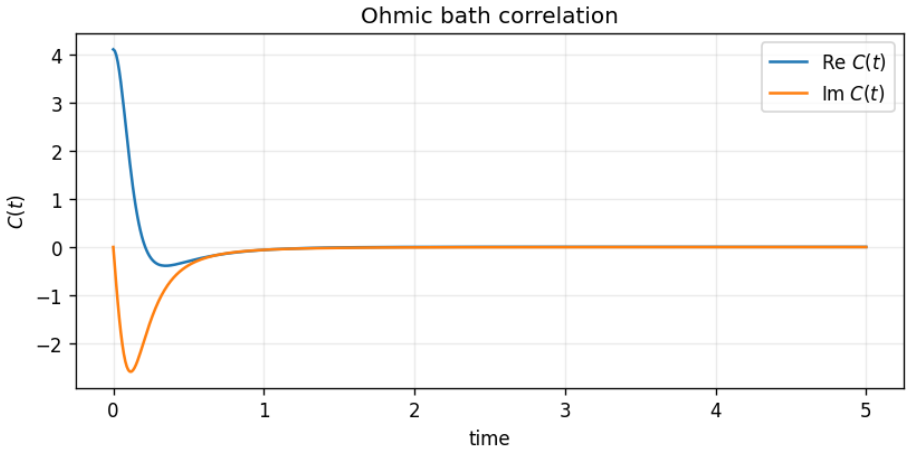


Figure 1: Ohmic bath correlation function evaluated by the notebook. The decay of $C(t)$ sets the physical scale that the QUAPI memory window must cover.

Table 3: Stage 1: QUAPI memory convergence at fixed $dt = 0.10$.

K	Kdt	successive change	$\lvert C(Kdt)\rvert/\lvert C(0)\rvert$	max bond	max discarded norm
10	1.0	–	2.0715×10^{-2}	16	6×10^{-6}
20	2.0	2.3481×10^{-2}	1.906×10^{-3}	18	8×10^{-6}
30	3.0	2.598×10^{-3}	8.30×10^{-4}	18	9×10^{-6}
40	4.0	7.57×10^{-4}	4.84×10^{-4}	18	7×10^{-6}

The memory sweep therefore freezes

$$\boxed{\tau_{\mathrm{mem}} = 4} \tag{16.6}$$

for the next stage. The maximum MPS bond remains 18, below the cap of 24, while the discarded norm stays of order 10^{-5} or smaller.

16.5 Question 5: is the Trotter step converged at the same physical memory time?

The second sweep changes dt while enforcing

$$K(dt) = \left\lceil \frac{\tau_{\mathrm{mem}}}{dt} \right\rceil, \tag{16.7}$$

so that reducing the time step never shortens the bath-memory window. The internal convergence metric is

$$\varepsilon_{dt} = \max_{t,i,j} |\rho_{ij}^{(dt)}(t) - \rho_{ij}^{(dt/2)}(t)|, \tag{16.8}$$

measured on the coarser nested grid.

Because Hermiticity gives identical changes for ρ_{10} and ρ_{01} and identical population changes for ρ_{11} and ρ_{00}, only two element columns are shown in the compact table; the notebook reports all four.

Thus the notebook selects

$$\boxed{dt = 0.05, \qquad K = 80, \qquad Kdt = 4.} \tag{16.9}$$

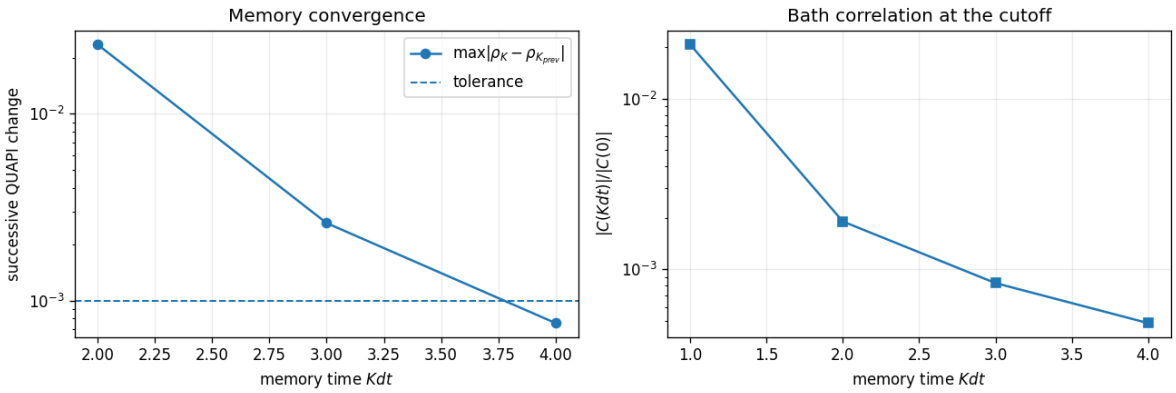


Figure 2: Stage 1 of the notebook. Left: successive QUAPI trajectory change as the physical memory time is increased. Right: the bath-correlation magnitude at the cutoff. The first tested point below the requested 10^{-3} trajectory tolerance is $K = 40$, corresponding to $\tau_{\mathrm{mem}} = Kdt = 4$.

Table 4: Stage 2: QUAPI time-step convergence at fixed $\tau_{\mathrm{mem}} = 4$.

dt	K	successive change	$\Delta\rho_{00}$	$\Delta\rho_{01}$	max bond
0.20	20	–	–	–	18
0.10	40	1.1759×10^{-2}	1.1759×10^{-2}	7.614×10^{-3}	18
0.05	80	1.5074×10^{-2}	6.038×10^{-3}	1.5074×10^{-2}	14

The important point is not that $dt = 0.05$ is universally sufficient; it is sufficient only for the stated demonstration tolerance. A stricter production calculation should append another nested step, for example $dt = 0.025$ with $K = 160$, and repeat the same internal comparison.

16.6 Question 6: how accurately does HEOM represent the same bath?

Only after equation (16.9) has been fixed does the notebook construct HEOM. The same numerically evaluated $C(t)$ is fitted as

$$C(t) \simeq \sum_{k=1}^{12} c_k \mathrm{e}^{-\gamma_k t}, \tag{16.10}$$

using positive logarithmically spaced γ_k and weighted ridge-regularized least squares for the complex amplitudes. Because the fitted rates are real, the notebook uses $\bar{c}_k = c_k^*$ in the downward coupling. The fit has relative L^2 error 1.326×10^{-2} and maximum absolute error 1.142×10^{-1}.

The hierarchy contains 455 ADOs through $L_{\max} = 3$ and 1820 through $L_{\max} = 4$. Classical RK4 is used with eight internal substeps per reported $dt = 0.05$ interval. The maximum change in any reduced-density-matrix element between the two hierarchy depths is

$$\varepsilon_{\mathrm{tier}} = 3.182 \times 10^{-3}. \tag{16.11}$$

16.7 Question 7: do the independently converged trajectories agree?

The final comparison uses all four density-matrix elements rather than only a population. The real parts are shown in figure 5, the imaginary parts in figure 6, and the element-resolved absolute differences in figure 7.

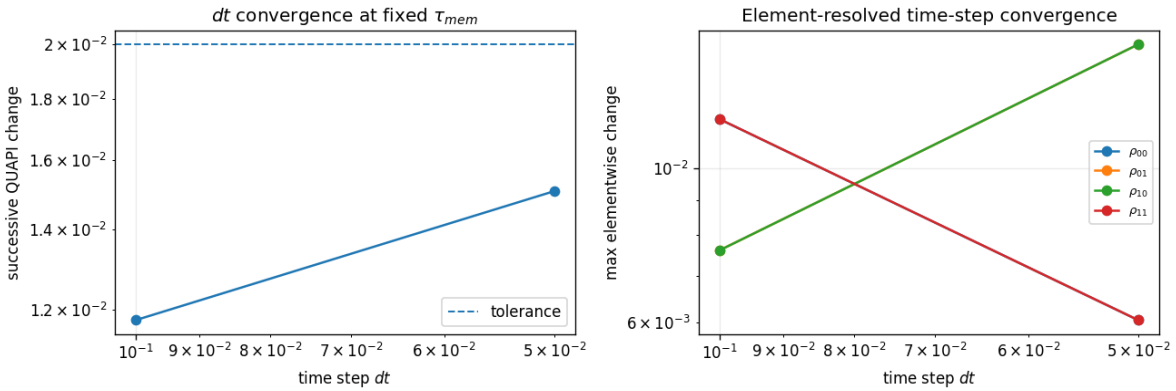


Figure 3: Stage 2 of the notebook. The time step is refined from 0.20 to 0.05 while K is increased from 20 to 80 so that $Kdt = 4$ throughout. The final successive change is 1.507×10^{-2}, below the demonstration tolerance 2×10^{-2}.

