30 January 2026

QFlux: Classical Foundations for Quantum Dynamics Simulation. Part I - Building Intuition and Computational Workflows

Brandon C Allen[1], Xiaohan Dan[1], Delmar G A Cabral[1], Nam P Vu[2,1,3], Cameron Cianci[4], Alexander V Soudackov[1], Rishab Dutta[1], Sabre Kais[5], Eitan Geva[6], Victor S Batista[1,7]

1. Department of Chemistry Yale University
2. ‡Department of Electrical Engineering and Computer Science Massachusetts Institute of Technology
3. Research Laboratory of Electronics Massachusetts Institute of Technology
4. Department of Physics University of Connecticut
5. Department of Electrical and Computer Engineering, Department of Chemistry North Carolina State University
6. Department of Chemistry University of Michigan
7. Yale Quantum Institute Yale University

Abstract

We introduce QFlux, an open-source Python package for simulating quantum dynamics in chemical systems across multiple levels of theory using a unified classical and quantum computational framework. QFlux integrates deterministic wavefunction propagation, Lindblad master equations, generalized quantum master equations (GQMEs), operator-splitting techniques, and variational quantum algorithms within a modular architecture designed for systematic benchmarking and hybrid classical-quantum execution. The framework supports both closed-and open-system dynamics, including Markovian and non-Markovian regimes, and provides explicit mappings between classical propagators and quantum circuit constructions executable on NISQ hardware. Built atop established scientific and quantum software ecosystems, QFlux emphasizes reproducibility, interoperability, and controlled cross-validation between numerical approximations and hardware-oriented algorithms. Part I in this series of papers presents the theoretical foundations, software design principles, and methodological scope of QFlux, and positions it within the broader landscape of classical, tensor-network, and quantum-circuit-based simulation platforms. The series is designed to serve as

a teaching reference for graduate students, a practical guide for researchers implementing custom quantum simulations, and a foundational reference for the broader QFlux ecosystem.

QFlux: Classical Foundations for Quantum Dynamics Simulation. Part I – Building Intuition and Computational Workflows

Brandon C. Allen,[†] Xiaohan Dan,[†] Delmar G. A. Cabral,[†] Nam P. Vu,[†,‡,¶] Cameron Cianci,[§] Alexander V. Soudackov,[†] Rishab Dutta,[†] Sabre Kais,[‖] Eitan Geva,[⊥] and Victor S. Batista[*,†,#]

[†]*Department of Chemistry, Yale University, New Haven, CT 06520, USA*

[‡]*Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA*

[¶]*Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, MA 02139, USA*

[§]*Department of Physics, University of Connecticut, Storrs, CT 06268, USA*

[‖]*Department of Electrical and Computer Engineering, Department of Chemistry, North Carolina State University, Raleigh, North Carolina 27606, USA*

[⊥]*Department of Chemistry, University of Michigan, Ann Arbor, MI 48109, USA*

[#]*Yale Quantum Institute, Yale University, New Haven, CT 06511, USA*

E-mail: victor.batista@yale.edu

Abstract

We introduce `QFlux`, an open-source Python package for simulating quantum dynamics in chemical systems across multiple levels of theory using a unified classical and quantum computational framework. `QFlux` integrates deterministic wavefunction propagation, Lindblad master equations, generalized quantum master equations (GQMEs), operator-splitting techniques, and variational quantum algorithms within a modular architecture designed for systematic benchmarking and hybrid classical-quantum execution. The framework supports both closed- and open-system dynamics, including Markovian and non-Markovian regimes, and provides explicit mappings between classical propagators and quantum circuit constructions executable on NISQ hardware. Built atop established scientific and quantum software ecosystems, `QFlux` emphasizes reproducibility, interoperability, and controlled cross-validation between numerical approximations and hardware-oriented algorithms. Part I in this series of papers presents the theoretical foundations, software design principles, and methodological scope of `QFlux`, and positions it within the broader landscape of classical, tensor-network, and quantum-circuit-based simulation platforms. The series is designed to serve as a teaching reference for graduate students, a practical guide for researchers implementing custom quantum simulations, and a foundational reference for the broader QFlux ecosystem.

1 Introduction

Accurate simulation of quantum dynamics is a cornerstone of chemical physics, linking microscopic quantum motion to experimentally accessible observables. Phenomena such as vibrational coherence, electronic energy transfer, nonadiabatic transitions, and environmentally induced decoherence are unified by a single mechanism: the time evolution of quantum amplitudes and phases. Through correlation functions and response properties, this evolution shapes spectroscopic signatures and governs dynamical mechanisms relevant to chemical

reactivity, from ultrafast excitonic transport in light-harvesting assemblies to long-time relaxation and thermalization in condensed phases. Quantitative simulation, however, quickly becomes difficult as Hilbert spaces expand with system size and as environmental couplings introduce additional degrees of freedom, competing timescales, and memory effects. Quantum dynamics methods can be viewed as a hierarchy of physical descriptions, each defined by controlled assumptions with clear numerical consequences. At the most fundamental level, closed-system dynamics follow the time-dependent Schrödinger equation, where direct propagation provides high-fidelity reference results for small to moderate Hilbert spaces. In realistic chemical environments, coupling to vibrational, solvent, or phononic modes induces dissipation and decoherence, motivating reduced descriptions in terms of density matrices and quantum master equations. Under weak coupling and negligible memory, Markovian dynamics are accurately captured by Gorini–Kossakowski–Sudarshan–Lindblad generators, which ensure complete positivity and trace preservation.[1–3] When these assumptions fail-owing to structured spectral densities, slow bath modes, strong coupling, or low temperatures-non-Markovian effects become essential, and projection-operator approaches such as Nakajima–Zwanzig naturally yield generalized quantum master equations (GQME) with explicit memory kernels.[4–7] In parallel, quantum information science has produced algorithmic approaches for simulating dynamics that, in principle, mitigate classical scaling bottlenecks.[8,9] In the noisy intermediate-scale quantum (NISQ) regime,[10] however, near-term devices remain constrained by finite coherence times, circuit-depth limits, and sampling overhead. Practical simulation workflows are therefore hybrid: classical computation remains indispensable for model construction, baseline validation, and error analysis, while quantum circuits are used selectively to implement dynamical primitives-approximate unitary evolution, dilations of non-unitary channels, or variational updates-under realistic hardware constraints.[11,12] Despite rapid progress in both classical and quantum approaches, the software ecosystem remains fragmented across these regimes. QuTiP provides mature operator representations and classical propagation tools for wavefunctions and master equations.[13] Tensor-network frame-

works exploit compressibility and low-entanglement structure to enable scalable reduced dynamics.[14] Quantum circuit toolkits supply compilation and execution pipelines for unitary simulation and variational algorithms.[15] What is often missing is a unified environment in which a physical model is specified once, propagated across multiple dynamical descriptions under consistent conventions, and benchmarked systematically across classical and quantum backends. To address this gap, we introduce **QFlux**,[16] an open-source Python framework for multilevel quantum dynamics in chemical systems. QFlux adopts a model-centric philosophy: the same physical Hamiltonian, initial state, and observables should remain portable across levels of theory and computational paradigms. Accordingly, QFlux supports closed-system Schrödinger dynamics, Markovian Lindblad evolution, non-Markovian GQME propagation, and multiple hardware-motivated simulation strategies within a shared workflow. Its implementation builds on established numerical libraries (NumPy and SciPy[17,18]), classical quantum dynamics tooling (QuTiP[13]), tensor-network capabilities (MPSQD[14]), and quantum circuit infrastructure (Qiskit[15]). A central design goal of QFlux is interoperability without sacrificing methodological breadth. The same Hamiltonian model can be propagated using deterministic ODE solvers, split-operator Fourier transform (SOFT) methods for grid-based wavepacket dynamics,[19] product-formula Hamiltonian simulation via Trotter–Suzuki decompositions,[20,21] randomized compilation strategies such as qDRIFT,[22] or variational quantum dynamics based on time-dependent variational principles.[23,24] For open-system dynamics, QFlux pairs classical master-equation solvers with circuit-oriented constructions based on Kraus representations and Stinespring dilations, enabling controlled comparisons between reduced non-unitary dynamics and their unitary embeddings on extended Hilbert spaces.[25,26] Beyond providing implementations, QFlux is explicitly structured as a benchmarking and research platform. Identical physical models can be propagated using multiple formalisms and backends, enabling systematic assessment of discretization error, Trotterization error, memory-kernel truncation, variational expressivity limits, and hardware-induced noise. This benchmark-centric perspective is particularly important in the NISQ era, where accuracy

must be evaluated alongside circuit depth, compilation overhead, and sampling cost.[10] This tutorial series develops QFlux through a deliberately pedagogical progression. **Part I** establishes the theoretical and numerical foundations of time-dependent quantum dynamics using classical propagation methods, emphasizing physical interpretation and cross-validation. **Part II** translates these ideas to closed-system quantum simulations on qubit-based hardware. **Part III** focuses on state preparation and unitary decomposition. **Part IV** extends the framework to open quantum systems using Lindblad dynamics and dilation techniques. **Part V** introduces adaptive variational algorithms tailored to near-term hardware. Finally, **Part VI** addresses non-Markovian dynamics through generalized quantum master equations with explicit memory effects. QFlux is not presented as a source of fundamentally new quantum dynamics algorithms. Rather, its contribution lies in unifying, validating, and systematically comparing established methods within a single, consistent framework. QFlux does not replace classical propagators, tensor-network techniques, or quantum-circuit primitives available in packages such as QuTiP, MPSQD, and Qiskit. Instead, it enforces a common abstraction in which the same physical Hamiltonian, initial state, and observables are propagated unchanged across classical wavefunction dynamics, open-system descriptions, tensor-network representations, and quantum-ready formulations. This design enables benchmark symmetry, allowing numerical, physical, and hardware-motivated approximations to be compared directly and their errors disentangled in a controlled manner.

2 QFlux: Scope, Design Philosophy, and Capabilities

QFlux is designed as a general-purpose research and benchmarking environment for quantum dynamics simulations in chemical and molecular systems (Fig. 1). Rather than privileging a single propagation strategy, QFlux is organized to make systematic comparisons routine: users can hold the physical model fixed while varying the dynamical description (closed vs. open, Markovian vs. non-Markovian) and the numerical or hardware-motivated propagation

scheme.

At the workflow level, QFlux centers on four reusable components:

- *model specification* (Hamiltonians, dissipators, bath structure, and kernel inputs);
- *state preparation* (pure states, mixed states, and thermally purified constructions);
- *time propagation* (interchangeable classical propagators and quantum-ready methods);
- *analysis* (expectation values, correlation functions, populations/coherences, and spectral observables).

This modular structure mirrors standard practice in quantum dynamics while keeping the computational backend-dense linear algebra, tensor networks, or quantum circuits-an implementation choice rather than a conceptual constraint.

QFlux builds on widely used libraries for numerical computing and quantum simulation. NumPy and SciPy provide array primitives, linear algebra routines, FFTs, and robust ODE solvers.[17,18] QuTiP supplies operator and state abstractions together with mature classical propagation utilities.[13] Qiskit provides circuit construction, transpilation, and access to IBM Quantum backends.[15] Where scalability is critical, QFlux incorporates tensor-train (matrix product state) representations through MPSQD,[14,27] enabling compressed propagation in selected regimes. The intent is not to replace these ecosystems, but to provide a coherent interface layer that makes them interoperable within a single simulation workflow. In contrast to workflows based on QuTiP augmented by method-specific scripts, QFlux enforces a strictly model-centric abstraction in which the same Hamiltonian, initial state, and observables are propagated unchanged across classical solvers, tensor-network methods, and quantum-circuit–ready algorithms. This *benchmark symmetry* enables controlled, apples-to-apples comparisons that disentangle physical approximations, numerical error, and hardware-driven constraints within a single reproducible framework.

Methodologically, QFlux spans a hierarchy of dynamical descriptions. Closed-system dynamics are supported via direct Schrödinger propagation and operator-splitting approaches

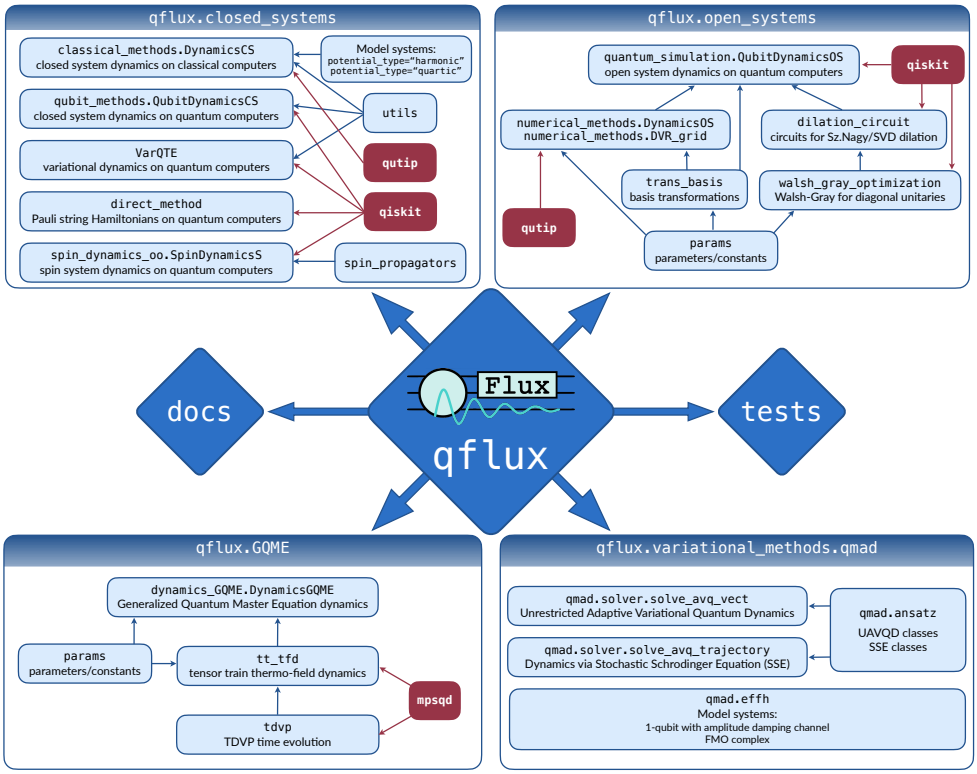

Figure 1: Complete package tree of `QFlux`. The diagram shows the organization of the codebase rooted at the top-level `qflux` package. The APIs are contained within four core modules `closed_systems`, `open_systems`, `GQME`, `variational_methods`, and testing and documentation are isolated in `tests` and `docs`. Directed edges indicate containment and dependency relationships, and red boxes indicate the external packages.

such as SOFT.[19] Markovian open-system dynamics are treated using Lindblad master equations,[1–3] paired where appropriate with circuit-oriented unitary embeddings.[25,26] Non-Markovian reduced dynamics are supported through GQME workflows rooted in projection-operator theory.[4–6] Hardware-motivated strategies for unitary evolution include Trotter–Suzuki product formulas,[20,21] randomized decompositions such as qDRIFT,[22] and variational quantum dynamics based on McLachlan-type time-dependent variational principles.[23,24] Together, these capabilities enable consistent exploration of accuracy–cost tradeoffs across classical and quantum paradigms.