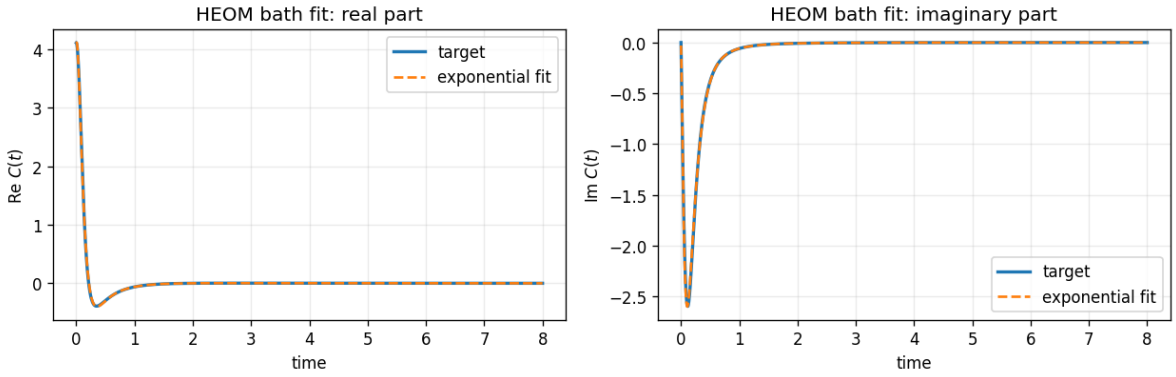


Figure 4: Real and imaginary parts of the numerically evaluated Ohmic correlation function and the 12-exponential representation used by HEOM. The fit error is a separate convergence variable from the HEOM hierarchy depth.

The maximum discrepancy, 2.9335×10^{-2}, occurs in the transient coherence. Both methods preserve trace to approximately machine precision and are Hermitian to the reported precision.

16.8 Question 8: what does the numerical error budget actually establish?

The calculation exposes five distinct internal scales:

1. finite QUAPI memory: final successive-K change 7.57×10^{-4};
2. QUAPI time discretization at fixed memory: final successive-dt change 1.507×10^{-2};
3. MPS compression: discarded norms of order 10^{-5} with the bond cap inactive;
4. HEOM bath representation: relative correlation-fit error 1.326×10^{-2};
5. HEOM hierarchy truncation: $L = 3 \to 4$ trajectory change 3.182×10^{-3}.

These diagnostics answer different questions and should not be collapsed into a single label such as "converged." In particular, the finite-memory error is already far below the final QUAPI–HEOM difference, so increasing K further at $dt = 0.05$ is not the efficient next test. The time-step and bath-fit scales are larger and should be tightened next.

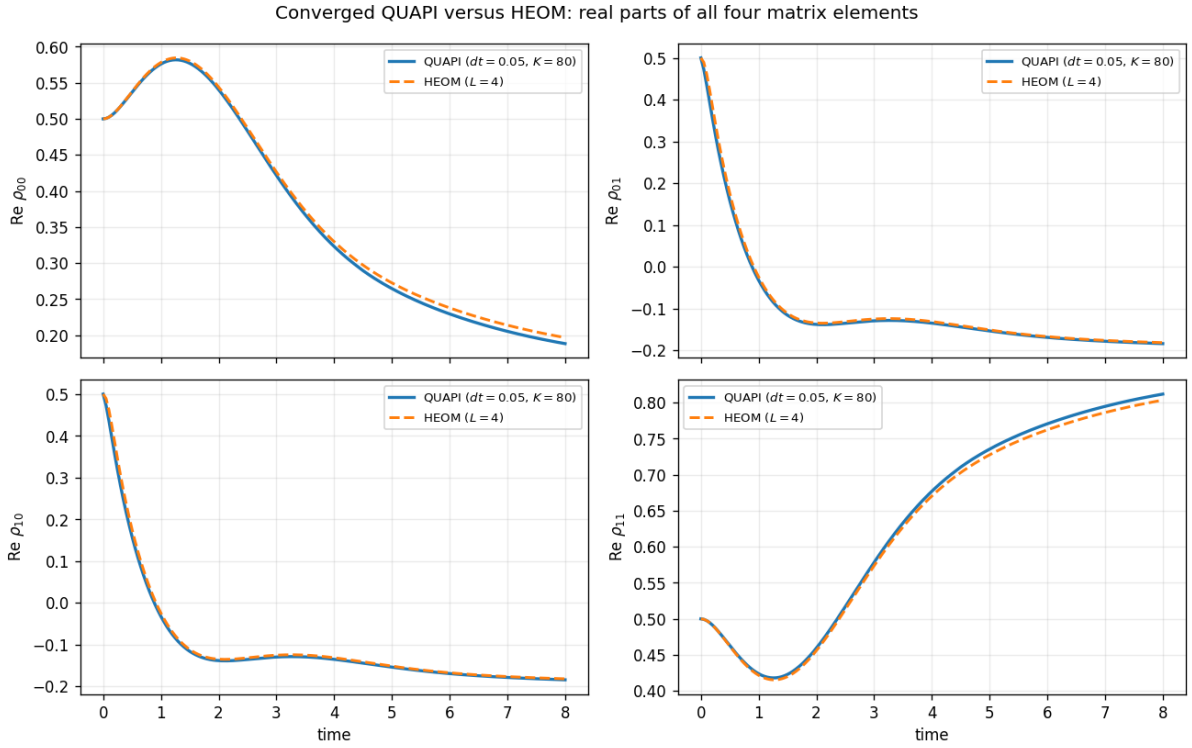


Figure 5: Real parts of all four reduced-density-matrix elements for the selected QUAPI trajectory $(dt, K) = (0.05, 80)$ and HEOM with $L_{\max} = 4$.

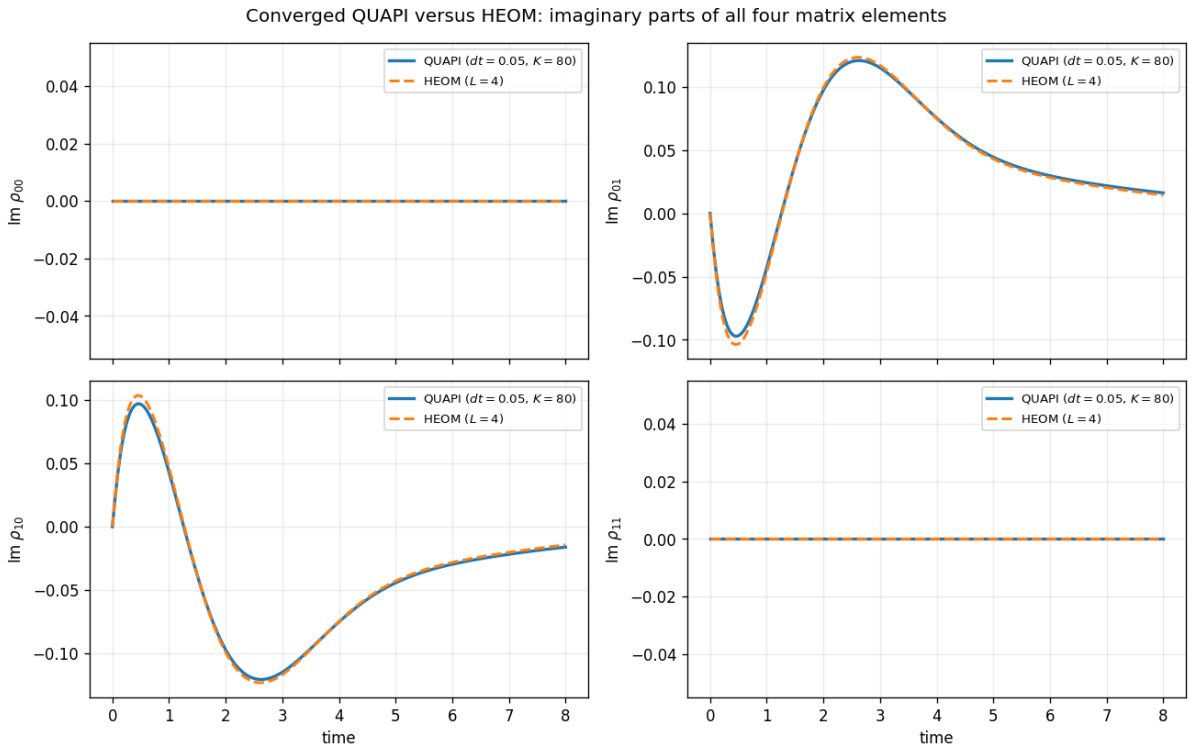


Figure 6: Imaginary parts of all four reduced-density-matrix elements for the same comparison. The off-diagonal elements provide the most sensitive transient test.

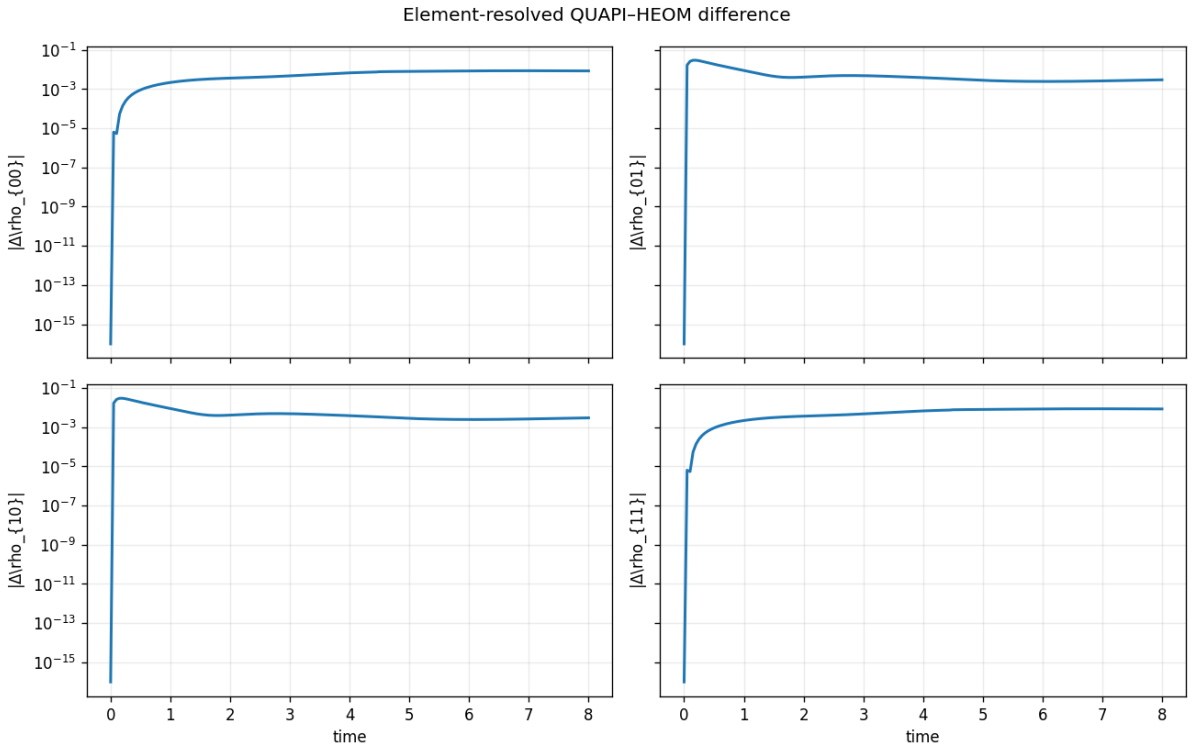


Figure 7: Absolute QUAPI–HEOM difference resolved by matrix element.

Table 5: Element-resolved QUAPI–HEOM discrepancy for the final notebook comparison.

Element	maximum absolute error	RMS absolute error	final absolute error
ρ_{00}	0.008550	0.006372	0.008372
ρ_{01}	0.029335	0.007669	0.002943
ρ_{10}	0.029335	0.007669	0.002943
ρ_{11}	0.008550	0.006372	0.008372

The practical convergence order

Converge the bath quadrature and influence coefficients; verify that MPS compression is not the limiting approximation; increase K at fixed dt; then decrease dt while holding Kdt fixed; only afterward build and converge HEOM from the same $C(t)$. Finally compare all density-matrix elements. Vary one approximation at a time.

17 Conceptual summary

The derivation can now be compressed into one common environmental-memory chain followed by two coordinate choices:

$$\boxed{\text{Gaussian bath}} \xrightarrow{\text{exact trace}} \boxed{\mathcal{F}[C]} \longrightarrow \begin{cases} C(t) \simeq \sum_k c_k e^{-\gamma_k t} \longrightarrow \{\mathcal{B}_k\} \longrightarrow \{\rho_{\boldsymbol{n}}\} & \text{HEOM,} \\ \{\eta_j\}_{j=0}^{K} \longrightarrow \{a_k, \ldots, a_{k-K}\} \longrightarrow \text{MPS} & \text{QUAPI/TEMPO.} \end{cases} \tag{17.1}$$

The one-exponential warm-up shows the essential mechanism without notation overhead: an exponential convolution satisfies a first-order ODE, and powers of that memory variable form a ladder of auxiliary operators. The general hierarchy simply has several such directions at once.

QUAPI/TEMPO retains the same Feynman–Vernon memory in a different coordinate system. QUAPI stores recent time slices explicitly; TEMPO compresses their correlations as an MPS; HEOM stores weighted mode amplitudes and their products as ADOs. In each case non-Markovian reduced dynamics is converted into time-local propagation of an enlarged memory state.

The numerical case study illustrates the methodological consequence of this viewpoint. Agreement between two methods is meaningful only after their internal approximations have been varied independently. In the supplied notebook, memory convergence is established before time-step convergence, the physical memory time is held fixed while dt is reduced, MPS compression is monitored separately, and HEOM is introduced only after QUAPI parameters are frozen. This separation of approximations is as important pedagogically as the final overlay of trajectories.

How to use the appendices

The appendices provide a slower route through the prerequisites and implementation checks. They are not required for the main logical chain, but a reader new to real-time path integrals can read Appendices A–D before Part I, while a reader implementing the algorithms can use the later appendices as debugging and code-navigation guides.

A Short-time propagator and path-integral construction

A finite-time Trotter product must be treated as an approximation. If $\hat{H} = \hat{T} + \hat{U}$, with $\hat{U}$ diagonal in the coordinate representation, then

$$\mathrm{e}^{-\mathrm{i}\epsilon(\hat{T}+\hat{U})} = \mathrm{e}^{-\mathrm{i}\epsilon\hat{T}}\mathrm{e}^{-\mathrm{i}\epsilon\hat{U}} + O(\epsilon^2). \tag{A.1}$$

The symmetric factorization

$$\mathrm{e}^{-\mathrm{i}\epsilon(\hat{T}+\hat{U})} = \mathrm{e}^{-\mathrm{i}\epsilon\hat{U}/2}\mathrm{e}^{-\mathrm{i}\epsilon\hat{T}}\mathrm{e}^{-\mathrm{i}\epsilon\hat{U}/2} + O(\epsilon^3) \tag{A.2}$$

is often preferable numerically.

For one spatial degree of freedom,

$$\langle q_{n+1}|\mathrm{e}^{-\mathrm{i}\epsilon\hat{H}}|q_n\rangle = \left(\frac{m}{2\pi\mathrm{i}\epsilon}\right)^{1/2}\exp\left[\frac{\mathrm{i}m(q_{n+1}-q_n)^2}{2\epsilon} - \mathrm{i}\epsilon U(q_n)\right] + O(\epsilon^2). \tag{A.3}$$

Inserting coordinate resolutions between M short-time factors and taking $M \to \infty$ with $M\epsilon = t$ yields

$$K(q_f, t; q_0, 0) = \int_{q(0)=q_0}^{q(t)=q_f} \mathcal{D}q\, \mathrm{e}^{\mathrm{i}S[q]}, \qquad S[q] = \int_0^t \mathrm{d}\tau\, \left[\frac{m}{2}\dot{q}^2 - U(q)\right]. \tag{A.4}$$

Applying the same construction to the bath coordinates and the system–bath coupling produces the forward–backward form in equation (3.3).