A defining feature of QFlux is its emphasis on validation. By allowing the same model to be propagated using multiple solvers, QFlux makes it straightforward to isolate physical assumptions (e.g., Markovianity), numerical approximations (time-step and splitting errors), and hardware-driven limitations (sampling noise and compilation constraints). This capability is particularly important for near-term quantum simulation studies, where reproducibility and cross-verification against trusted classical references are often the limiting factors in interpreting hardware results.[10]

In this tutorial series, QFlux serves as the unifying framework through which propagation methods are introduced, tested, and interpreted. Part I emphasizes foundations-closed-system propagation, operator structure, and benchmarking-to build intuition and establish reference workflows. These same operator decompositions and validation strategies on conventional classical computers then provide a direct conceptual bridge to the quantum circuit constructions and hybrid quantum–classical simulations developed in subsequent parts.

3 Workflow for Quantum Dynamics Simulations

Regardless of whether a simulation runs on a laptop or a quantum processor, the QFlux workflow is the same: initialization, time propagation, and analysis (Fig. 2).

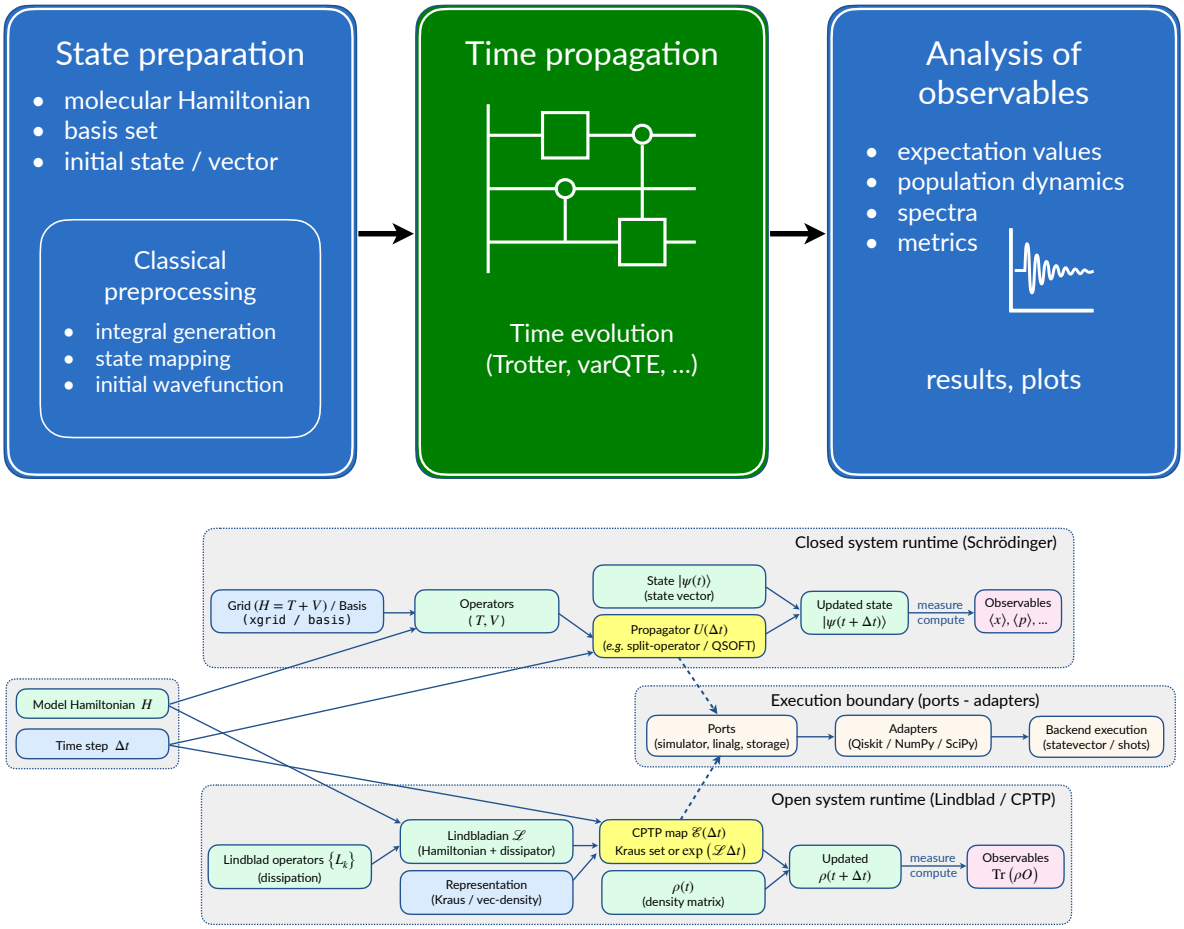

Figure 2: Top: QFlux workflow for quantum dynamics simulations: state preparation, time propagation, and analysis of observables. Color-coded modules show classical (blue) and quantum-ready (green) components. Bottom: Runtime data flow in QFlux for closed and open quantum systems. For closed systems (Bottom lane), the state vector $|\psi(t)\rangle$ is propagated by a time-step propagator $U(\Delta t)$ constructed from discretized kinetic and potential operators derived from the Hamiltonian and grid/basis representation. For open systems (top lane), the density operator $\rho(t)$ evolves under a completely positive trace-preserving (CPTP) map $\mathcal{E}(\Delta t)$ obtained from the Liouvillian, using either Kraus or vectorized-density representations. In both cases, time evolution is simulated through abstract ports that dispatch to concrete numerical or quantum backends (e.g., NumPy, SciPy, Qiskit), and observables are evaluated from the updated state at each time step.

1. **Initialization:** Define the system and choose a representation – grid, Fock basis, or harmonic oscillator basis. For example, the ground vibrational state of a diatomic molecule can be represented as a Gaussian (coherent-state) wavepacket centered near equilibrium.

2. **Propagation:** Evolve the state under the chosen Hamiltonian using a suitable numerical

method, such as a ODE solver, or SOFT for efficiency on uniform grids. In both cases, the formal propagator $U(t) = \mathrm{e}^{-\mathrm{i}Ht/\hbar}$ is realized either by direct integration of the TDSE or by short-time operator splitting.

3. Analysis: Compute observables such as $\langle x(t)\rangle$, $\langle p(t)\rangle$, populations in a chosen basis, and correlation functions like the survival amplitude $\xi(t) = \langle\psi(0)|\psi(t)\rangle$, which connects directly to spectroscopy.

4 Dynamics of Pure Quantum States

The time-dependent Schrödinger equation (TDSE),

$$\mathrm{i}\hbar\frac{\partial}{\partial t}|\psi(t)\rangle = H|\psi(t)\rangle, \tag{4.1}$$

governs the time evolution of a pure quantum state described by the wavefunction $|\psi(t)\rangle$ under the action of the Hamiltonian H. When the Hamiltonian is time-independent, the formal solution can be written in closed form as

$$|\psi(t)\rangle = \mathrm{e}^{-\mathrm{i}Ht/\hbar}|\psi(0)\rangle, \tag{4.2}$$

where $\mathrm{e}^{-\mathrm{i}Ht/\hbar}$ is the unitary time-evolution operator. Consequently, as introduced in Sec. 3, the dynamics simulation may be viewed as a two-step procedure: preparation of an initial state $|\psi(0)\rangle$ (i.e., initialization), followed by its propagation in time through application of the evolution operator $\mathrm{e}^{-\mathrm{i}Ht/\hbar}$ (i.e., time-propagation). This perspective underlies numerical propagation schemes with both conventional computers and their quantum-circuit-based counterparts.

Hamiltonians in molecular problems. For molecular systems, the Hamiltonian $H = T + V$ combines kinetic and potential energy operators that govern nuclear and electronic

motion. This split is not just notation: it is what makes SOFT possible (Sec. 5.2).

From states to observables. From $|\psi(t)\rangle$, measurable observables follow as expectation values

$$\langle O(t)\rangle = \langle\psi(t)|O|\psi(t)\rangle. \tag{4.3}$$

In practice, tracking a small set of observables is also a numerical check: conserved norm and physically sensible $\langle x(t)\rangle$ and $\langle p(t)\rangle$ are early indicators that the simulation is behaving correctly.

From dynamics to spectra. A particularly useful correlation function is the *survival amplitude*,

$$\xi(t) = \langle\psi(0) \mid \psi(t)\rangle, \tag{4.4}$$

whose Fourier transform yields the photoabsorption spectrum, directly connecting microscopic time evolution with spectroscopic observables.

Wavepacket intuition. Intuitively, $|\psi(t)\rangle$ can be pictured as a wavepacket moving across a potential energy landscape. Tracking its motion reveals vibrational oscillations, tunneling, and coherence – behaviors that determine chemical reactivity and spectroscopy.

5 Classical Simulations of Quantum Dynamics

In the following sections, we separate physical interpretation, numerical algorithms, and software implementation to clarify how each layer contributes to a reliable quantum-dynamics workflow.

5.1 ODE solver Integration

From a physics perspective, expanding the wavefunction in a chosen basis converts the Schrödinger equation into coupled equations of motion for probability amplitudes. This formulation makes conservation laws, phase relationships, and analytical benchmarks explicit and readily accessible.

For low-dimensional quantum systems, the time-dependent Schrödinger equation (TDSE) can be integrated efficiently on classical computers by expanding the state in a fixed basis $\{|j\rangle\}$,

$$|\psi(t)\rangle = \sum_j c_j(t)|j\rangle, \tag{5.1}$$

with initial amplitudes $c_j(0) = \langle j|\psi(0)\rangle$. Substitution into the TDSE yields a system of coupled linear ordinary differential equations,

$$\frac{dc_k}{dt} = -\frac{i}{\hbar}\sum_j \langle k|\hat{H}|j\rangle\, c_j(t), \tag{5.2}$$

which can be propagated numerically using high-order adaptive integrators such as Runge–Kutta methods,[28–32] VODE,[33] CVODE,[34] and related multistep schemes.[35–37] Implementations in packages such as SciPy (RK45, DOP853, LSODA)[18] and QuTiP (vern7, vern9, Adams, BDF)[38,39] employ adaptive step-size control to maintain prescribed error tolerances, enabling accurate propagation in the presence of fast oscillations and widely separated timescales.

The choice of basis – such as a truncated Fock basis or any other convenient representation – determines the dimensionality and numerical efficiency of the resulting propagation but does not alter the underlying physical description.

Benchmark: harmonic oscillator. The harmonic oscillator, described by the following Hamiltonian:

$$\hat{H} = \frac{p^2}{2m} + V(x), \tag{5.3}$$

with $V(x) = \frac{1}{2}m\omega^2 x^2$, serves as a stringent benchmark since its analytical solution is known, with time-dependent expectation values of position and momentum:

$$\begin{aligned} \langle x(t)\rangle &= x_0 \cos(\omega t) + \frac{p_0}{m\omega}\sin(\omega t) \\ \langle p(t)\rangle &= -m\omega x_0 \sin(\omega t) + p_0 \cos(\omega t) \end{aligned} \tag{5.4}$$

Matching Eq. (5.4) is a useful first diagnostic because it checks the full pipeline: operator construction, state preparation, propagation, and expectation value evaluation.

What to look for in the numerical result. A correct propagation reproduces (i) the oscillation frequency ω, (ii) the correct phase relationship between $\langle x(t)\rangle$ and $\langle p(t)\rangle$, and (iii) constant norm. If any of these fail, refine timestep and basis size before moving on to more complex models.

Additional implementation details are provided in the Supporting Information, including a description of Scipy's RK45 (Section S.1).[18] As a hands-on implementation, Script S.2.1 illustrates the simulation workflow by using a ODE solver with QuTiP,[38,39] which can be accessed through the **QFlux** framework.[40] The script initializes a harmonic oscillator, constructs its Hamiltonian, and propagates the wavefunction in time using a ODE solver. The resulting time evolution can then be analyzed to compute expectation values of observables and compared with the exact analytical expressions, as shown in Script S.2.2.

Fig. 3 shows excellent agreement between the numerical propagation and the analytical results for both $\langle x(t)\rangle$ and $\langle p(t)\rangle$, confirming the accuracy of the ODE solver integration implemented in QuTiP. This example establishes a solid foundation for studying more complex quantum systems within the same QFlux framework.

5.2 Split-Operator Fourier Transform (SOFT)

The SOFT method[41–44] propagates quantum states by alternating between potential and kinetic operators, using fast Fourier transforms (FFTs) to switch between position and mo-

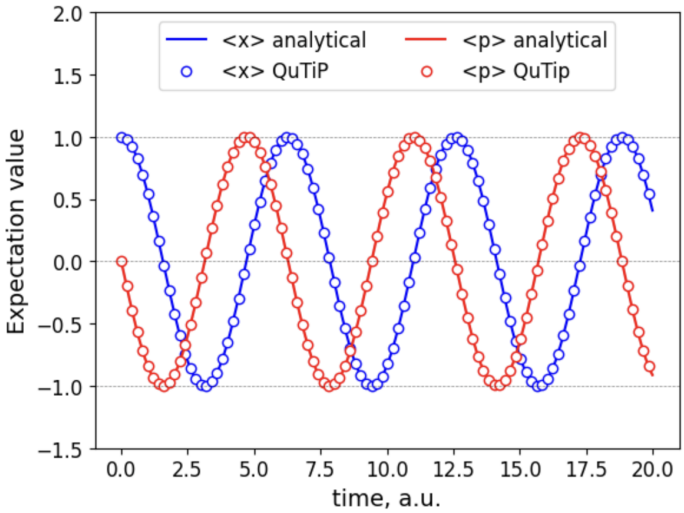

Figure 3: Time-dependent expectation values of position and momentum for the harmonic oscillator, computed using a ODE solver integrator in QuTiP. The results show excellent agreement with the analytical expressions in Eq. (5.4).

mentum representations. One full SOFT time step of size $\tau = t_{i+1} - t_i$ reads

$$\psi(x, t_{i+1}) = \mathrm{e}^{\frac{-\mathrm{i}V(x)\tau}{2\hbar}} \mathcal{F}^{-1}\left[\mathrm{e}^{\frac{-\mathrm{i}p^2\tau}{2m\hbar}} \mathcal{F}\left(\mathrm{e}^{\frac{-\mathrm{i}V(x)\tau}{2\hbar}} \psi(x, t_i)\right)\right], \tag{5.5}$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ denote the Fourier and inverse Fourier transforms, respectively. From an algorithmic perspective, each SOFT time step consists of alternating diagonal operations in position and momentum space, with fast Fourier transforms providing the basis changes at $\mathcal{O}(N \log N)$ cost.