B Recovering the Schrödinger equation from the short-time kernel

The short-time free-particle kernel provides a useful factor-of-two check. Write

$$K(x, t+\epsilon; x_0, 0) = \int \frac{\mathrm{d}\eta}{A}\exp\left(\frac{\mathrm{i}m\eta^2}{2\epsilon}\right)[1 - \mathrm{i}\epsilon V(x)]K(x+\eta, t; x_0, 0), \tag{B.1}$$

with $A = \sqrt{2\pi\mathrm{i}\epsilon/m}$. Expand

$$K(x+\eta, t) = K + \eta\partial_x K + \frac{\eta^2}{2}\partial_x^2 K + \cdots. \tag{B.2}$$

The required Gaussian moments are

$$\frac{1}{A}\int \mathrm{d}\eta\, \mathrm{e}^{\mathrm{i}m\eta^2/(2\epsilon)} = 1, \tag{B.3}$$

$$\frac{1}{A}\int \mathrm{d}\eta\, \eta\, \mathrm{e}^{\mathrm{i}m\eta^2/(2\epsilon)} = 0, \tag{B.4}$$

$$\frac{1}{A}\int \mathrm{d}\eta\, \eta^2 \mathrm{e}^{\mathrm{i}m\eta^2/(2\epsilon)} = \frac{\mathrm{i}\epsilon}{m}. \tag{B.5}$$

Hence

$$(1 + \epsilon\partial_t)K = \left[1 - \mathrm{i}\epsilon V(x) + \frac{\mathrm{i}\epsilon}{2m}\partial_x^2\right]K, \tag{B.6}$$

and therefore

$$\partial_t K = -\mathrm{i}\hat{H}K, \qquad \hat{H} = -\frac{1}{2m}\partial_x^2 + V(x). \tag{B.7}$$

The coefficient of $\partial_x^2 K$ in equation (B.6) must therefore be $\mathrm{i}\epsilon/(2m)$.

C Quadratic actions and the driven harmonic oscillator

For a quadratic Lagrangian, decompose a path as

$$x(\tau) = x_{\mathrm{cl}}(\tau) + \eta(\tau), \qquad \eta(0) = \eta(T) = 0. \tag{C.1}$$

Because x_{cl} satisfies the Euler–Lagrange equation, all terms linear in η vanish after integration by parts. The propagator therefore factors into a classical phase and a fluctuation determinant,

$$K(x_f, T; x_0, 0) = F(T)\mathrm{e}^{\mathrm{i}S_{\mathrm{cl}}(x_f, x_0)}. \tag{C.2}$$

For

$$L = \frac{m}{2}\dot{x}^2 - \frac{m\omega^2}{2}x^2 + j(t)x, \tag{C.3}$$

the exact propagator is

$$K_j(x_f, T; x_0, 0) = \left(\frac{m\omega}{2\pi\mathrm{i}\sin\omega T}\right)^{1/2} \exp\left[\mathrm{i}S_{\mathrm{cl}}(x_f, x_0; j)\right], \tag{C.4}$$

where

$$\begin{aligned} S_{\mathrm{cl}} = {} & \frac{m\omega}{2\sin\omega T}\left[(x_f^2 + x_0^2)\cos\omega T - 2x_f x_0\right] \\ & + \frac{x_f}{\sin\omega T}\int_0^T \mathrm{d}t\, j(t)\sin\omega t + \frac{x_0}{\sin\omega T}\int_0^T \mathrm{d}t\, j(t)\sin\omega(T-t) \\ & - \frac{1}{m\omega\sin\omega T}\int_0^T \mathrm{d}t \int_0^t \mathrm{d}s\, j(t)j(s)\sin\omega(T-t)\sin\omega s. \end{aligned} \tag{C.5}$$

The expression is understood with the usual phase continuation across the caustics $\sin\omega T = 0$.

If one writes $x = x_h + x_p$, then expansion of the quadratic kinetic and potential terms generates the cross term

$$m\left(\dot{x}_h\dot{x}_p - \omega^2 x_h x_p\right), \tag{C.6}$$

with coefficient m, not $m/2$. For a homogeneous x_h and a fluctuation or particular solution x_p vanishing at the endpoints, the integral of this cross term vanishes after integration by parts.

D Thermal oscillator kernel and Gaussian bath elimination

For a single harmonic oscillator,

$$\langle x|\rho_{\mathrm{B}}^{\mathrm{eq}}|x'\rangle = \left[\frac{m\omega}{\pi}\tanh\left(\frac{\beta\omega}{2}\right)\right]^{1/2} \exp\left\{-\frac{m\omega}{2\sinh\beta\omega}\left[(x^2 + x'^2)\cosh\beta\omega - 2xx'\right]\right\}. \tag{D.1}$$

This follows either from the spectral representation of $\mathrm{e}^{-\beta\hat{H}_{\mathrm{B}}}$ or by analytically continuing the harmonic-oscillator propagator to imaginary time $T = -\mathrm{i}\beta$ and normalizing by Z_{B}.

For a prescribed forward system path, each bath oscillator is a driven harmonic oscillator. The forward propagator, backward propagator, and thermal kernel together form a Gaussian function of the initial and final bath coordinates. Integrating those coordinates exactly for every independent oscillator produces equation (4.3) and the correlation function in equation (5.1). This is why the environmental elimination itself does not require a perturbative approximation.

E Consistency checks and common pitfalls

The following checks are useful when deriving or implementing HEOM:

1. Treat short-time Trotter products as asymptotic factorizations, not exact finite-step identities.
2. Keep the forward and backward endpoint actions distinct: V^+ maps to left multiplication and V^- to right multiplication.
3. Verify the zeroth-tier bath term is a commutator, $-\mathrm{i}\sum_k[\hat{V}, \rho_{\boldsymbol{e}_k}]$.
4. In the short-time kernel, the second-derivative coefficient is $\mathrm{i}\epsilon/(2m)$.
5. State the counterterm convention explicitly.
6. When complex decay rates are used, ensure that $C(t)$ and $C^*(t)$ are represented in a mutually consistent exponential basis rather than assuming coefficient-wise complex conjugation without checking the basis.

F Code-to-equation guide for the convergence notebook

The notebook is intentionally short enough that each major routine can be associated with one mathematical object. Reading the code through this map is more useful than reading it linearly.

Table 6: Map between the principal notebook routines and the equations in the tutorial.

Routine	Mathematical role
`bath_alpha`	Evaluates the continuum bath correlation $C(t)$ from the Ohmic spectral density.
`influence_coefficients`	Integrates $C(t)$ over pairs of Trotter cells to obtain $\eta_0, \ldots, \eta_K$.
`build_system_kernel`	Constructs the forward–backward system superpropagator $U \otimes U^*$.
`influence_table`	Precomputes the pairwise influence factors in equation (16.4).
`_mps_add_new_star`	Attaches the new path-pair site and all influence links to the retained history.
`_mps_right_canonicalize`	Uses right-to-left QR sweeps to place the MPS in a stable gauge before truncation.
`_mps_compress_canonical`	Performs left-to-right SVD compression and accumulates discarded singular-value norm.
`_mps_sum_oldest`	Sums the oldest temporal physical index directly on the MPS when the memory window is full.
`_reduced_from_persistent_mps`	Contracts historical sites and reshapes the newest four-state index into the 2×2 reduced density matrix.
`propagate_quapi`	Repeats the persistent finite-memory QUAPI/TEMPO update and records bond/compression diagnostics.
`trajectory_difference`	Measures maximum elementwise changes between nested QUAPI trajectories.
`heom_multiindices`	Enumerates all ADO multi-indices through a requested hierarchy tier.
`prepare_heom_hierarchy`	Precomputes upward/downward neighbors and $\sum_k n_k \gamma_k$.
`propagate_heom`	Integrates the complete unscaled HEOM with RK4 substepping.

F.1 How the persistent MPS update should be read

The persistent update is easiest to understand as five operations repeated at every time step:

1. if necessary, trace out the oldest retained path-pair index;
2. attach a new four-state path-pair site;
3. apply the system propagator to the newest pair and all retained influence factors to the new site;
4. canonicalize and compress the MPS;
5. sum historical physical indices to recover $\hat{\rho}_S(t)$ and record compression diagnostics.