How to read Eq. (5.5). It helps to interpret the SOFT step as a sequence of unitary substeps: (i) apply a half-step potential phase in x-space, (ii) transform to momentum space, (iii) apply a kinetic phase in p-space, (iv) transform back, (v) apply the second half-step potential phase. This picture will be reused directly in Part II when mapping propagation to quantum circuits.

Benchmark against ODE solver and analytics. From a benchmarking perspective, the split-operator Fourier transform (SOFT) method should reproduce the same physical

observables as high-order ODE solvers when both approaches are numerically converged. Agreement between the two therefore serves as a validation of numerical correctness, rather than as a comparison of algorithmic performance.

Indeed, Fig. 4 demonstrates that properly converged SOFT and ODE solver simulations yield indistinguishable results, in excellent agreement with the analytical expressions for the time-dependent expectation values of position and momentum of the harmonic oscillator described in Section 5.1. This benchmark validates the SOFT implementation and establishes it as a reliable reference before extending the analysis to anharmonic systems, such as asymmetric double-well potentials relevant to proton-transfer dynamics.

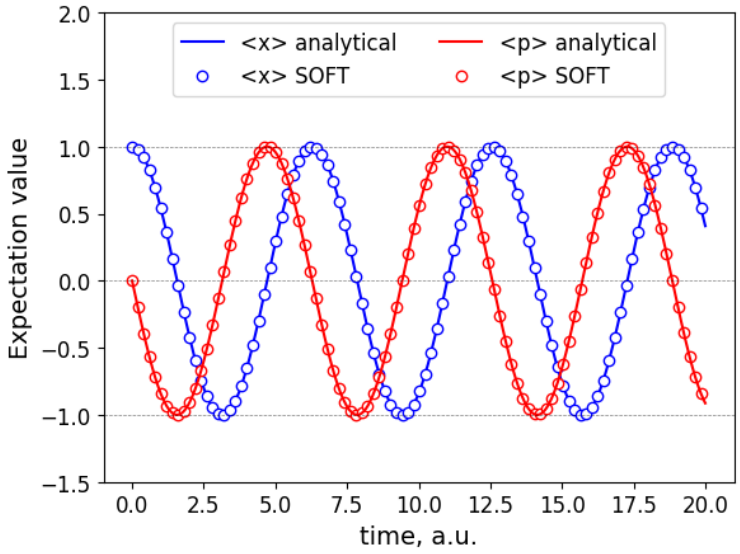

Figure 4: Comparison of time-dependent expectation values of position and momentum obtained from SOFT propagation and analytical solutions for the harmonic oscillator.

When SOFT is the right tool. From a software-workflow perspective, SOFT is particularly efficient for grid-based representations and integrates naturally into QFlux as an interchangeable propagator alongside ODE solvers. On uniform grids, SOFT is often faster and more memory-efficient, whereas ODE solver methods provide fine-grained adaptive time-step control and can be more convenient in operator-diagonal or compact basis representations.

In practice, the choice between these approaches is guided by (i) the underlying repre-

sentation (uniform grids versus compact bases), (ii) the desired mode of error control (fixed versus adaptive time stepping), and (iii) how naturally the propagation scheme maps onto a quantum implementation. Care must be taken to control grid resolution, boundary conditions, and spectral aliasing in SOFT calculations, as inadequate grids or poorly chosen boundaries can introduce nonphysical artifacts.

Together, ODE solvers and SOFT provide complementary classical references: the former emphasizes numerical control, while the latter exposes operator structure that anticipates quantum-circuit implementations.

Takeaway statement: Split-Operator Fourier Transform (SOFT)

What insight have we gained? At this point, the reader should understand why SOFT reproduces exact dynamics when converged, and how its operator structure anticipates quantum circuit layouts.

5.3 Connecting Classical and Quantum Simulations

As we will see in Part II, the SOFT method forms a conceptual bridge to quantum computing. From a quantum-computing perspective, the SOFT sequence directly mirrors quantum-circuit structure: diagonal operators correspond to phase rotations, and Fourier transforms correspond to quantum Fourier transforms (QFT). This analogy makes SOFT an ideal pedagogical and practical on-ramp to quantum algorithms: one can prototype classically, verify against analytical results, and then translate to qubit-based propagation within the QFlux framework.

The following section introduces more advanced techniques required for finite-temperature simulations and systems with many degrees of freedom. Readers primarily interested in pure-state dynamics may skip ahead to Sec. 7 without loss of continuity.

6 Quantum Dynamics Simulations of Mixed States

Realistic chemical systems rarely evolve in isolation. Instead, they are typically embedded in an environment-such as a solvent, lattice, or radiation field-that exchanges energy with the system and maintains it at a finite temperature. Under these conditions, the system generally occupies a statistical ensemble of quantum states, and its dynamics can no longer be described by a single wave function evolving unitarily according to the time-dependent Schrödinger equation. While the propagation schemes introduced in the previous sections are therefore adequate for pure quantum states, finite-temperature and open-system effects call for a more general description in terms of mixed states, represented by density matrices rather than statevectors.

As described in the following subsections, Thermo-Field Dynamics (TFD) provides a numerically exact and efficient framework that maps the thermal density matrix onto a pure thermal state that evolves under the time-dependent Schrödinger equation in an enlarged, doubled Hilbert space. This construction allows one to retain standard wave-function propagation techniques, which in practice are typically combined with low-rank tensor-train (TT) or matrix-product-state (MPS) representations of the evolving density matrix, as discussed below.[44–46]

6.1 Motivation and background

For a system with Hamiltonian $\hat{H}$, the canonical thermal density matrix at inverse temperature $\beta = (k_B T)^{-1}$ is

$$\hat{\rho}(0;\beta) = Z_\beta^{-1} \mathrm{e}^{-\beta \hat{H}}, \qquad Z_\beta = \mathrm{Tr}\left[\mathrm{e}^{-\beta \hat{H}}\right]. \tag{6.1}$$

Although this form is compact, it is not directly amenable to the wavefunction-based propagation methods introduced in previous sections of this tutorial because it describes a *mixed* state that does not evolve according to the Schrödinger equation but according to the quan-

tum Liouville equation:

$$\frac{\partial \hat{\rho}(t)}{\partial t} = -\frac{\mathrm{i}}{\hbar}[\hat{H}, \hat{\rho}(t)]. \tag{6.2}$$

Thermo-Field Dynamics (TFD) addresses this by enlarging the system with fictitious variables and forming a pure thermal wavefunction. This state evolves according to a time-dependent Schrödinger equation which is equivalent to Eq. (6.2) once the fictitious variables are traced out, as shown in subsequent subsections.

6.2 Thermal wavefunction construction

The purified thermal wavefunction is defined as

$$|\psi(0;\beta)\rangle = Z_\beta^{-1/2} \sum_n \mathrm{e}^{-\beta E_n/2} \, |n, \tilde{n}\rangle, \tag{6.3}$$

where $\{|n\rangle\}$ are eigenstates of $\hat{H}$ with energies E_n, and $|\tilde{n}\rangle$ are the corresponding states in the auxiliary space. Equivalently, this can be written as

$$|\psi(0;\beta)\rangle = Z_\beta^{-1/2} \mathrm{e}^{-\beta \hat{H}/2} \sum_n |n, \tilde{n}\rangle. \tag{6.4}$$

Example: harmonic oscillator. For a single bosonic mode with creation operators $a^\dagger$ and $\tilde{a}^\dagger$, the thermal wavefunction becomes

$$\begin{aligned} |\psi(0;\beta)\rangle &= Z_\beta^{-1/2} \sum_n \mathrm{e}^{-\beta n\omega/2} \frac{1}{n!} (a^\dagger \tilde{a}^\dagger)^n |0_{\mathrm{vib}}, \tilde{0}_{\mathrm{vib}}\rangle \\ &= \sqrt{1 - \mathrm{e}^{-\beta\omega}} \, \exp\left[\mathrm{e}^{-\beta\omega/2} a^\dagger \tilde{a}^\dagger\right] |0_{\mathrm{vib}}, \tilde{0}_{\mathrm{vib}}\rangle. \end{aligned} \tag{6.5}$$

This exponential form shows that each physical mode is entangled with its fictitious partner, encoding the correct thermal occupation $n(\omega, \beta) = 1/(\mathrm{e}^{\beta\omega} - 1)$.

6.3 Thermal ensemble averages

Expectation values of observables at temperature T can be computed directly as pure-state averages:

$$\begin{aligned}\langle \hat{F} \rangle_\beta &= \langle \psi(0;\beta)|\hat{F}|\psi(0;\beta)\rangle \\ &= Z_\beta^{-1} \sum_{n,m} \mathrm{e}^{-\beta(E_n+E_m)/2} \langle n|\hat{F}|m\rangle\langle \tilde{n}|\tilde{m}\rangle \\ &= Z_\beta^{-1} \sum_n \mathrm{e}^{-\beta E_n} \langle n|\hat{F}|n\rangle, \end{aligned} \tag{6.6}$$

which recovers the standard canonical ensemble result $\langle \hat{F} \rangle_\beta = \mathrm{Tr}[\hat{\rho}(0;\beta)\hat{F}]$.

6.4 Thermal density matrix recovery

Tracing out the fictitious subsystem returns the physical mixed density operator:

$$\begin{aligned}\mathrm{Tr}_{\tilde{f}}\left[|\psi(0;\beta)\rangle\langle\psi(0;\beta)|\right] &= \mathrm{Tr}_{\tilde{f}}\Big[Z_\beta^{-1}\mathrm{e}^{-\beta\hat{H}} \sum_n |n,\tilde{n}\rangle\langle n,\tilde{n}|\Big] \\ &= Z_\beta^{-1}\mathrm{e}^{-\beta\hat{H}} = \hat{\rho}(0;\beta). \end{aligned} \tag{6.7}$$

Hence, the purified wavefunction faithfully reproduces the thermal density matrix upon tracing over the auxiliary space.

6.5 Dynamics: the TF Schrödinger equation

After preparing $|\psi(0;\beta)\rangle$, its time evolution is governed by the *thermo-field Schrödinger equation*:

$$\frac{\partial\,|\psi(\beta,t)\rangle}{\partial t} = -\frac{\mathrm{i}}{\hbar}\,\bar{H}\;|\psi(\beta,t)\rangle\,, \tag{6.8}$$

where the doubled-space Hamiltonian operator is

$$\bar{H} = \hat{H}\otimes\tilde{I} - I\otimes\tilde{H}. \tag{6.9}$$

Here, $\hat{H}$ acts on the physical system and $\tilde{H}$ on its fictitious counterpart. This form ensures that time evolution in the doubled Hilbert space preserves the correct thermal correlations.

6.6 TF Liouville equation: recovering the physical dynamics

In the thermo-field (TF) construction, the physical open-system system–bath Hilbert space

$$\mathcal{H}_{\text{phys}} = \mathcal{H}_S \otimes \mathcal{H}_B \tag{6.10}$$

is embedded into an enlarged space

$$\mathcal{H}_{\text{tot}} = \mathcal{H}_S \otimes \mathcal{H}_B \otimes \tilde{\mathcal{H}}, \tag{6.11}$$

where $\tilde{\mathcal{H}}$ contains all fictitious degrees of freedom introduced to purify the thermal state. For the open–system system–bath model we consider two natural TF choices:

- *Bath-only doubling:* $\tilde{\mathcal{H}} = \tilde{\mathcal{H}}_B$ (only the bath is doubled),
- *Full system–bath doubling:* $\tilde{\mathcal{H}} = \tilde{\mathcal{H}}_S \otimes \tilde{\mathcal{H}}_B$ (both the system and the bath are doubled).

In either case, the physical density matrix at time t is obtained by tracing out all fictitious degrees of freedom,

$$\hat{\rho}(t) = \text{Tr}_{\tilde{f}} \left[|\psi(\beta, t)\rangle\langle\psi(\beta, t)| \right], \tag{6.12}$$

where the trace is taken over whichever fictitious Hilbert space is present.

The purified TF state evolves under the TF Schrödinger equation

$$\mathrm{i}\hbar \frac{\partial \, |\psi(\beta, t)\rangle}{\partial t} = \bar{H} \; |\psi(\beta, t)\rangle \, , \tag{6.13}$$

with a TF Hamiltonian of the form

$$\bar{H} = H_{\text{SB}} \otimes \tilde{I} - I_{\text{SB}} \otimes \tilde{H}, \tag{6.14}$$

where H_{SB} is the physical open–system Hamiltonian and $\tilde{H}$ acts only on the fictitious space $\tilde{\mathcal{H}}$. For bath-only doubling one has $\tilde{H} = \tilde{H}_B$, while for full doubling $\tilde{H} = \tilde{H}_{\mathrm{SB}}$; the derivation below is identical for both constructions.

Differentiating Eq. (6.12) and using Eq. (6.13) gives

$$\frac{\partial \hat{\rho}(t)}{\partial t} = \mathrm{Tr}_{\tilde{f}}\left[-\frac{\mathrm{i}}{\hbar}\bar{H}\,|\psi\rangle\langle\psi| + \frac{\mathrm{i}}{\hbar}\,|\psi\rangle\langle\psi|\,\bar{H}\right] = -\frac{\mathrm{i}}{\hbar}\,\mathrm{Tr}_{\tilde{f}}\left([\bar{H}, |\psi\rangle\langle\psi|]\right). \tag{6.15}$$

Inserting the decomposition of Eq. (6.14),

$$[\bar{H}, |\psi\rangle\langle\psi|] = [H_{\mathrm{SB}} \otimes \tilde{I}, |\psi\rangle\langle\psi|] - [I_{\mathrm{SB}} \otimes \tilde{H}, |\psi\rangle\langle\psi|], \tag{6.16}$$

the partial trace separates into two contributions:

$$\frac{\partial \hat{\rho}(t)}{\partial t} = -\frac{\mathrm{i}}{\hbar}\Big\{\mathrm{Tr}_{\tilde{f}}\Big([H_{\mathrm{SB}} \otimes \tilde{I}, |\psi\rangle\langle\psi|]\Big) - \mathrm{Tr}_{\tilde{f}}\Big([I_{\mathrm{SB}} \otimes \tilde{H}, |\psi\rangle\langle\psi|]\Big)\Big\}. \tag{6.17}$$

Physical Hamiltonian term. Since H_{SB} acts only on the physical space and commutes with operators on $\tilde{\mathcal{H}}$,

$$\mathrm{Tr}_{\tilde{f}}\Big([H_{\mathrm{SB}} \otimes \tilde{I}, |\psi\rangle\langle\psi|]\Big) = H_{\mathrm{SB}}\,\hat{\rho}(t) - \hat{\rho}(t)\,H_{\mathrm{SB}} = [H_{\mathrm{SB}}, \hat{\rho}(t)]. \tag{6.18}$$

Fictitious Hamiltonian term. Because $\tilde{H}$ acts only on $\tilde{\mathcal{H}}$ and the trace is cyclic in the fictitious space,

$$\mathrm{Tr}_{\tilde{f}}\Big[(I_{\mathrm{SB}} \otimes \tilde{H})X\Big] = \mathrm{Tr}_{\tilde{f}}\Big[X(I_{\mathrm{SB}} \otimes \tilde{H})\Big] \quad \text{for any operator } X, \tag{6.19}$$

the second contribution vanishes:

$$\mathrm{Tr}_{\tilde{f}}\Big([I_{\mathrm{SB}} \otimes \tilde{H}, |\psi\rangle\langle\psi|]\Big) = 0. \tag{6.20}$$

Importantly, this cancellation holds regardless of whether the system is doubled: the argument uses only the cyclicity of the partial trace over the fictitious space.

Result. Combining Eqs. (6.17)–(6.20) yields

$$\frac{\partial \hat{\rho}(t)}{\partial t} = -\frac{\mathrm{i}}{\hbar}[H_{\mathrm{SB}}, \hat{\rho}(t)], \tag{6.21}$$

which is precisely the Liouville–von Neumann equation for the physical spin–boson density matrix.

Remark. Equation (6.21) is recovered independently of whether the system is doubled in the TF construction. Thus, unless the system itself is thermalized, for the open–system model one may freely use the simpler bath-only doubling—as typically done in practical TT-TFD simulations—without affecting the physical reduced dynamics.

6.7 Tensor-train (TT/MPS) representation

In practice, the composite space (system $\otimes$ bath $\otimes$ tilde bath) becomes exponentially large, and the pure state $|\psi(\beta, t)\rangle$ is efficiently represented as a tensor train (MPS):

$$|\psi(\beta, t)\rangle \simeq \sum_{\{i_k\}} A^{[1]}_{i_1} A^{[2]}_{i_2} \cdots A^{[N]}_{i_N} |i_1 i_2 \cdots i_N\rangle. \tag{6.22}$$

The doubled Hamiltonian $\bar{H}$ is encoded as a matrix product operator (MPO). Time propagation is performed using the time-dependent variational principle (TDVP)[47] or time-evolving block decimation (TEBD) algorithms with adaptive bond-dimension control.

6.8 Test Case: Qubit Coupled to a Bath of Harmonic Oscillators

In this subsection, we describe the implementation of the TT-TFD approach as applied to simulations of finite-temperature wavepacket dynamics. As an illustrative example, we

consider population dynamics of a qubit coupled to bath of quantum harmonic oscillators.

6.8.1 Algorithmic workflow

In the specific case of a harmonic bath (*i.e.,* a bath composed of harmonic oscillators), the algorithmic workflow is as follows:

1. **Discretize the bath:** Choose $\{\omega_k, c_k\}$ to reproduce the spectral density $J(\omega)$.
2. **Construct the doubled Hamiltonian:** Build $\bar{H} = \hat{H} \otimes \tilde{I} - I \otimes \tilde{H}$ as an MPO.
3. **Initialize the thermal wavefunction:** Generate $|\psi(0;\beta)\rangle$ using Eqs. (6.3) - (6.5).
4. **Time propagate:** Evolve via Eq. (6.8) using TDVP or TEBD.
5. **Compute observables:** Obtain $\hat{\rho}(t)$ via Eq. (6.12) and evaluate $\langle \hat{F}(t) \rangle = \mathrm{Tr}[\hat{\rho}(t)\hat{F}]$.

6.8.2 Hamiltonian

The system is described by the spin–boson Hamiltonian:

$$H = \epsilon\sigma_z + \Gamma\sigma_x + \sum_{k=1}^{N_n} \omega_k a_k^\dagger a_k + \sigma_z \sum_{k=1}^{N_n} g_k \left(a_k^\dagger + a_k\right), \tag{6.23}$$

where $a_k^\dagger$ and a_k are the creation and annihilation operators of the vibrational bath. The linear coupling coefficients are

$$g_k = -\frac{c_k}{\sqrt{2\omega_k}}, \tag{6.24}$$

which represent the coupling strength between the electronic sites and bath modes.

Discretization of bath frequencies. The bath frequencies ω_k are discretized logarithmically as

$$\omega_k = -\omega_c \log\left(1 - k\frac{\Omega}{\omega_c}\right), \tag{6.25}$$

where

$$\Omega = (1 - \mathrm{e}^{-\omega_{\max}/\omega_c})\frac{\omega_c}{N_n}, \qquad \omega_{\max} \text{ is the cutoff frequency.} \tag{6.26}$$

For an Ohmic spectral density, the coupling constants take the form

$$g_k = -\sqrt{\xi\,\omega_k\,\Omega/2}. \tag{6.27}$$

6.8.3 Thermo Field Dynamics Initial State

The TT-TFD method solves the thermal Schrödinger equation

$$\frac{\partial}{\partial t}\left|\Psi(t,\beta)\right\rangle = -i\bar{H}\left|\Psi(t,\beta)\right\rangle, \tag{6.28}$$

with an initial thermal wavepacket $|\Psi(0,\beta)\rangle$ satisfying

$$\left|\Psi(0,\beta)\right\rangle\left\langle\Psi(0,\beta)\right| = \rho_S(0)\otimes Z_B^{-1}\mathrm{e}^{-\beta H_B}. \tag{6.29}$$

Where $\rho_S(0)$ is the initial system density operator and H_B is the bath Hamiltonian.

The thermal TFD state can be generated from the doubled vacuum $\left|0,\tilde{0}\right\rangle$ by a unitary transformation (equivalent to Eq. (6.5)),[48]

$$\left|\Psi(0,\beta)\right\rangle = \mathrm{e}^{-iG}\left|0,\tilde{0}\right\rangle, \tag{6.30}$$

where, for a harmonic environment, the generator is

$$G = -i\sum_{k=1}^{N_n}\theta_k\left(a_k\tilde{a}_k - a_k^\dagger\tilde{a}_k^\dagger\right), \qquad \theta_k = \operatorname{arctanh}\left(\mathrm{e}^{-\beta\omega_k/2}\right). \tag{6.31}$$

The unitary $S_\theta = \mathrm{e}^{-iG}$ entangles each physical mode with its tilde counterpart through a two-mode squeezing transformation.

The action of G on the operators is determined by the bosonic relations

$$[a_k, a_{k'}^\dagger] = \delta_{kk'}, \qquad [\tilde{a}_k, \tilde{a}_{k'}^\dagger] = \delta_{kk'}, \qquad [a_k, \tilde{a}_{k'}] = [a_k, \tilde{a}_{k'}^\dagger] = 0, \tag{6.32}$$

together with the Baker-Campbell-Hausdorff expansion. Straightforward algebra yields the Bogoliubov transformations

$$b_k(\theta) = \mathrm{e}^{iG} a_k \mathrm{e}^{-iG} = a_k \cosh\theta_k - \tilde{a}_k^\dagger \sinh\theta_k, \tag{6.33a}$$

$$\tilde{b}_k(\theta) = \mathrm{e}^{iG} \tilde{a}_k \mathrm{e}^{-iG} = \tilde{a}_k \cosh\theta_k - a_k^\dagger \sinh\theta_k, \tag{6.33b}$$

with analogous expressions for the creation operators. Thus the thermal state may be represented either as a squeezed state in the original $(a_k, \tilde{a}_k)$ basis or, equivalently, as the simple product vacuum $\left|0, \tilde{0}\right\rangle$ when expressed in the rotated operators $b_k(\theta), \tilde{b}_k(\theta)$.

To apply this transformation to dynamics, we consider the doubled Hamiltonian used in TFD,

$$\bar{H} = \hat{H} \otimes \tilde{I} - I \otimes \tilde{H}, \tag{6.34}$$

where $\hat{H}$ is the physical spin–boson Hamiltonian

$$\hat{H} = \epsilon\sigma_z + \Gamma\sigma_x + \sum_{k=1}^{N_n} \omega_k \, a_k^\dagger a_k + \sigma_z \sum_{k=1}^{N_n} g_k (a_k^\dagger + a_k). \tag{6.35}$$

The "tilde" Hamiltonian $\tilde{H}$ has the same functional form as $\hat{H}$ but is written in terms of the fictitious operators $\tilde{a}_k, \tilde{a}_k^\dagger$ and acts on the auxiliary Hilbert space. The minus sign in Eq. (6.34) ensures that thermal expectation values can be written as pure-state overlaps in the doubled space.

Applying the Bogoliubov unitary to $\bar{H}$ defines the rotated Hamiltonian

$$\bar{H}_\theta = \mathrm{e}^{iG} \, \bar{H} \, \mathrm{e}^{-iG}, \tag{6.36}$$

which is expressed entirely in terms of the transformed operators $b_k(\theta)$ and their tilde counterparts. Using Eqs. (6.33), one finds

$$\begin{aligned}\bar{H}_\theta = \epsilon\sigma_z + \Gamma\sigma_x + \sum_{k=1}^{N_n} \omega_k \left(b_k^\dagger b_k - \tilde{b}_k^\dagger \tilde{b}_k\right) \\ + \sigma_z \sum_{k=1}^{N_n} g_k \Big[\cosh\theta_k \,(b_k^\dagger + b_k) - \sinh\theta_k \,(\tilde{b}_k^\dagger + \tilde{b}_k)\Big].\end{aligned} \tag{6.37}$$

Temperature enters entirely through the coefficients $\cosh\theta_k$ and $\sinh\theta_k$ that weight the couplings to the physical and fictitious modes, respectively. The structure of the temperature-dependent Hamiltonian $\bar{H}_\theta$ illustrates a key advantage of the rotated picture: the initial state may be kept in the simple vacuum configuration

$$|\Psi_\theta(0,\beta)\rangle = \left|0,\tilde{0}\right\rangle, \tag{6.38}$$

while all thermal effects appear in the transformed Hamiltonian.

Consequently, the dynamics follow the rotated Schrödinger equation,

$$\frac{\partial}{\partial t}\,|\Psi_\theta(t,\beta)\rangle = -i\,\bar{H}_\theta\;|\Psi_\theta(t,\beta)\rangle\,, \tag{6.39}$$

a form particularly convenient for TT-TFD simulations of the spin–boson model, where the vacuum TT/MPS structure is simple and all temperature dependence is consolidated into the MPO representation of $\bar{H}_\theta$.[49]

6.8.4 TT-TFD Simulations

Script S.4.2–Script S.4.4 show the implementation of TT-TFD simulations with QFlux for the dynamics of a qubit coupled to a bath of quantum harmonic oscillators. The resulting evolution of populations and coherences is shown in Fig. 5.

Fig. 5 shows the time evolution of the spin populations and coherences obtained from the TT–TFD simulation of a qubit coupled to a harmonic oscillator bath. The population

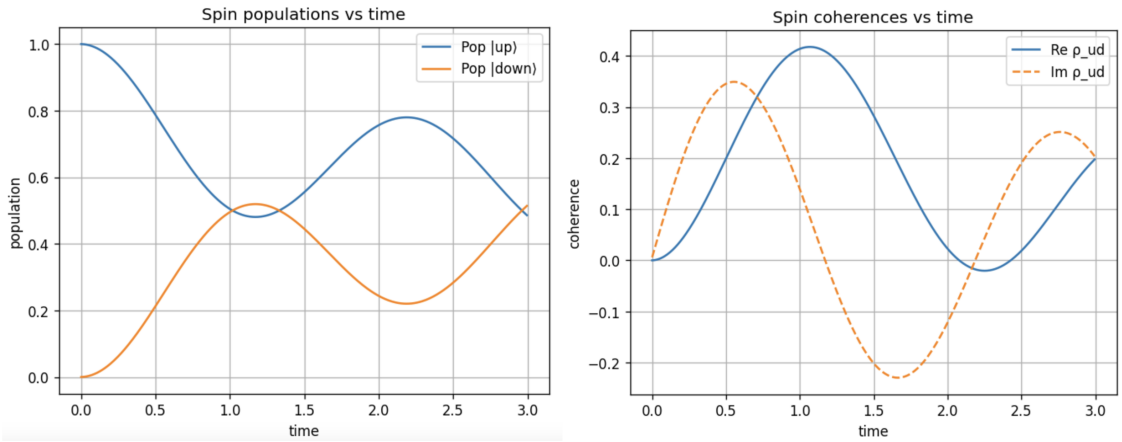

Figure 5: Comparison of time-dependent populations and coherences obtained by TT-TFD simulations of a qubit coupled to a bath of quantum harmonic oscillators.

dynamics (left panel) exhibit coherent oscillations between the $|\uparrow\rangle$ and $|\downarrow\rangle$ states, with partial population transfer and no unphysical leakage, indicating that the reduced density matrix remains properly normalized throughout the propagation. These oscillations reflect coherent system–bath energy exchange rather than simple monotonic relaxation. The corresponding coherences (right panel) display oscillatory behavior in both the real and imaginary components, with a clear phase shift between them, consistent with unitary precession modulated by environmental coupling. Importantly, the smooth temporal behavior of both populations and coherences, together with the absence of numerical instabilities or spurious discontinuities, demonstrates that the TT–TFD approach accurately captures coupled coherent and dissipative dynamics while maintaining numerical stability.

6.9 Summary and significance

The TT-TFD formalism provides a unified, wavefunction-based description of finite-temperature quantum dynamics. By working in the doubled Hilbert space, it converts a mixed-state problem into a pure-state evolution, allowing efficient tensor-network algorithms to be applied. This method is particularly powerful for short-time, numerically exact simulations that can be combined with Generalized Quantum Master Equation (GQME) methods for efficient long-time predictions, as discussed in Part VI of this QFlux tutorial.[50]

6.10 Practical considerations

- **Mode ordering & entanglement:** order physical and tilde modes to minimize entanglement growth (often interleaving k and $\tilde{k}$ helps).
- **Local truncation:** choose per-mode occupation cutoffs based on T and coupling strength; higher T generally requires larger cutoffs.
- **Error control:** monitor discarded weight / bond dimensions and validate against converged short-time reference runs.
- **Reuse via propagators:** precompute and cache the TT-TFD propagator (superoperator) when solving families of initial states or GQME kernels.