The temporary bond enlargement carries the value of the new path-pair label through the complete star of influence factors. This is why the entire update can be exact before SVD truncation without forming a dense augmented tensor.

F.2 How the HEOM code mirrors the analytical hierarchy

The array of ADOs has shape $(N_{\mathrm{ADO}}, 2, 2)$, with element zero corresponding to $\rho_{\mathbf{0}} = \hat{\rho}_S$. The right-hand side is a literal transcription of equation (6.46): the Hamiltonian commutator gives coherent evolution, a precomputed damping array gives $\sum_k n_k \gamma_k$, the `up` table indexes $\rho_{\boldsymbol{n}+\boldsymbol{e}_k}$, and the `down` table indexes $\rho_{\boldsymbol{n}-\boldsymbol{e}_k}$. Precomputing these graph connections keeps tuple manipulation out of the inner time-integration loop.

F.3 Recommended extension of the calculation

For a stricter production calculation, extend the notebook in the following order:

1. verify the frequency-grid and time-cell quadratures defining $C(t)$ and η_j;
2. tighten the MPS cutoff and/or raise the bond cap until the trajectory is unchanged;
3. append larger K values if the memory criterion is not met;
4. preserve the selected τ_{mem} and append another factor-of-two time-step refinement, e.g. $dt = 0.025$ and $K = 160$ for $\tau_{\mathrm{mem}} = 4$;
5. improve the exponential representation of $C(t)$, for example with more terms or a more structured Padé decomposition;
6. compare additional HEOM tiers until the hierarchy change is below the desired tolerance;
7. only then interpret the remaining element-resolved QUAPI–HEOM difference.