These best practices are reflected in the `QFlux` tutorials and examples.

Takeaway statement: Quantum Dynamics of Mixed States

What skill or insight have we gained? The key lesson is that finite-temperature dynamics can be simulated using pure-state propagation at the cost of Hilbert-space doubling, with tensor networks controlling the scaling.

7 Conclusions

Time-dependent quantum dynamics provide a unifying language for understanding how microscopic quantum states evolve and give rise to experimentally observable behavior. Across chemistry, physics, and emerging quantum technologies, this perspective connects wavefunction motion to quantities such as populations, spectra, and correlation functions, and it underlies both classical simulation methods and quantum algorithms for real-time dynamics.

The primary goal of **Part I** of this tutorial series is to build intuition and good computational practice. We introduced two complementary classical propagation strategies-direct

numerical integration and split-operator methods-and showed how they arrive at the same physical results when properly converged. This comparison emphasizes a central lesson for students: reliable simulations require validation across independent methods, not blind reliance on a single algorithm or software package. Throughout, simple benchmark systems were used to highlight how numerical choices affect accuracy, stability, and physical interpretation.

A key pedagogical message of this work is that classical propagation methods are more than numerical tools. Their operator structure closely mirrors that of quantum algorithms used for Hamiltonian simulation. In particular, split-operator schemes anticipate the structure of quantum circuits built from basis changes and phase operations. Classical simulations therefore serve a dual purpose: they provide physical insight and reference data, and they act as conceptual prototypes for quantum implementations. This viewpoint helps demystify quantum simulation by revealing it as a natural extension of familiar classical ideas.

Within this framework, **QFlux plays a distinct educational role**. While packages such as QuTiP and Qiskit are essential within their respective domains, QFlux is designed to connect them. It enforces a model-centered workflow in which the same Hamiltonian, initial state, and observables are carried consistently across classical solvers, tensor-network methods, and quantum-ready algorithms. For students, this structure makes differences between approaches transparent and traceable to physical approximations, numerical error, or hardware constraints, rather than to software-specific conventions.

Part I serves as the foundation for the remainder of the tutorial series. **Part II** translates the classical ideas developed here into executable quantum circuits for closed-system dynamics. **Part III** focuses on state preparation and unitary decomposition, showing how abstract states and operators are mapped to hardware-efficient circuits. **Part IV** extends the framework to open quantum systems using Lindblad dynamics and dilation methods. **Part V** introduces adaptive variational algorithms tailored to near-term quantum hardware. **Part VI** addresses non-Markovian dynamics and memory effects through generalized quantum

master equations.

Together, these installments form a coherent learning pathway from classical intuition to quantum implementation. By emphasizing validation, operator structure, and consistent workflows, this series aims to equip students with the conceptual tools needed to navigate quantum dynamics across classical computation, hybrid algorithms, and emerging quantum hardware.

Supporting Information

Detailed Python scripts implementing ODE solver and SOFT propagation methods, convergence tests, and additional figures are provided in the Supporting Information and corresponding Google Colab notebook as well as through the QFlux Documentation site.

Acknowledgments

This work was supported by the National Science Foundation under Award No. 2124511 (CCI Phase I: NSF Center for Quantum Dynamics on Modular Quantum Devices, CQD-MQD) and Award No. 2302908 (Engines Development Award: Advancing Quantum Technologies, CT). The authors also acknowledge the use of IBM Quantum services and open-source software packages, including Qiskit, Bosonic Qiskit, Strawberry Fields, QuTiP, and MPSQD.

References

(1) Breuer, H.; Petruccione, F. *The Theory of Open Quantum Systems*; Oxford University Press, 2002.

(2) Lindblad, G. On the Generators of Quantum Dynamical Semigroups. *Communications in Mathematical Physics* **1976**, *48*, 119–130.

(3) Gorini, V.; Kossakowski, A.; Sudarshan, E. C. G. Completely Positive Dynamical Semigroups of N-Level Systems. *Journal of Mathematical Physics* **1976**, *17*, 821–825.

(4) Nakajima, S. On Quantum Theory of Transport Phenomena: Steady Diffusion. *Progress of Theoretical Physics* **1958**, *20*, 948–959.

(5) Zwanzig, R. Ensemble Method in the Theory of Irreversibility. *The Journal of Chemical Physics* **1960**, *33*, 1338–1341.

(6) Vacchini, B. Generalized Master Equations Leading to Completely Positive Dynamics. *Journal of Mathematical Physics* **2016**, *57*, 072101.

(7) Dan, X.; Xu, M.; Yan, Y.; Shi, Q. Generalized master equation for charge transport in a molecular junction: Exact memory kernels and their high order expansion. *Journal of Chemical Physics* **2022**, *156*, 134114.

(8) Nielsen, M. A.; Chuang, I. L. *Quantum Computation and Quantum Information*; Cambridge University Press, 2010.

(9) Dutta, R.; Cabral, D. G. A.; Lyu, N.; Vu, N. P.; Wang, Y.; Allen, B.; Dan, X.; Cortiñas, R. G.; Khazaei, P.; Schäfer, M.; Albornoz, A. C. C. d.; Smart, S. E.; Nie, S.; Devoret, M. H.; Mazziotti, D. A.; Narang, P.; Wang, C.; Whitfield, J. D.; Wilson, A. K.; Hendrickson, H. P.; Lidar, D. A.; Pérez-Bernal, F.; Santos, L. F.; Kais, S.; Geva, E.; Batista, V. S. Simulating Chemistry on Bosonic Quantum Devices. *Journal of Chemical Theory and Computation* **2024**, *20*, 6426–6441.

(10) Preskill, J. Quantum Computing in the NISQ Era and Beyond. *Quantum* **2018**, *2*, 79.

(11) Dan, X.; Geva, E.; Batista, V. S. Simulating Non-Markovian Quantum Dynamics on NISQ Computers Using the Hierarchical Equations of Motion. *Journal of Chemical Theory and Computation* **2025**, *21*, 1530–1546.

(12) Vu, N. P.; Dong, D.; Dan, X.; Lyu, N.; Batista, V.; Liu, Y. A Computational Framework for Simulations of Dissipative Nonadiabatic Dynamics on Hybrid Oscillator-Qubit Quantum Devices. *Journal of Chemical Theory and Computation* **2025**, *21*, 6258–6279.

(13) Lambert, N.; Giguère, E.; Menczel, P.; Li, B.; Hopf, P.; Suárez, G.; Gali, M.; Lishman, J.; Gadhvi, R.; Agarwal, R.; Galicia, A.; Shammah, N.; Nation, P.; Johansson, J.; Ahmed, S.; Cross, S.; Pitchford, A.; Nori, F. QuTiP 5: The Quantum Toolbox in Python. *Physics Reports* **2026**, *1153*, 1–62.

(14) Guan, W.; Bao, P.; Peng, J.; Lan, Z.; Shi, Q. mpsqd: A matrix product state based Python package to simulate closed and open system quantum dynamics. *The Journal of Chemical Physics* **2024**, *161*, 122501.

(15) Javadi-Abhari, A.; Treinish, M.; Krsulich, K.; Wood, C. J.; Lishman, J.; Gacon, J.; Martiel, S.; Nation, P. D.; Bishop, L. S.; Cross, A. W.; Johnson, B. R.; Gambetta, J. M. Quantum computing with Qiskit. 2024; https://arxiv.org/abs/2405.08810.

(16) Allen, B. C.; Batista, V. S.; Cabral, D. G. A.; Cianci, C.; Dan, X.; Dutta, R.; Geva, E.; Hu, Z.; Kais, S.; Khazaei, P.; Lyu, N.; Mulvihill, E.; Shivpuje, S.; Soudackov, A. V.; Vu, N. P.; Wang, Y.; Wilson, C. QFlux — An Open-Source Python Package for Quantum Dynamics Simulations. https://qflux.batistalab.com, 2025; (accessed: 2025-10-12).

(17) Harris, C. R.; Millman, K. J.; van der Walt, S. J.; Gommers, R.; Virtanen, P.; Cournapeau, D.; Wieser, E.; Taylor, J.; Berg, S.; Smith, N. J.; Kern, R.; Picus, M.; Hoyer, S.; van Kerkwijk, M. H.; Brett, M.; Haldane, A.; del Río, J. F.; Wiebe, M.; Peterson, P.; Gérard-Marchant, P.; Sheppard, K.; Reddy, T.; Weckesser, W.; Abbasi, H.; Gohlke, C.; Oliphant, T. E. Array programming with NumPy. *Nature* **2020**, *585*, 357–362.

(18) Virtanen, P.; Gommers, R.; Oliphant, T. E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; van der Walt, S. J.;

Brett, M.; Wilson, J.; Millman, K. J.; Mayorov, N.; Nelson, A. R. J.; Jones, E.; Kern, R.; Larson, E.; Carey, C. J.; Polat, İ.; Feng, Y.; Moore, E. W.; VanderPlas, J.; Laxalde, D.; Perktold, J.; Cimrman, R.; Henriksen, I.; Quintero, E. A.; Harris, C. R.; Archibald, A. M.; Ribeiro, A. H.; Pedregosa, F.; van Mulbregt, P.; SciPy 1.0 Contributors SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* **2020**, *17*, 261–272.

(19) Feit, M.; Fleck, J.; Steiger, A. Solution of the Schrödinger equation by a spectral method. *Journal of Computational Physics* **1982**, *47*, 412–433.

(20) Trotter, H. F. On the Product of Semi-Groups of Operators. *Proceedings of the American Mathematical Society* **1959**, *10*, 545–551.

(21) Suzuki, M. Fractal Decomposition of Exponential Operators with Applications to Many-Body Theories and Monte Carlo Simulations. *Physics Letters A* **1990**, *146*, 319–323.

(22) Campbell, E. T. Randomized Compiling for Scalable Quantum Computing. *Physical Review Letters* **2019**, *123*, 070503.

(23) McLachlan, A. D. A Variational Solution of the Time-Dependent Schrödinger Equation. *Molecular Physics* **1964**, *8*, 39–44.

(24) Endo, S.; Sun, J.; Li, Y.; Benjamin, S. C.; Yuan, X. Variational Quantum Simulation of General Processes. *Phys. Rev. Lett.* **2020**, *125*, 010501.

(25) Kraus, K. General State Changes in Quantum Theory. *Annals of Physics* **1971**, *64*, 311–335.

(26) Stinespring, W. F. Positive Functions on C^{ast}-Algebras. *Proceedings of the American Mathematical Society* **1955**, *6*, 211–216.

(27) Shi, Q.; Xu, Y.; Yan, Y.; Xu, M. Efficient propagation of the hierarchical equations

of motion using the matrix product state method. *Journal of Chemical Physics* **2018**, *148*, 174102.

(28) Runge, C. Über die numerische Auflöung von Differentialgleichungen. *Mathematische Annalen* **1895**, *46*, 167–178.

(29) Kutta, W. Beitrag zur näerungsweisen Integration totaler Differentialgleichungen. *Zeitschrift für Mathematik und Physik* **1901**, *46*, 435–453.

(30) Ascher, U. M.; Petzold, L. R. *Computer methods for ordinary differential equations and differential-algebraic equations*; SIAM: Society for Industrial and Applied Mathematics, 1998.

(31) Dormand, J. R.; Prince, P. J. A Family of Embedded Runge–Kutta Formulae. *Journal of Computational and Applied Mathematics* **1980**, *6*, 19–26.

(32) Butcher, J. C. *Numerical Methods for Ordinary Differential Equations*, 3rd ed.; John Wiley & Sons: Chichester, UK, 2016.

(33) Brown, P. N.; Byrne, G. D.; Hindmarsh, A. C. VODE: A Variable-Coefficient ODE Solver. *SIAM Journal on Scientific and Statistical Computing* **1989**, *10*, 1038–1051.

(34) Cohen, S. D.; Hindmarsh, A. C. CVODE, a Stiff/Nonstiff ODE Solver in C. *Computers in Physics* **1996**, *10*, 138–143.

(35) Gear, C. W. *Numerical Initial Value Problems in Ordinary Differential Equations*; Prentice Hall: Englewood Cliffs, NJ, 1971.

(36) Shampine, L. F.; Gordon, M. K. *Computer Solution of Ordinary Differential Equations: The Initial Value Problem*; W. H. Freeman: San Francisco, 1975.

(37) Hindmarsh, A. C. In *Scientific Computing*; Stepleman, R. S., others, Eds.; IMACS Transactions on Scientific Computation, Vol. 1; North-Holland: Amsterdam, 1983; pp 55–64.

(38) Johansson, J. R.; Nation, P. D.; Nori, F. QuTiP: An open-source Python framework for the dynamics of open quantum systems. *Computer Physics Communications* **2012**, *183*, 1760–1772.

(39) Johansson, J. R.; Nation, P. D.; Nori, F. QuTiP 2: A Python framework for the dynamics of open quantum systems. *Computer Physics Communications* **2013**, *184*, 1234–1240.

(40) QFlux includes QuTiP as well as other libraries like Qiskit as dependencies, enabling the QFlux workflow to be executed on both classical simulators and quantum backends.

(41) Feit, M. D.; Jr., J. A. F.; Steiger, A. Solution of the Schrödinger Equation by a Spectral Method. *Journal of Computational Physics* **1982**, *47*, 412–433.

(42) Feit, M. D.; Fleck Jr., J. A. Solution of the Schrödinger equation by a spectral method II: Vibrational energy levels of triatomic molecules. *Journal of Chemical Physics* **1983**, *78*, 301–308.

(43) Greene, S. M.; Batista, V. S. Tensor-train split-operator Fourier transform (TT-SOFT) method: Multidimensional nonadiabatic quantum dynamics. *Journal of Chemical Theory and Computation* **2017**, *13*, 4034–4042.

(44) Lyu, N.; Soley, M. B.; Batista, V. S. Tensor-train split-operator KSL (TT-SOKSL) method for quantum dynamics simulations. *Journal of Chemical Theory and Computation* **2022**, *18*, 3327–3346.

(45) Lyu, N.; Mulvihill, E.; Soley, M. B.; Geva, E.; Batista, V. S. Tensor-Train Thermo-Field Memory Kernels for Generalized Quantum Master Equations. *Journal of Chemical Theory and Computation* **2023**, *19*, 1111–1129.

(46) Borrelli, R.; Gelin, M. F. Finite temperature quantum dynamics of complex systems: In-

tegrating thermo-field theories and tensor-train methods. *WIREs Computational Molecular Science* **2021**, *11*, e1539.

(47) Lubich, C.; Oseledets, I.; Vandereycken, B. Time Integration of Tensor Trains. *SIAM Journal on Numerical Analysis* **2015**, *53*, 917–941.

(48) Here, we show that Eq. (6.5) is equivalent to Eq. (6.30) for a single mode. Eq. (6.5) for a single harmonic mode of frequency ω is

$$|\psi(0;\beta)\rangle = Z_\beta^{-1/2} \sum_{n=0}^{\infty} e^{-\beta\omega n/2} \frac{1}{n!} (\hat{a}^\dagger \tilde{a}^\dagger)^n |0,\tilde{0}\rangle \tag{7.1}$$

$$= \sqrt{1-e^{-\beta\omega}}\, \exp\Big[e^{-\beta\omega/2} \hat{a}^\dagger \tilde{a}^\dagger\Big] |0,\tilde{0}\rangle, \tag{7.2}$$

where $Z_\beta = (1-e^{-\beta\omega})^{-1}$. Expanding the exponential reproduces the sum, and acting $(\hat{a}^\dagger\tilde{a}^\dagger)^n$ on the double vacuum yields $|n,\tilde{n}\rangle$, giving the standard thermofield double series with Boltzmann weights. Eq. (6.30) for the same state is written as a unitary generated by the two-mode squeezing operator:

$$|\Psi(0,\beta)\rangle = e^{-iG}|0,\tilde{0}\rangle, \qquad G = -i\theta(a\tilde{a} - a^\dagger\tilde{a}^\dagger), \qquad \theta = \operatorname{arctanh}(e^{-\beta\omega/2}). \tag{7.3}$$

Using the SU(1,1) disentangling identity, the squeezing operator can be written as

$$e^{-iG} = \exp\Big[\gamma\, a^\dagger\tilde{a}^\dagger\Big]\, \exp\Big[\eta\,(a^\dagger a + \tilde{a}^\dagger\tilde{a} + 1)\Big]\, \exp\Big[-\gamma\, a\tilde{a}\Big], \tag{7.4}$$

with $\gamma = \tanh\theta$ and $\eta = -\ln(\cosh\theta)$. Acting on the vacuum, the last factor is trivial, and the middle factor yields a scalar:

$$\exp\Big[\eta(a^\dagger a + \tilde{a}^\dagger\tilde{a} + 1)\Big]|0,\tilde{0}\rangle = \frac{1}{\cosh\theta}\,|0,\tilde{0}\rangle. \tag{7.5}$$

Thus,

$$e^{-iG}|0,\tilde{0}\rangle = \frac{1}{\cosh\theta}\exp\left[\gamma\, a^\dagger \tilde{a}^\dagger\right]|0,\tilde{0}\rangle = \frac{1}{\cosh\theta}\sum_{n=0}^{\infty}(\tanh\theta)^n|n,\tilde{n}\rangle. \tag{7.6}$$

Matching this with the coefficients of the first expression requires

$$\tanh\theta = e^{-\beta\omega/2}, \tag{7.7}$$

which also gives $\cosh^{-1}\theta = \sqrt{1-\tanh^2\theta} = \sqrt{1-e^{-\beta\omega}}$. With this identification, both forms yield identical expansions, establishing their equivalence. For multiple bath modes, each mode k has its own θ_k with $\tanh\theta_k = e^{-\beta\omega_k/2}$, and the full bath state is simply the product over modes since the generators commute for different k. .

(49) **Anharmonic baths: no Bogoliubov simplification.** The Bogoliubov transformation relies crucially on the *quadratic* nature of the harmonic oscillator Hamiltonian. For an anharmonic potential—for example a Morse oscillator, quartic oscillator, or double-well potential—*no unitary of the Bogoliubov/squeezing type exists*. Consequently, for general anharmonic baths,

- one *cannot* work in the rotated picture with a compact Hamiltonian $\bar{H}_\theta$,
- and one *cannot* encode finite temperature by a simple squeezing of the vacuum.

Thermal-state preparation for anharmonic oscillators. Instead, each oscillator's thermal density matrix

$$\rho_k(\beta) = \frac{e^{-\beta H_k}}{Z_k}$$

must be represented directly in the TFD formalism. A standard route is: 1. Start from the high-temperature mixed state $\rho_k(\beta_0)$ with small β_0. 2. Perform *imaginary-time evolution*

$$\rho_k(\beta) \propto e^{-(\beta-\beta_0)H_k}\,\rho_k(\beta_0),$$

using purification or TT/MPS representations. 3. Build the full thermal TFD state as

a tensor product

$$|\Psi(0,\beta)\rangle = \bigotimes_{k=1}^{N_n} |\Psi_k(\beta)\rangle\,,$$

where each $|\Psi_k(\beta)\rangle$ purifies $\rho_k(\beta)$. Once the initial thermal state is prepared explicitly in this way, the dynamics are governed by the *unrotated* doubled Hamiltonian $\bar{H}$,

$$\frac{\partial}{\partial t}\,|\Psi(t,\beta)\rangle = -i\,\bar{H}\;|\Psi(t,\beta)\rangle\,, \tag{7.8}$$

because no simple analytic G exists that could absorb temperature into a rotated Hamiltonian for anharmonic modes. In summary:

- For harmonic baths, thermal effects can be shifted entirely into the rotated Hamiltonian $\bar{H}_\theta$ using a Bogoliubov transformation.
- For anharmonic baths, no such squeezing transformation exists, so the thermal TFD state must be explicitly constructed (e.g. via imaginary-time evolution), and time evolution must be performed with the unrotated Hamiltonian $\bar{H}$.

.

(50) Short-time, numerically exact TT-TFD trajectories provide the correlation functions required to construct Generalized Quantum Master Equation (GQME) memory kernels and inhomogeneous terms, enabling long-time dynamics at reduced cost. This hybrid (TT-TFD $\rightarrow$ GQME) strategy yields accurate kinetics and mechanistic observables across coupling/temperature regimes, and has been validated on spin–boson benchmarks.

Supporting Information for

QFlux: Classical Foundations for Quantum Dynamics Simulation.

Part I - Building Intuition and Computational Workflows

Brandon C. Allen[†], Xiaohan Dan[†], Delmar G. A. Cabral[†], Nam P. Vu[†], Cameron Cianci[‡], Alexander V. Soudackov[†], Rishab Dutta[†], Sabre Kais[¶], Eitan Geva[§] and Victor S. Batista[*,‖,⊥]

[†]Department of Chemistry, Yale Quantum Institute, Yale University, New Haven, CT 06511, USA

[‡]Department of Physics, University of Connecticut, Storrs, CT 06268, USA

[¶]Department of Electrical and Computer Engineering, Department of Chemistry, North Carolina State University, Raleigh, North Carolina 27606, USA

[§]Department of Chemistry, University of Michigan, Ann Arbor, MI 48109, USA

[‖]Department of Chemistry, Yale University, New Haven, CT 06520, USA

[⊥]Yale Quantum Institute, Yale University, New Haven, CT 06511, USA

E-mail: victor.batista@yale.edu

Contents

S.1 Runge–Kutta–Fehlberg 4(5) Propagation

Consider the time-dependent Schrödinger equation

$$\mathrm{i}\hbar\frac{\partial\,|\psi(t)\rangle}{\partial t} = \hat{H}(t)\,|\psi(t)\rangle\,, \qquad |\psi(t)\rangle = \sum_{j=1}^{N} c_j(t)\,|\phi_j(t)\rangle\,. \tag{S.1}$$

S.1.1 Case A: Orthonormal, Time-Independent Basis

If $\{|\phi_j\rangle\}_{j=1}^{N}$ is orthonormal and time independent, define $H_{ij}(t) = \langle\phi_i|\,\hat{H}(t)\,|\phi_j\rangle$ and collect the coefficients into $c(t) = (c_1, \ldots, c_N)^{\top}$. The Schrödinger equation becomes the ordinary differential equation

$$\dot{c}(t) = f(t, c) \equiv -\frac{\mathrm{i}}{\hbar}\,H(t)\,c(t). \tag{S.2}$$

Given (t_n, c_n) and a step size h, the classical Runge–Kutta–Fehlberg 4(5) stages are

$$k_1 = f(t_n,\, c_n), \tag{S.3}$$

$$k_2 = f\Big(t_n + \frac{1}{4}h,\, c_n + \frac{1}{4}h\,k_1\Big)\,, \tag{S.4}$$

$$k_3 = f\Big(t_n + \frac{3}{8}h,\, c_n + h\Big(\frac{3}{32}k_1 + \frac{9}{32}k_2\Big)\Big)\,, \tag{S.5}$$

$$k_4 = f\Big(t_n + \frac{12}{13}h,\, c_n + h\Big(\frac{1932}{2197}k_1 - \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3\Big)\Big)\,, \tag{S.6}$$

$$k_5 = f\Big(t_n + h,\, c_n + h\Big(\frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4\Big)\Big)\,, \tag{S.7}$$

$$k_6 = f\Big(t_n + \frac{1}{2}h,\, c_n + h\Big(-\frac{8}{27}k_1 + 2k_2 - \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5\Big)\Big)\,. \tag{S.8}$$

The embedded 4th- and 5th-order updates are

$$c_{n+1}^{(4)} = c_n + h\Big(\frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5\Big)\,, \tag{S.9}$$

$$c_{n+1}^{(5)} = c_n + h\Big(\frac{16}{135}k_1 + \frac{6656}{12825}k_3 + \frac{28561}{56430}k_4 - \frac{9}{50}k_5 + \frac{2}{55}k_6\Big)\,. \tag{S.10}$$

The embedded pair provides a local error estimate

$$\Delta = \left\| c_{n+1}^{(5)} - c_{n+1}^{(4)} \right\|, \tag{S.11}$$

and a standard adaptive-step update,

$$h_{\text{new}} = h \cdot \min\Big(4,\, \max\Big(0.1,\, 0.84\big(\tfrac{\text{tol}}{\Delta}\big)^{1/4}\Big)\Big), \tag{S.12}$$

where `tol` is the user-defined tolerance and the safety factors 0.1 and 4 bound the step changes.

Remark on unitarity. RK45 is not exactly norm-preserving. For Hermitian H in an orthonormal basis, it can be useful to monitor and optionally renormalize the coefficient vector, $c_{n+1} \leftarrow c_{n+1}/\|c_{n+1}\|$, or tighten the tolerance to control norm drift.

S.1.2 Case B: Non-Orthonormal Time-Independent Basis

For a time-independent but non-orthonormal basis with overlap matrix $S_{ij} = \langle \phi_i | \phi_j \rangle$, the coefficients satisfy

$$\mathrm{i}\hbar\, S\, \dot{c}(t) = H(t)\, c(t) \quad \Rightarrow \quad \dot{c}(t) = g(t, c) \equiv -\frac{\mathrm{i}}{\hbar}\, S^{-1} H(t)\, c(t). \tag{S.13}$$

To avoid forming S^{-1} explicitly, each Runge–Kutta stage is implemented via linear solves:

$$S\,k_1 = -\frac{\mathrm{i}}{\hbar} H(t_n)\,c_n, \tag{S.14}$$

$$S\,k_2 = -\frac{\mathrm{i}}{\hbar} H\Big(t_n + \tfrac{1}{4}h\Big)\Big(c_n + \tfrac{1}{4}h\,k_1\Big), \tag{S.15}$$

$$S\,k_3 = -\frac{\mathrm{i}}{\hbar} H\Big(t_n + \tfrac{3}{8}h\Big)\Big(c_n + h(\tfrac{3}{32}k_1 + \tfrac{9}{32}k_2)\Big), \tag{S.16}$$

$$S\,k_4 = -\frac{\mathrm{i}}{\hbar} H\Big(t_n + \tfrac{12}{13}h\Big)\Big(c_n + h(\tfrac{1932}{2197}k_1 - \tfrac{7200}{2197}k_2 + \tfrac{7296}{2197}k_3)\Big), \tag{S.17}$$

$$S\,k_5 = -\frac{\mathrm{i}}{\hbar} H(t_n + h)\Big(c_n + h(\tfrac{439}{216}k_1 - 8k_2 + \tfrac{3680}{513}k_3 - \tfrac{845}{4104}k_4)\Big), \tag{S.18}$$

$$S\,k_6 = -\frac{\mathrm{i}}{\hbar} H\Big(t_n + \tfrac{1}{2}h\Big)\Big(c_n + h(-\tfrac{8}{27}k_1 + 2k_2 - \tfrac{3544}{2565}k_3 + \tfrac{1859}{4104}k_4 - \tfrac{11}{40}k_5)\Big). \tag{S.19}$$

The 4th- and 5th-order updates are then constructed from the same linear combinations of $\{k_\ell\}$ as in the orthonormal case.

Pre-factorizing S (e.g., via a Cholesky decomposition $S = LL^\dagger$) allows each linear system solve to be implemented efficiently.

S.1.3 Optional: Time-Dependent Basis

If the basis functions themselves are time dependent, $|\phi_j\rangle = |\phi_j(t)\rangle$, nonadiabatic couplings

$$\tau_{ij}(t) = \Big\langle \phi_i(t) \Big| \dot{\phi}_j(t) \Big\rangle \tag{S.20}$$

enter the equations of motion as

$$\mathrm{i}\hbar\, S(t)\,\dot{c}(t) = \Big(H(t) - \mathrm{i}\hbar\,\tau(t)\Big)c(t), \quad S_{ij}(t) = \langle \phi_i(t) | \phi_j(t) \rangle\,. \tag{S.21}$$

The RK45 structure is unchanged, but at each stage both H and S are evaluated at the intermediate time t_* and the stage vectors k_* are obtained from linear solves $S(t_*)\,k_* = \ldots$.

S.1.4 Practical Notes

- For Hermitian H in an orthonormal basis, monitor norm drift $\|c(t)\|^2$ and renormalize if necessary.
- With a non-orthonormal basis, monitor the generalized norm $c^\dagger Sc$.
- If H is sparse or has an MPO / TT structure, evaluate products Hc using the corresponding matrix–vector routine to keep each stage efficient.

S.2 Harmonic Oscillator Benchmarks

This section presents scripts that model the dynamics of a quantum harmonic oscillator, implemented using both a ODE solver in QuTiP and the SOFT method.

S.2.1 Dynamics with ODE solver

The first script prepares and propagates a coherent state of the harmonic oscillator using QuTiP's Schrödinger-equation solver `sesolve`. It defines the Hamiltonian, time grid, and numerical solver options, and returns the full time series of state vectors (`result.states`) used in later analysis.