G Complete Python source used by the notebook

For reproducibility, the complete executable sequence of code cells from `QUAPI_HEOM_convergence(1).ipynb` is supplied below as a standalone Python file. The notebook markdown is represented by section comments; executable code is otherwise kept in notebook order.

```python
# Complete Python source extracted from QUAPI_HEOM_convergence(1).ipynb
# Markdown cells are summarized as section comments; code cell order is unchanged.


# ============================================================================
# QUAPI convergence first, then comparison with HEOM
# ============================================================================


# --- notebook code cell 1 ---
```

```python
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.linalg import expm
from itertools import combinations_with_replacement
from IPython.display import display

np.set_printoptions(precision=6, suppress=True)
plt.rcParams.update({"figure.dpi": 120, "axes.grid": True, "grid.alpha": 0.25})

# ---------------------------- physical model ----------------------------
epsilon = 1.0
Delta = 1.0
alpha_SB = 0.08
omega_c = 5.0
temperature = 0.5
beta = 1.0 / temperature

t_final = 8.0

# --------------------- QUAPI convergence controls ----------------------
# Stage 1: converge K at this reference dt.
dt_memory_test = 0.10
K_candidates = [10, 20, 30, 40]
memory_tolerance = 1.0e-3

# Stage 2: converge dt while holding tau_mem fixed at the value selected above.
# Powers of two make successive trajectories share exact time points.
dt_candidates = [0.20, 0.10, 0.05]
dt_tolerance = 2.0e-2

# Persistent MPS/TEMPO controls. Tighten these if compression becomes the limiting error.
mps_rel_cutoff = 3.0e-6
mps_max_bond = 24

# Bath quadrature controls used to build eta_j.
bath_n_omega = 3000
bath_omega_max_factor = 12.0
influence_subdivisions = 120

# ---------------------------- HEOM controls ----------------------------
heom_n_exp = 12
heom_tiers = (3, 4)
heom_fit_ridge = 1.0e-6
heom_rel_floor = 1.0e-2

print("QUAPI convergence protocol")
print(f"  memory sweep: dt = {dt_memory_test:g}, K = {K_candidates}")
print(f"  dt sweep:     dt = {dt_candidates}, with K adjusted to keep K*dt fixed")
print(f"  tolerances:   memory = {memory_tolerance:.1e}, dt = {dt_tolerance:.1e}")

# ============================================================================
# Bath correlation function and discrete influence coefficients
# ============================================================================


# --- notebook code cell 3 ---
def coth_stable(x):
    x = np.asarray(x, dtype=float)
    out = np.empty_like(x)
    small = np.abs(x) < 1e-5
    out[small] = 1.0/x[small] + x[small]/3.0
    out[~small] = 1.0/np.tanh(x[~small])
    return out


def bath_alpha(t, n_omega=bath_n_omega,
```

```
                omega_max_factor=bath_omega_max_factor, chunk_size=256):
    """Numerical Ohmic bath correlation C(t), evaluated in chunks."""
    scalar_input = np.ndim(t) == 0
    t = np.atleast_1d(t).astype(float)

    w = np.linspace(1e-7, omega_max_factor*omega_c, n_omega)
    J = 2*np.pi*alpha_SB*w*np.exp(-w/omega_c)
    thermal = coth_stable(0.5*beta*w)
    pref = J/np.pi

    ans = np.empty(t.size, dtype=complex)
    for start in range(0, t.size, chunk_size):
        stop = min(start + chunk_size, t.size)
        phase = np.outer(t[start:stop], w)
        integrand = pref[None, :] * (
            thermal[None, :]*np.cos(phase) - 1j*np.sin(phase)
        )
        ans[start:stop] = np.trapezoid(integrand, w, axis=1)

    return ans[0] if scalar_input else ans


def influence_coefficients(K, dt, subdivisions=influence_subdivisions):
    """Return eta_0,...,eta_K for a uniform QUAPI grid."""
    M = int(subdivisions)
    if M < 8:
        raise ValueError("Use at least 8 subdivisions per time cell.")

    # All eta integrals can be indexed from this single C(t) grid.
    h = dt / M
    t_grid = np.arange((K + 1)*M + 1, dtype=float) * h
    C_grid = bath_alpha(t_grid)

    eta = np.zeros(K + 1, dtype=complex)

    x0 = np.arange(M + 1, dtype=float) * h
    eta[0] = np.trapezoid((dt - x0)*C_grid[:M+1], x0)

    x = np.arange(-M, M + 1, dtype=float) * h
    weight = dt - np.abs(x)
    for j in range(1, K + 1):
        lo = (j - 1)*M
        hi = (j + 1)*M + 1
        eta[j] = np.trapezoid(weight*C_grid[lo:hi], x)

    return eta


# Visual check of the bath time scale.
t_plot = np.linspace(0.0, 5.0, 500)
C_plot = bath_alpha(t_plot)
fig, ax = plt.subplots(figsize=(7.0, 3.6))
ax.plot(t_plot, C_plot.real, label=r"Re $C(t)$")
ax.plot(t_plot, C_plot.imag, label=r"Im $C(t)$")
ax.set(xlabel="time", ylabel=r"$C(t)$", title="Ohmic bath correlation")
ax.legend()
plt.tight_layout()
plt.show()

# ============================================================================
# Persistent finite-memory QUAPI/TEMPO propagator
# ============================================================================


# --- notebook code cell 5 ---
sx = np.array([[0, 1], [1, 0]], dtype=complex)
sz = np.array([[1, 0], [0, -1]], dtype=complex)
HS = 0.5*epsilon*sz + 0.5*Delta*sx
```

```

sval = np.array([+1.0, -1.0])
sp = np.repeat(sval, 2)
sm = np.tile(sval, 2)

rho0 = 0.5*np.array([[1, 1], [1, 1]], dtype=complex)  # |+x><+x|


def build_system_kernel(dt):
    U = expm(-1j*HS*dt)
    Ksys = np.empty((4, 4), dtype=complex)
    for anew in range(4):
        ip, im = divmod(anew, 2)
        for aold in range(4):
            jp, jm = divmod(aold, 2)
            Ksys[anew, aold] = U[ip, jp] * U[im, jm].conjugate()
    return Ksys


def influence_table(eta_coeffs):
    return [
        np.exp(-(sp[:, None]-sm[:, None]) *
               (z*sp[None, :] - z.conjugate()*sm[None, :]))
        for z in eta_coeffs
    ]


def physical_cleanup(rho):
    """Remove tiny trace/Hermiticity drift without enforcing positivity."""
    rho = 0.5*(rho + rho.conj().T)
    return rho/np.trace(rho)


def _mps_sum_oldest(cores):
    out = list(cores)
    tail = out[-1].sum(axis=(1, 2))
    prev = np.tensordot(out[-2], tail, axes=([2], [0]))[:, :, None]
    return out[:-2] + [prev]


def _mps_add_new_star(cores, K, Ilag, Ksys):
    m = len(cores)
    if not (1 <= m <= K):
        raise ValueError("Expected 1 <= number of retained old sites <= K.")

    self_factor = np.diag(Ilag[0])
    G0 = np.zeros((1, 4, 4), dtype=complex)
    for a0 in range(4):
        G0[0, a0, a0] = self_factor[a0]

    new_cores = [G0]
    for j, G in enumerate(cores, start=1):
        rL, d, rR = G.shape
        factor = Ilag[j].copy()
        if j == 1:
            factor *= Ksys

        if j < m:
            H = np.zeros((4*rL, 4, 4*rR), dtype=complex)
            for a0 in range(4):
                H[a0*rL:(a0+1)*rL, :, a0*rR:(a0+1)*rR] = (
                    G * factor[a0][None, :, None]
                )
        else:
            H = np.zeros((4*rL, 4, rR), dtype=complex)
            for a0 in range(4):
                H[a0*rL:(a0+1)*rL, :, :] = (
                    G * factor[a0][None, :, None]
```

```
                )
        new_cores.append(H)

    return new_cores


def _mps_right_canonicalize(cores):
    cores = list(cores)
    for i in range(len(cores)-1, 0, -1):
        G = cores[i]
        rL, d, rR = G.shape
        M = G.reshape(rL, d*rR)
        Q, R = np.linalg.qr(M.T, mode="reduced")
        rnew = Q.shape[1]
        cores[i] = Q.T.reshape(rnew, d, rR)
        cores[i-1] = np.tensordot(cores[i-1], R.T, axes=([2], [0]))
    return cores


def _mps_compress_canonical(cores, rel_cutoff=mps_rel_cutoff,
                            max_bond=mps_max_bond):
    cores = _mps_right_canonicalize(cores)
    ranks = []
    discarded_sq = 0.0

    for i in range(len(cores)-1):
        G = cores[i]
        rL, d, rR = G.shape
        M = G.reshape(rL*d, rR)
        Uloc, s, Vh = np.linalg.svd(M, full_matrices=False)

        threshold = rel_cutoff*s[0] if s.size else 0.0
        keep = max(1, int(np.count_nonzero(s > threshold)))
        if max_bond is not None:
            keep = min(keep, max_bond)

        discarded_sq += float(np.sum(s[keep:]**2))
        cores[i] = Uloc[:, :keep].reshape(rL, d, keep)
        transfer = s[:keep, None]*Vh[:keep, :]
        cores[i+1] = np.tensordot(transfer, cores[i+1], axes=([1], [0]))
        ranks.append(keep)

    return cores, ranks, np.sqrt(discarded_sq)


def _reduced_from_persistent_mps(cores):
    env = np.ones(1, dtype=complex)
    for G in cores[:0:-1]:
        env = np.einsum("lar,r->l", G, env)
    newest = np.einsum("lar,r->a", cores[0], env)
    return newest.reshape(2, 2)


def propagate_quapi(dt, K, t_final=t_final, eta_coeffs=None,
                    rel_cutoff=mps_rel_cutoff, max_bond=mps_max_bond):
    """Persistent-MPS finite-memory QUAPI/TEMPO propagation."""
    n_steps_float = t_final/dt
    n_steps = int(round(n_steps_float))
    if not np.isclose(n_steps*dt, t_final, rtol=0, atol=1e-12):
        raise ValueError("Choose dt so that t_final/dt is an integer.")

    if eta_coeffs is None:
        eta_coeffs = influence_coefficients(K, dt)
    eta_coeffs = np.asarray(eta_coeffs)
    if eta_coeffs.size < K + 1:
        raise ValueError("eta_coeffs does not contain all lags through K.")

    Ilag = influence_table(eta_coeffs[:K+1])
```

```
    Ksys = build_system_kernel(dt)
    cores = [rho0.reshape(1, 4, 1).copy()]

    rhos = [rho0.copy()]
    max_ranks = [1]
    discarded = [0.0]

    tic = time.perf_counter()
    for _ in range(n_steps):
        if len(cores) == K + 1:
            cores = _mps_sum_oldest(cores)

        cores = _mps_add_new_star(cores, K, Ilag, Ksys)
        cores, ranks, disc = _mps_compress_canonical(
            cores, rel_cutoff=rel_cutoff, max_bond=max_bond
        )

        rhos.append(physical_cleanup(_reduced_from_persistent_mps(cores)))
        max_ranks.append(max(ranks, default=1))
        discarded.append(disc)

    runtime = time.perf_counter() - tic
    times = dt*np.arange(n_steps + 1)
    return {
        "dt": dt, "K": K, "times": times, "rho": np.asarray(rhos),
        "rank": np.asarray(max_ranks), "discarded": np.asarray(discarded),
        "runtime": runtime,
    }


def trajectory_difference(run_a, run_b):
    """Maximum elementwise |Delta rho| on the coarser of two nested time grids."""
    if run_a["dt"] >= run_b["dt"]:
        coarse, fine = run_a, run_b
    else:
        coarse, fine = run_b, run_a

    ratio = coarse["dt"] / fine["dt"]
    iratio = int(round(ratio))
    if not np.isclose(ratio, iratio, atol=1e-12):
        raise ValueError("For convergence tests use nested time steps, e.g. dt, dt/2, dt/4.")

    rho_fine = fine["rho"][::iratio]
    rho_fine = rho_fine[:len(coarse["rho"])]
    if len(rho_fine) != len(coarse["rho"]):
        raise RuntimeError("Time grids do not cover the same final time.")

    delta = np.abs(coarse["rho"] - rho_fine)
    per_element = np.max(delta, axis=0)
    return float(np.max(delta)), per_element

# ============================================================================
# Stage 1 -- convergence of the influence-functional memory length $K$
# ============================================================================


# --- notebook code cell 7 ---
# Compute eta_j once through the largest K for this fixed dt.
eta_memory = influence_coefficients(max(K_candidates), dt_memory_test)

memory_runs = {}
memory_rows = []
previous = None
K_converged = None

for K in K_candidates:
    run = propagate_quapi(
        dt_memory_test, K, eta_coeffs=eta_memory,
```

```
        rel_cutoff=mps_rel_cutoff, max_bond=mps_max_bond
    )
    memory_runs[K] = run

    if previous is None:
        change = np.nan
    else:
        change, _ = trajectory_difference(previous, run)

    tail = abs(bath_alpha(K*dt_memory_test))/max(abs(bath_alpha(0.0)), 1e-15)
    memory_rows.append({
        "K": K,
        "tau_mem": K*dt_memory_test,
        "successive_change": change,
        "bath_tail_ratio": float(tail),
        "max_MPS_bond": int(np.max(run["rank"])),
        "max_discarded_norm": float(np.max(run["discarded"])),
        "runtime_s": run["runtime"],
    })

    if previous is not None and K_converged is None and change < memory_tolerance:
        K_converged = K

    previous = run

memory_table = pd.DataFrame(memory_rows)
display(memory_table)

fig, ax = plt.subplots(1, 2, figsize=(11, 3.8))