Script S.2.1: Harmonic Oscillator Dynamics with ODE solver

```python
import qutip as qt
import numpy as np

# Define the system parameters
mass = 1.0
hbar = 1.0
omega = 1.0

# Initial state: coherent state with amplitude alpha = (x0 + i p_0)/sqrt(2)
x_0, p_0 = 1.0, 0.0
N = 128  # Number of basis states
psi_0 = qt.coherent(N, alpha=(x_0 + 1.j*p_0)/np.sqrt(2))

# Time grid
n_steps, total_time = 400, 20.0
tlist = np.linspace(0, total_time, n_steps)

# Define the Hamiltonian
a = qt.destroy(N)
H_ho = hbar * omega * (a.dag() * a + 0.5)

# Propagate using the Runge-Kutta solver
solver_options = {'nsteps': len(tlist), 'progress_bar': True}
result = qt.sesolve(H_ho, psi_0, tlist, options=solver_options)
```

S.2.2 Expectation Values

The second script evaluates the expectation values of position and momentum using the propagated QuTiP states from the previous listing and compares them to the analytical expressions for a coherent-state trajectory. It produces the benchmark plots shown in the main text.

Script S.2.2: Expectation Values for the Harmonic Oscillator

```
import matplotlib.pyplot as plt

# Operators for position and momentum
X_op = (a.dag() + a) / np.sqrt(2)
P_op = 1j * (a.dag() - a) / np.sqrt(2)

# Compute numerical expectation values
exp_x_qt = qt.expect(X_op, result.states)
exp_p_qt = qt.expect(P_op, result.states)

# Analytical results
exp_x_ana = [x_0*np.cos(omega*t) + (p_0/mass/omega)*np.sin(omega*t) for t in
    tlist]
exp_p_ana = [-mass*omega*x_0*np.sin(omega*t) + p_0*np.cos(omega*t) for t in
    tlist]

# Plot
fig, ax = plt.subplots()
ax.plot(tlist, exp_x_ana, '-', color='blue',
      label=r'$\langle x \rangle$ (Analytical)')
ax.plot(tlist, exp_x_qt, 'o', color='blue',
      label=r'$\langle x \rangle$ (QuTiP)',
      markeredgecolor='blue', markevery=4,
      fillstyle='full', markerfacecolor='white')
ax.plot(tlist, exp_p_ana, '-', color='red',
      label=r'$\langle p \rangle$ (Analytical)')
ax.plot(tlist, exp_p_qt, 'o', color='red',
      label=r'$\langle p \rangle$ (QuTiP)',
      markeredgecolor='red', markevery=4,
      fillstyle='full', markerfacecolor='white')
ax.axhline(0, ls='--', lw=0.5, color='black', alpha=0.5)
ax.set_xlabel('Time (a.u.)')
ax.set_ylabel('Expectation Value')
plt.legend(loc='upper center', ncol=2)
ax.set_ylim(-1.5, 1.825)
plt.hlines([-1, 0, 1], min(tlist), max(tlist),
        ls='--', lw=0.85, color='tab:grey', zorder=2)
ax.set_xlim(min(tlist), max(tlist))
plt.show()
```

S.3 SOFT Dynamics Simulations

S.3.1 State Initialization

The SOFT initialization script constructs the real-space and momentum grids and prepares a coherent-state wavepacket matching the QuTiP initial state. These arrays (`xgrid`, `pgrid`, and `psi_0`) are reused by all subsequent SOFT routines.

Script S.3.1: Coherent State Wavepacket Initialization

```python
import numpy as np

def get_xgrid(xmin, xmax, N_pts):
    """Generate an evenly spaced position grid."""
    dx = (xmax - xmin)/N_pts
    xgrid = np.arange(-N_pts/2, N_pts/2)*dx
    return xgrid

def get_pgrid(xmin, xmax, N_pts, reorder=True):
    """Generate a momentum grid using FFT-compatible ordering."""
    dp = 2 * np.pi / (xmax-xmin)
    pmin = -dp * N_pts / 2
    pmax = dp * N_pts / 2
    plus_pgrid = np.linspace(0, pmax, N_pts//2+1)
    minus_pgrid = - np.flip(np.copy(plus_pgrid))
    if reorder:
        pgrid = np.concatenate((plus_pgrid[:-1], minus_pgrid[:-1]))
    else:
        pgrid = np.concatenate((minus_pgrid, plus_pgrid))
    return pgrid

def get_coherent_state(x, p_0, x_0, mass=1, omega=1, hbar=1):
    """Generate an initial coherent state wavefunction."""
    normalization = (mass*omega/np.pi/hbar)**(0.25)
    y = normalization*np.exp(
        -1*(mass*omega/hbar/2)*((x-x_0)**2) + 1j*p_0*x/hbar
    )
    return y

xmin = -7.0
xmax = 7.0
N_pts = 128
mass = 1.0  # mass in atomic units
omega = 1.0  # oscillator frequency
xgrid = get_xgrid(xmin, xmax, N_pts)
dx = xgrid[1] - xgrid[0]
pgrid = get_pgrid(xmin, xmax, N_pts, reorder=True)

x_0 = 1.0
p_0 = 0.0
psi_0 = get_coherent_state(xgrid, p_0, x_0, mass, omega)
```

S.3.2 Propagators for the Harmonic Oscillator

This helper script defines the harmonic potential and kinetic-energy functions on the grids initialized above. The resulting arrays `Vx_harm` and `K_harm` serve as the input operators for the SOFT propagator.

Script S.3.2: SOFT Operators for Harmonic Oscillator

```python
import numpy as np

def get_harmonic_potential(x, x_0=0.0, mass=1, omega=1):
    return mass * omega**2 * (x - x_0)**2 / 2

def get_kinetic_energy(p, mass=1):
    return p**2 / (2 * mass)

Vx_harm = get_harmonic_potential(xgrid)
K_harm = get_kinetic_energy(pgrid, mass)
```

S.3.3 SOFT Propagation

The main SOFT propagation routine constructs the position- and momentum-space propagators and iteratively advances the wavefunction in time using FFTs. It returns a list of wavefunctions `propagated_states_harm` sampled on the same grid as the QuTiP trajectory.

Script S.3.3: SOFT Propagation

```python
import numpy as np

def get_propagator_on_grid(operator_grid, tau, hbar=1):
    return np.exp(-1.0j * operator_grid * tau / hbar)

def do_SOFT_propagation(psi, K_prop, V_prop):
    psi_t_position_grid = V_prop * psi
    psi_t_momentum_grid = K_prop * np.fft.fft(psi_t_position_grid, norm="ortho")
    psi_t = V_prop * np.fft.ifft(psi_t_momentum_grid, norm="ortho")
```

```
    return psi_t

tmin, tmax, N_tsteps = 0.0, 20.0, 400
tgrid = np.linspace(tmin, tmax, N_tsteps)
tau = tgrid[1] - tgrid[0]

V_prop = get_propagator_on_grid(Vx_harm/2, tau)
K_prop = get_propagator_on_grid(K_harm, tau)

propagated_states_harm = [psi_0]
psi_t = psi_0
for _ in range(len(tgrid)):
    psi_t = do_SOFT_propagation(psi_t, K_prop, V_prop)
    propagated_states_harm.append(psi_t)

propagated_states_harm = np.asarray(propagated_states_harm)[:-1]
```

S.3.4 Expectation Values

The final SOFT script computes position and momentum expectation values from the propagated wavefunctions and compares them with the analytical expressions as well as with the QuTiP benchmarks, producing the plots used in the manuscript.

Script S.3.4: Harmonic Oscillator Expectation Values

```
def position_expectation_value(xgrid, psi):
    dx = xgrid[1]-xgrid[0]
    return dx*np.real(np.sum(xgrid * np.conjugate(psi) * psi))

def momentum_expectation_value(dx, pgrid, psi):
    psip = np.fft.fft(psi)
    return dx*np.real(np.sum(pgrid * np.conjugate(psip) * psip))/len(psi)

avx_soft = [position_expectation_value(xgrid, propagated_states_harm[i])
          for i in range(len(propagated_states_harm))]
dx = xgrid[1]-xgrid[0]
avp_soft = [momentum_expectation_value(dx, pgrid, propagated_states_harm[i])
          for i in range(len(propagated_states_harm))]

avx_ana = [x_0*np.cos(omega*t) + (p_0/mass/omega)*np.sin(omega*t)
         for t in tgrid]
```

```
avp_ana = [-x_0*omega*mass*np.sin(omega*t) + p_0*np.cos(omega*t)
        for t in tgrid]

# Plot
fig, ax = plt.subplots()
ax.plot(tlist, avx_ana, '-', color='blue',
      label=r'$\langle x \rangle$ (Analytical)')
ax.plot(tlist, avx_soft, 'o', color='blue',
      label=r'$\langle x \rangle$ (SOFT)',
      markeredgecolor='blue', markevery=4,
      fillstyle='full', markerfacecolor='white')
ax.plot(tlist, avp_ana, '-', color='red',
      label=r'$\langle p \rangle$ (Analytical)')
ax.plot(tlist, avp_soft, 'o', color='red',
      label=r'$\langle p \rangle$ (SOFT)',
      markeredgecolor='red', markevery=4,
      fillstyle='full', markerfacecolor='white')
ax.axhline(0, ls='--', lw=0.5, color='black', alpha=0.5)
ax.set_xlabel('Time (a.u.)')
ax.set_ylabel('Expectation Value')
plt.legend(loc='upper center', ncol=2)
ax.set_ylim(-1.5, 1.825)
plt.hlines([-1, 0, 1], min(tlist), max(tlist),
        ls='--', lw=0.85, color='tab:grey', zorder=2)
ax.set_xlim(min(tlist), max(tlist))
plt.show()
```

S.4 TT-TFD Simulations of the Spin–Boson Model

S.4.1 QFlux Installation

This short cell installs the `qflux` package with the GQME/TT-TFD extras and imports the TT-TFD and TDVP modules used throughout the remainder of this section. It should be executed once at the beginning of a notebook.

Script S.4.1: QFlux Installation

```
!pip install qflux[gqme]
from qflux.GQME.tt_tfd import *
from qflux.GQME.tdvp import _tdvp1
from qflux.GQME.tt_utils import *
from __future__ import annotations

import matplotlib.pyplot as plt
```

S.4.2 Spin–Boson Model Parameters

The parameter container `Params` defines all physical and numerical settings for the TT-TFD calculations (model parameters, time step, TT ranks, etc.). A single global instance `pp` is created and accessed by the subsequent construction and propagation routines.

Script S.4.2: Model Parameters

```
class Params:
    def __init__(self):
        # ==== Spin-Boson Model parameters ====
        self.GAMMA_DA = 1         # diabatic coupling
        self.EPSILON  = 1
        self.BETA     = 5         # inverse temperature beta = 1 / (k_B * T)
        self.XI       = 0.1
        self.OMEGA_C  = 2

        # Spin-up and spin-down states
        self.spin_up   = np.array([1.0, 0.0], dtype=np.float64)
        self.spin_down = np.array([0.0, 1.0], dtype=np.float64)

        # ==== General constants for simulation ====
        self.TIME_STEPS = 500                # number of time steps
        self.au2ps      = 0.00002418884254     # as -> a.u. conversion
        self.timeau     = 12.409275
        self.DT         = 20 * self.au2ps * self.timeau  # time step in au

        self.FINAL_TIME = self.TIME_STEPS * self.DT
        self.DOF_E      = 2                 # number of electronic states
        self.DOF_E_SQ   = self.DOF_E * self.DOF_E
```

```
        # ==== Simulation parameters for TT-TFD ====
        self.DOF_N     = 50    # number of nuclear DOF
        self.OMEGA_MAX  = 10

        # TT constants
        self.eps       = 1e-12       # tt approx error
        self.dim       = self.DOF_N   # number of coords
        self.occ       = 10          # max occupation number
        self.MAX_TT_RANK = 10

        # ==== Simulation parameters for GQME ====
        self.MEM_TIME        = self.DT * self.TIME_STEPS
        self.HBAR           = 1
        self.MAX_ITERS       = 30
        self.CONVERGENCE_PARAM = 10.0**(-10.0)

        # ==== Parameter string for output files ====
        self.PARAM_STR  = "_Spin-Boson_Ohmic_TT-TFD_b%sG%s_e%s_" % (
           self.BETA, self.GAMMA_DA, self.EPSILON
        )
        self.PARAM_STR += "xi%swc%s_wmax%s_dofn%s" % (
           self.XI, self.OMEGA_C, self.OMEGA_MAX, self.DOF_N
        )

        # ==== Pauli matrices ====
        self.X = np.array([[0, 1], [1, 0]], dtype=np.complex128)
        self.Y = np.array([[0, -1j], [1j, 0]], dtype=np.complex128)
        self.Z = np.array([[1, 0], [0, -1]], dtype=np.complex128)
        self.I = np.eye(2, dtype=np.complex128)

# create a global instance, so you can do: pp.xx
pp = Params()
```

S.4.3 TT-TFD Simulation Driver

This driver script demonstrates two equivalent ways of running a TT-TFD spin–boson simulation: (i) a high-level call to the convenience function `tt_tfd`, and (ii) a more explicit route that builds the initial state and Hamiltonian and then calls the generic TDVP-based propagator `tt_ksl_propagator`. The outputs are the time grid `t` and the array of reduced density operators `RDO_arr`.

Script S.4.3: TT-TFD Simulation of Spin-Boson Model

```
# ---- user choices for the test ----
initial_state = 0      # 0: |up>, 1: (|up>+|down>)/sqrt2, etc.
update_type   = "rk4"   # "rk4" or "krylov"
rk4slices     = 1      # only used if update_type == "rk4"
mmax          = 4      # only used if update_type == "krylov"
verbose       = True
show_steptime = True

# ---- run simulation ----
print("Building initial state and Hamiltonian")
y0 = tt_initial_state(initial_state)
A  = tt_hamiltonian(eps=pp.eps, pp=pp)

print("Propagating")
Is_qflux_tt_tfd = True
if Is_qflux_tt_tfd:
   t, RDO_arr = tt_tfd(initial_state=0, show_steptime=True, update_type='rk4')
else:
   t, RDO_arr = tt_ksl_propagator(
      y0,
      A,
      update_type=update_type,
      rk4slices=rk4slices,
      mmax=mmax,
      RDO_arr_bench=None,
      property_fn=cal_property,
      verbose=verbose,
      show_steptime=show_steptime,
      copy_state=False,   # set True if you want to keep `y0` unchanged
      pp=pp,
      )

print("Propagation finished.")
print("RDO_arr shape:", RDO_arr.shape)
```

S.4.4 Post-Processing: Populations and Coherences

The post-processing script extracts populations and coherences from the flattened reduced density matrices stored in `RDO_arr`, and then generates diagnostic plots of ρ_{uu}, ρ_{dd}, and the real and imaginary parts of ρ_{ud} as functions of time.