valid = np.isfinite(memory_table["successive_change"].to_numpy(float))
ax[0].semilogy(memory_table.loc[valid, "tau_mem"],
               memory_table.loc[valid, "successive_change"], "o-",
               label=r"$\max|\rho_K-\rho_{K_{prev}}|$")
ax[0].axhline(memory_tolerance, ls="--", lw=1.2, label="tolerance")
ax[0].set(xlabel=r"memory time $Kdt$", ylabel="successive QUAPI change",
          title="Memory convergence")
ax[0].legend()

ax[1].semilogy(memory_table["tau_mem"], memory_table["bath_tail_ratio"], "s-")
ax[1].set(xlabel=r"memory time $Kdt$", ylabel=r"$|C(Kdt)|/|C(0)|$",
          title="Bath correlation at the cutoff")
plt.tight_layout()
plt.show()

if K_converged is None:
    raise RuntimeError(
        "K did not converge over K_candidates. Extend K_candidates before running the dt sweep."
    )

tau_mem_converged = K_converged * dt_memory_test
print(f"Selected memory depth: K = {K_converged}")
print(f"Selected physical memory time: tau_mem = {tau_mem_converged:.6g}")

# ============================================================================
# Stage 2 -- convergence of the Trotter time step $dt$ at fixed $\tau_{\rm mem}$
# ============================================================================


# --- notebook code cell 9 ---
dt_runs = {}
dt_rows = []
previous = None

for dt_i in dt_candidates:
    K_i = int(np.ceil(tau_mem_converged/dt_i - 1e-12))
    eta_i = influence_coefficients(K_i, dt_i)
    run = propagate_quapi(
        dt_i, K_i, eta_coeffs=eta_i,
```

```
        rel_cutoff=mps_rel_cutoff, max_bond=mps_max_bond
    )
    dt_runs[dt_i] = run

    if previous is None:
        change = np.nan
        per_element = np.full((2, 2), np.nan)
    else:
        change, per_element = trajectory_difference(previous, run)

    dt_rows.append({
        "dt": dt_i,
        "K": K_i,
        "actual_tau_mem": K_i*dt_i,
        "successive_change": change,
        "max_MPS_bond": int(np.max(run["rank"])),
        "max_discarded_norm": float(np.max(run["discarded"])),
        "runtime_s": run["runtime"],
        "change_rho00": per_element[0,0],
        "change_rho01": per_element[0,1],
        "change_rho10": per_element[1,0],
        "change_rho11": per_element[1,1],
    })
    previous = run

dt_table = pd.DataFrame(dt_rows)
display(dt_table)

fig, ax = plt.subplots(1, 2, figsize=(11, 3.8))
valid = np.isfinite(dt_table["successive_change"].to_numpy(float))
ax[0].loglog(dt_table.loc[valid, "dt"],
             dt_table.loc[valid, "successive_change"], "o-")
ax[0].axhline(dt_tolerance, ls="--", lw=1.2, label="tolerance")
ax[0].invert_xaxis()
ax[0].set(xlabel=r"time step $dt$", ylabel="successive QUAPI change",
          title=r"$dt$ convergence at fixed $\tau_{mem}$")
ax[0].legend()

for col, lab in [
    ("change_rho00", r"$\rho_{00}$"),
    ("change_rho01", r"$\rho_{01}$"),
    ("change_rho10", r"$\rho_{10}$"),
    ("change_rho11", r"$\rho_{11}$"),
]:
    ax[1].loglog(dt_table.loc[valid, "dt"], dt_table.loc[valid, col], "o-", label=lab)
ax[1].invert_xaxis()
ax[1].set(xlabel=r"time step $dt$", ylabel="max elementwise change",
          title="Element-resolved time-step convergence")
ax[1].legend(fontsize=8)
plt.tight_layout()
plt.show()

# The final pair must pass before HEOM is used as an external benchmark.
last_change = float(dt_table["successive_change"].iloc[-1])
if not np.isfinite(last_change) or last_change >= dt_tolerance:
    raise RuntimeError(
        "dt is not converged at the requested tolerance. Append a smaller nested value "
        "to dt_candidates (for example dt_candidates[-1]/2) and rerun this stage."
    )

dt_converged = float(dt_candidates[-1])
quapi_converged_run = dt_runs[dt_converged]
K_final = int(quapi_converged_run["K"])
times = quapi_converged_run["times"]
rho_quapi = quapi_converged_run["rho"]

print("QUAPI convergence passed")
print(f"  dt = {dt_converged:g}")
```

```
print(f"  K  = {K_final}")
print(f"  K*dt = {K_final*dt_converged:g}")
print(f"  final successive-dt change = {last_change:.3e}")

# ============================================================================
# HEOM benchmark -- evaluated only after QUAPI convergence
# ============================================================================


# --- notebook code cell 11 ---
# ---------------- 1) Exponential representation of C(t) ----------------
fit_t = np.unique(np.concatenate([
    np.linspace(0.0, t_final, 600),
    np.geomspace(max(1e-6, 1e-3*dt_converged), t_final, 350)
]))
C_target = bath_alpha(fit_t)

gamma_min = max(0.05, 0.25/max(t_final, dt_converged))
gamma_max = 8.0*omega_c
heom_gamma = np.geomspace(gamma_min, gamma_max, heom_n_exp)

E = np.exp(-np.outer(fit_t, heom_gamma))
fit_floor = heom_rel_floor*max(abs(C_target[0]), 1e-14)
fit_w = 1.0/np.sqrt(np.abs(C_target)**2 + fit_floor**2)
Ew = E*fit_w[:, None]
bw = C_target*fit_w

normal = Ew.conj().T @ Ew + heom_fit_ridge*np.eye(heom_n_exp)
heom_c = np.linalg.solve(normal, Ew.conj().T @ bw)
C_fit = E @ heom_c

fit_rel_l2 = np.linalg.norm(C_fit-C_target)/np.linalg.norm(C_target)
fit_max_abs = np.max(np.abs(C_fit-C_target))
print(f"HEOM correlation fit: relative L2 = {fit_rel_l2:.3e}, max abs = {fit_max_abs:.3e}")

fig, ax = plt.subplots(1, 2, figsize=(10.5, 3.5), sharex=True)
ax[0].plot(fit_t, C_target.real, lw=2.0, label="target")
ax[0].plot(fit_t, C_fit.real, "--", lw=1.6, label="exponential fit")
ax[0].set(xlabel="time", ylabel=r"Re $C(t)$", title="HEOM bath fit: real part")
ax[0].legend()
ax[1].plot(fit_t, C_target.imag, lw=2.0, label="target")
ax[1].plot(fit_t, C_fit.imag, "--", lw=1.6, label="exponential fit")
ax[1].set(xlabel="time", ylabel=r"Im $C(t)$", title="HEOM bath fit: imaginary part")
ax[1].legend()
plt.tight_layout()
plt.show()


# ----------------------- 2) HEOM hierarchy -----------------------------
def heom_multiindices(n_exp, max_tier):
    out = []
    for tier in range(max_tier + 1):
        for combo in combinations_with_replacement(range(n_exp), tier):
            n = np.zeros(n_exp, dtype=int)
            if tier:
                n += np.bincount(combo, minlength=n_exp)
            out.append(tuple(n))
    return out


def prepare_heom_hierarchy(gammas, max_tier):
    n_exp = len(gammas)
    multi = heom_multiindices(n_exp, max_tier)
    lookup = {n: i for i, n in enumerate(multi)}
    narr = np.asarray(multi, dtype=int)
    n_ado = len(multi)

    up = np.full((n_ado, n_exp), -1, dtype=int)
```

```
    down = np.full((n_ado, n_exp), -1, dtype=int)

    for i, n in enumerate(narr):
        if n.sum() < max_tier:
            for k in range(n_exp):
                m = n.copy(); m[k] += 1
                up[i, k] = lookup[tuple(m)]
        for k in range(n_exp):
            if n[k] > 0:
                m = n.copy(); m[k] -= 1
                down[i, k] = lookup[tuple(m)]

    damping = narr @ np.asarray(gammas)
    return narr, up, down, damping


def propagate_heom(gammas, coeffs, max_tier, rho0, dt_out, n_out):
    gammas = np.asarray(gammas, float)
    coeffs = np.asarray(coeffs, complex)
    narr, up, down, damping = prepare_heom_hierarchy(gammas, max_tier)
    n_ado, n_exp = narr.shape

    R = np.zeros((n_ado, 2, 2), dtype=complex)
    R[0] = rho0

    # RK4 substep chosen so gamma_max*h <= 0.25.
    n_sub = max(4, int(np.ceil(dt_out*gammas.max()/0.25)))
    h = dt_out/n_sub

    def rhs(Rnow):
        dR = -1j*(HS @ Rnow - Rnow @ HS)
        dR -= damping[:, None, None]*Rnow

        for k in range(n_exp):
            valid = up[:, k] >= 0
            if np.any(valid):
                X = Rnow[up[valid, k]]
                dR[valid] += -1j*(sz @ X - X @ sz)

            valid = down[:, k] >= 0
            if np.any(valid):
                X = Rnow[down[valid, k]]
                nk = narr[valid, k, None, None]
                dR[valid] += -1j*nk*(
                    coeffs[k]*(sz @ X) - coeffs[k].conjugate()*(X @ sz)
                )
        return dR

    rhos = [R[0].copy()]
    tic = time.perf_counter()
    for _ in range(n_out):
        for _ in range(n_sub):
            k1 = rhs(R)
            k2 = rhs(R + 0.5*h*k1)
            k3 = rhs(R + 0.5*h*k2)
            k4 = rhs(R + h*k3)
            R += (h/6.0)*(k1 + 2*k2 + 2*k3 + k4)
        rhos.append(R[0].copy())

    return np.asarray(rhos), time.perf_counter()-tic, n_ado, n_sub


n_out = len(times) - 1
heom_runs = {}
for L in heom_tiers:
    rho_L, runtime_L, n_ado_L, n_sub_L = propagate_heom(
        heom_gamma, heom_c, L, rho0, dt_converged, n_out
    )
```

```
    heom_runs[L] = rho_L
    print(f"HEOM L={L}: {n_ado_L} ADOs, {n_sub_L} RK4 substeps/output step, "
          f"runtime={runtime_L:.2f} s")

rho_heom = heom_runs[max(heom_tiers)]
rho_heom_prev = heom_runs[min(heom_tiers)]
heom_tier_change = np.max(np.abs(rho_heom-rho_heom_prev))
print(f"Max HEOM tier change L={min(heom_tiers)} -> L={max(heom_tiers)}: "
      f"{heom_tier_change:.3e}")

# ============================================================================
# Final comparison of all four reduced-density-matrix elements
# ============================================================================


# --- notebook code cell 13 ---
elements = [
    (0, 0, r"$\rho_{00}$"),
    (0, 1, r"$\rho_{01}$"),
    (1, 0, r"$\rho_{10}$"),
    (1, 1, r"$\rho_{11}$"),
]

# ----------------------------- real parts -------------------------------
fig, axes = plt.subplots(2, 2, figsize=(11, 7), sharex=True)
for ax, (i, j, lab) in zip(axes.ravel(), elements):
    ax.plot(times, rho_quapi[:, i, j].real, lw=2.0,
            label=fr"QUAPI ($dt={dt_converged:g}$, $K={K_final}$)")
    ax.plot(times, rho_heom[:, i, j].real, "--", lw=1.8,
            label=fr"HEOM ($L={max(heom_tiers)}$)")
    ax.set_ylabel(fr"Re {lab}")
    ax.legend(fontsize=8)
for ax in axes[-1]:
    ax.set_xlabel("time")
fig.suptitle("Converged QUAPI versus HEOM: real parts of all four matrix elements")
plt.tight_layout()
plt.show()

# ---------------------------- imaginary parts -----------------------------
fig, axes = plt.subplots(2, 2, figsize=(11, 7), sharex=True)
for ax, (i, j, lab) in zip(axes.ravel(), elements):
    ax.plot(times, rho_quapi[:, i, j].imag, lw=2.0,
            label=fr"QUAPI ($dt={dt_converged:g}$, $K={K_final}$)")
    ax.plot(times, rho_heom[:, i, j].imag, "--", lw=1.8,
            label=fr"HEOM ($L={max(heom_tiers)}$)")
    ax.set_ylabel(fr"Im {lab}")
    ax.legend(fontsize=8)
for ax in axes[-1]:
    ax.set_xlabel("time")
fig.suptitle("Converged QUAPI versus HEOM: imaginary parts of all four matrix elements")
plt.tight_layout()
plt.show()

# --------------------------- absolute errors ------------------------------
fig, axes = plt.subplots(2, 2, figsize=(11, 7), sharex=True, sharey=True)
summary_rows = []
for ax, (i, j, lab) in zip(axes.ravel(), elements):
    err = np.abs(rho_quapi[:, i, j] - rho_heom[:, i, j])
    ax.semilogy(times, np.maximum(err, 1e-16), lw=1.8)
    ax.set_ylabel(fr"$|\Delta {lab}|$")
    summary_rows.append({
        "element": lab.replace("$", ""),
        "max_abs_error": float(np.max(err)),
        "rms_abs_error": float(np.sqrt(np.mean(err**2))),
        "final_abs_error": float(err[-1]),
    })
for ax in axes[-1]:
    ax.set_xlabel("time")
```

```
fig.suptitle("Element-resolved QUAPI-HEOM difference")
plt.tight_layout()
plt.show()

comparison_table = pd.DataFrame(summary_rows)
display(comparison_table)

# ------------------------ numerical sanity checks -------------------------
trace_q = np.trace(rho_quapi, axis1=1, axis2=2)
trace_h = np.trace(rho_heom, axis1=1, axis2=2)
herm_q = np.max(np.abs(rho_quapi-rho_quapi.conj().transpose(0,2,1)))
herm_h = np.max(np.abs(rho_heom-rho_heom.conj().transpose(0,2,1)))

print("Final comparison summary")
print(f"  converged QUAPI: dt={dt_converged:g}, K={K_final}, K*dt={K_final*dt_converged:g}")
print(f"  max QUAPI trace error: {np.max(np.abs(trace_q-1)):.3e}")
print(f"  max HEOM trace error:  {np.max(np.abs(trace_h-1)):.3e}")
print(f"  max QUAPI Hermiticity error: {herm_q:.3e}")
print(f"  max HEOM Hermiticity error:  {herm_h:.3e}")
print(f"  max over all four QUAPI-HEOM element errors: "
      f"{comparison_table['max_abs_error'].max():.3e}")

# ============================================================================
# Interpretation of the convergence logic
# ============================================================================
```

Listing 10: Complete Python implementation used for the two-stage QUAPI convergence study and subsequent HEOM benchmark.

References

[1] Yoshitaka Tanimura and Ryogo Kubo. "Time Evolution of a Quantum System in Contact with a Nearly Gaussian–Markoffian Noise Bath". In: *Journal of the Physical Society of Japan* 58.1 (1989), pp. 101–114. DOI: 10.1143/JPSJ.58.101.

[2] Yoshitaka Tanimura. "Stochastic Liouville, Langevin, Fokker–Planck, and Master Equation Approaches to Quantum Dissipative Systems". In: *Journal of the Physical Society of Japan* 75.8 (2006), p. 082001. DOI: 10.1143/JPSJ.75.082001.

[3] Rui-Xue Xu, Ping Cui, Xin-Qi Li, Yi Mo, and YiJing Yan. "Exact Quantum Master Equation via the Calculus on Path Integrals". In: *The Journal of Chemical Physics* 122.4 (2005), p. 041103. DOI: 10.1063/1.1850899.

[4] Qiang Shi, Liping Chen, Guangjun Nan, Rui-Xue Xu, and YiJing Yan. "Efficient Hierarchical Liouville Space Propagator to Quantum Dissipative Dynamics". In: *The Journal of Chemical Physics* 130.8 (2009), p. 084105. DOI: 10.1063/1.3077918.

[5] Richard P. Feynman, Albert R. Hibbs, and Daniel F. Styer. *Quantum Mechanics and Path Integrals*. Emended. New York: McGraw–Hill, 2005.

[6] Richard P. Feynman and Frank L. Vernon. "The Theory of a General Quantum System Interacting with a Linear Dissipative System". In: *Annals of Physics* 24 (1963), pp. 118–173. DOI: 10.1016/0003-4916(63)90068-X.

[7] Jie Hu, Rui-Xue Xu, and YiJing Yan. "Communication: Padé spectrum decomposition of Fermi function and Bose function". In: *The Journal of Chemical Physics* 133.10 (2010), p. 101106. DOI: 10.1063/1.3484491.

[8] Jie Hu, Meng Luo, Feng Jiang, Rui-Xue Xu, and YiJing Yan. “Padé spectrum decompositions of quantum distribution functions and optimal hierarchical equations of motion construction for quantum open systems”. In: *The Journal of Chemical Physics* 134.24 (2011), p. 244106. DOI: 10.1063/1.3602466.

[9] Jin-Jin Ding, Jian Xu, Jie Hu, Rui-Xue Xu, and YiJing Yan. “Optimized hierarchical equations of motion theory for Drude dissipation and efficient implementation to nonlinear spectroscopies”. In: *The Journal of Chemical Physics* 135.16 (2011), p. 164107. DOI: 10.1063/1.3653479.