Script S.4.4: Extract and plot populations and coherences

```python
# RDO_arr: shape (TIME_STEPS, 4) for a 2-level system
pop_up   = RDO_arr[:, 0].real
pop_down = RDO_arr[:, 3].real
coh_ud   = RDO_arr[:, 1]  # complex

# ---- population plot ----
plt.figure()
plt.plot(t, pop_up,   label="Pop |up>")
plt.plot(t, pop_down, label="Pop |down>")
plt.xlabel("time")
plt.ylabel("population")
plt.legend()
plt.title("Spin populations vs time")
plt.grid(True)
plt.show()

# ---- coherence plot ----
plt.figure()
plt.plot(t, coh_ud.real, label="Re rho_ud")
plt.plot(t, coh_ud.imag, label="Im rho_ud", linestyle="--")
plt.xlabel("time")
plt.ylabel("coherence")
plt.legend()
plt.title("Spin coherences vs time")
plt.grid(True)
plt.show()
```

S.4.5 State Initialization in TT-TFD

This script implements `tt_initial_state`, which builds the thermo-field initial state as an MPS/TT with one electronic site and $2\,\mathrm{DOF}_N$ bosonic sites. The function is called by the simulation driver to prepare the starting TT state for propagation.

Script S.4.5: State Initialization

```python
def tt_initial_state(istate: int) -> MPS:
    """
    Initialize the state in tensor-train (MPS) format for a TT-TFD calculation.
```

```
    Parameters
    ----------
    istate : int
        Type of initial electronic state:
        0 : spin-up
        1 : (spin-up + spin-down) / sqrt(2)
        2 : (spin-up + i * spin-down) / sqrt(2)
        3 : spin-down

    Returns
    -------
    MPS
        Initialized MPS with the chosen electronic state at the first site
        and vacuum/ground states on the remaining sites.
        QFlux uses mpsqd https://github.com/qiangshi-group/MPSQD
    """
    # Sanity check on istate
    if istate not in (0, 1, 2, 3):
        raise ValueError(f"Invalid istate={istate}. Must be in {{0, 1, 2, 3}}.")

    # ------------------------------------------------------------------------
    # Define single-site electronic tensors
    # ------------------------------------------------------------------------
    su = np.zeros((1, pp.DOF_E, pp.MAX_TT_RANK), dtype=np.complex128)
    sd = np.zeros((1, pp.DOF_E, pp.MAX_TT_RANK), dtype=np.complex128)

    su[0, :, 0] = pp.spin_up
    sd[0, :, 0] = pp.spin_down

    # Superpositions
    inv_sqrt2 = 1.0 / np.sqrt(2.0)
    e1 = inv_sqrt2 * (su + sd)
    e2 = inv_sqrt2 * (su + 1j * sd)

    # Select the initial electronic core
    electronic_cores = {
        0: su,
        1: e1,
        2: e2,
        3: sd,
    }
    first_core = electronic_cores[istate]

    # ------------------------------------------------------------------------
    # Build MPS structure
    # ------------------------------------------------------------------------
    # nbarr: local dimensions for each site
    num_sites = 1 + 2 * pp.DOF_N
    nbarr = np.full(num_sites, pp.occ, dtype=int)
    nbarr[0] = pp.DOF_E  # first site is electronic
```

```
    y0 = MPS(num_sites, nb=nbarr)
    y0.nodes.append(first_core)

    # Middle sites: identity-like / vacuum cores
    middle_core = np.zeros(
        (pp.MAX_TT_RANK, pp.occ, pp.MAX_TT_RANK),
        dtype=np.complex128
    )
    middle_core[0, 0, 0] = 1.0

    # Append 2 * DOF_N - 1 middle cores
    for _ in range(2 * pp.DOF_N - 1):
        y0.nodes.append(middle_core)

    # Last site: right boundary core with rank-1 right bond
    last_core = np.zeros(
        (pp.MAX_TT_RANK, pp.occ, 1),
        dtype=np.complex128
    )
    last_core[0, 0, 0] = 1.0
    y0.nodes.append(last_core)

    return y0
```

S.4.6 Hamiltonian Construction

S.4.6.1 Bath Parameter Discretization

This utility converts the continuous Ohmic spectral density into a finite set of discrete bath modes. It returns frequencies, couplings, and TFD mixing angles, which are subsequently used to build the TT representation of the bath Hamiltonian and system–bath couplings.

Script S.4.6: Bath Frequency Discretization

```
def discretize_ohmic(freq_count: int):
    """
    Discretize an Ohmic spectral density into `freq_count` modes.

    Returns
```

```
    -------
    freq : (N,) array
    ck   : (N,) array
    gk   : (N,) array
    thetak, sinhthetak, coshthetak : (N,) arrays
    """
    N = freq_count

    om = pp.OMEGA_C / N * (1.0 - np.exp(-pp.OMEGA_MAX / pp.OMEGA_C))

    freq = np.zeros(N, dtype=float)
    ck = np.zeros(N, dtype=float)
    gk = np.zeros(N, dtype=float)
    thetak = np.zeros(N, dtype=float)
    sinhthetak = np.zeros(N, dtype=float)
    coshthetak = np.zeros(N, dtype=float)

    for i in range(N):
        freq[i] = -pp.OMEGA_C * np.log(
            1.0 - (i + 1) * om / pp.OMEGA_C
        )
        ck[i] = np.sqrt(pp.XI * om) * freq[i]
        gk[i] = -ck[i] / np.sqrt(2.0 * freq[i])

        th = np.arctanh(np.exp(-pp.BETA * freq[i] / 2.0))
        thetak[i] = th
        sinhthetak[i] = np.sinh(th)
        coshthetak[i] = np.cosh(th)

    return freq, ck, gk, thetak, sinhthetak, coshthetak
```

S.4.6.2 Local TT Blocks

The next group of scripts defines the small building blocks (local electronic Hamiltonian, number operator, displacement operator, and TT helper routines) from which the full TT-TFD Hamiltonian is assembled.

Script S.4.7: Two-Level Hamiltonian

```python
def build_electronic_hamiltonian(epsilon: float, gamma_da: float):
    """2x2 electronic Hamiltonian in matrix form."""
    px = np.array([[0.0, 1.0],
                   [1.0, 0.0]], dtype=np.complex128)
    pz = np.array([[1.0, 0.0],
                   [0.0, -1.0]], dtype=np.complex128)
    return epsilon * pz + gamma_da * px
```

Script S.4.8: Kronecker-extend a 2x2 TT-matrix with 2*DOF modes

```python
def tt_embed_electronic(tt_He, total_boson_modes: int, occ: int):
    """
    Kronecker-extend a 2x2 TT-matrix to include 2*DOF_N bosonic modes.
    """
    return tt_kron(tt_He, tt_eye(2 * total_boson_modes, occ))
```

Script S.4.9: Number Operator

```python
def build_number_operator_local(occ: int):
    """Local harmonic number operator in matrix form."""
    return np.diag(np.arange(occ, dtype=np.complex128))
```

Script S.4.10: Create a TT with identity structure

```python
def tt_zero_like_eye(num_sites: int, occ: int):
    """Create a TT with identity structure and then zero all cores."""
    tt_obj = tt_eye(num_sites, occ)
    for i in range(num_sites):
        tt_obj.nodes[i] *= 0.0
    return tt_obj
```

Script S.4.11: Sum Local Operator

```python
def tt_sum_local_operators(num_sites: int,
                           occ: int,
                           local_mats,
                           site_coeffs,
                           eps: float):
    """
    Build sum_k site_coeffs[k] * (I ... x local_mats[k] x ... I) in TT form.
    """
    tt_total = tt_zero_like_eye(num_sites, occ)

    for k, (Mloc, coeff) in enumerate(zip(local_mats, site_coeffs)):
        tmp0 = tt_matrix(Mloc)
        tmp0.nodes[0] *= coeff

        if k == 0:
            tmp = tt_kron(tmp0, tt_eye(num_sites - 1, occ))
        elif k < num_sites - 1:
            tmp = tt_kron(tt_eye(k - 1, occ), tmp0)
            tmp = tt_kron(tmp, tt_eye(num_sites - k, occ))
        else:  # last site
            tmp = tt_kron(tt_eye(k, occ), tmp0)

        tt_total = add_tensor(tt_total, tmp, small=eps)

    return tt_total
```

Script S.4.12: Local Displacement Operator x

```python
def build_displacement_local(occ: int):
    """
    Local displacement operator (x operator) in HO basis.
    """
    D = np.zeros((occ, occ), dtype=np.complex128)
    for i in range(occ - 1):
        s = np.sqrt(i + 1.0)
        D[i, i + 1] = s
        D[i + 1, i] = s
    return D
```

S.4.6.3 Spin–Boson-Specific TT Blocks

The following scripts assemble the bath Hamiltonian and system–bath couplings in TT form using the generic blocks defined above.

Script S.4.13: TT representation of $\sum_k freq[k] a_k^\dagger a_k$

```
def tt_number_operator_physical(freq, eps: float):
    """
    TT representation of sum_k freq[k] * a_k^\dagger a_k on DOF_N sites.
    """
    N = pp.DOF_N
    numoc = build_number_operator_local(pp.occ)
    local_mats = [numoc] * N
    return tt_sum_local_operators(N, pp.occ, local_mats, freq, eps)
```

Script S.4.14: TT representation of $\sum_k gk[k] \cosh(\theta_k)(a_k + a_k^\dagger)$

```
def tt_displacement_physical(gk, coshthetak, eps: float):
    r"""
    TT representation of sum_k gk[k] cosh(theta_k) (a_k + a_k^\dagger)
    """
    N = pp.DOF_N
    D = build_displacement_local(pp.occ)
    local_mats = [D] * N
    coeffs = gk * coshthetak
    return tt_sum_local_operators(N, pp.occ, local_mats, coeffs, eps)
```

Script S.4.15: TT representation of $\sum_k gk[k] \sinh(\theta_k)(\tilde{a}_k + \tilde{a}_k^\dagger)$

```
def tt_displacement_fictitious(gk, sinhthetak, eps: float):
    r"""
    TT representation of sum_k gk[k] sinh(theta_k) (tilde a_k + tilde a_k^\dagger)
    """
    N = pp.DOF_N
    D = build_displacement_local(pp.occ)
```

```
    local_mats = [D] * N
    coeffs = gk * sinhthetak
    return tt_sum_local_operators(N, pp.occ, local_mats, coeffs, eps)
```

Script S.4.16: Lift a bosonic TT operator

```
def tt_lift_to_system(tt_boson, system_op):
    """
    Lift a bosonic TT operator to include a 2D electronic system:

    result = system_op (x) tt_boson (x) I_boson (or variations).
    """
    tt_sys = tt_matrix(system_op)
    return tt_kron(tt_sys, tt_boson)
```

Script S.4.17: $\text{left}_{\text{op}} \otimes \text{tt}_{\text{boson}} \otimes I$

```
def tt_lift_physical_with_fictitious(tt_boson, left_op, eps: float):
    """
    Construct (left_op (x) tt_boson (x) I).
    """
    tt_left = tt_matrix(left_op)
    tt = tt_kron(tt_left, tt_boson)
    tt = tt_kron(tt, tt_eye(pp.DOF_N, pp.occ))
    return tt
```

Script S.4.18: $\text{left}_{\text{op}} \otimes I \otimes \text{tt}_{\text{boson}}$

```
def tt_lift_fictitious_with_physical(tt_boson, left_op, eps: float):
    """
    Construct (left_op x I x tt_boson).
    """
    tt_left = tt_matrix(left_op)
    tt = tt_kron(tt_left, tt_eye(pp.DOF_N, pp.occ))
    tt = tt_kron(tt, tt_boson)
```

```
    return tt
```

S.4.6.4 High-Level Constructor for $-\mathrm{i}H$

The final Hamiltonian-construction script, `tt_hamiltonian`, assembles all electronic, physical, and fictitious contributions into a single TT/MPO object representing the generator $-\mathrm{i}H$ used in the time-evolution algorithms.

Script S.4.19: Build $-\mathrm{i}H$ for the TFD Spin–Boson Model

```
def tt_hamiltonian(eps: float = 1e-14):
    """
    Build -iH for the TFD spin-boson model using modular building blocks.

    Returns
    -------
    MPO (MPS-like TT object)
    """
    # --- parameters ---
    freq, ck, gk, thetak, sinhthetak, coshthetak = discretize_ohmic(pp.DOF_N)

    # --- electronic part ---
    He = build_electronic_hamiltonian(pp.EPSILON, pp.GAMMA_DA)
    tt_He = tt_matrix(He)
    tt_He = tt_embed_electronic(tt_He, pp.DOF_N, pp.occ)

    # --- physical and fictitious number operators ---
    tt_num_physical = tt_number_operator_physical(freq, eps)
    tt_Ie = tt_matrix(np.eye(2, dtype=np.complex128))

    tt_systemnumoc = tt_kron(tt_Ie, tt_num_physical)
    tt_systemnumoc = tt_kron(tt_systemnumoc, tt_eye(pp.DOF_N, pp.occ))

    tt_tildenumoc = tt_kron(tt_Ie, tt_eye(pp.DOF_N, pp.occ))
    tt_tildenumoc = tt_kron(tt_tildenumoc, tt_num_physical)

    # --- displacement operators ---
    tt_energy = tt_displacement_physical(gk, coshthetak, eps)
    tt_systemenergy = tt_kron(tt_matrix(np.array([[1, 0], [0, -1]],
                                      dtype=np.complex128)),
                        tt_energy)
```

```
    tt_systemenergy = tt_kron(tt_systemenergy, tt_eye(pp.DOF_N, pp.occ))

    tt_tilenergy = tt_displacement_fictitious(gk, sinhthetak, eps)
    tt_tildeenergy = tt_kron(tt_matrix(np.array([[1, 0], [0, -1]],
                                        dtype=np.complex128)),
                          tt_eye(pp.DOF_N, pp.occ))
    tt_tildeenergy = tt_kron(tt_tildeenergy, tt_tilenergy)

    # --- assemble H ---
    H = add_tensor(tt_He, tt_systemnumoc, small=eps)
    H = add_tensor(H, tt_tildenumoc, coeff=-1.0, small=eps)
    H = add_tensor(H, tt_systemenergy, coeff=1.0, small=eps)
    H = add_tensor(H, tt_tildeenergy, coeff=1.0, small=eps)

    # fold -i into the first core
    H.nodes[0] *= -1j

    # convert to MPO and truncate
    A = MPS2MPO(H).truncation(small=eps)
    return A
```

S.4.7 ODE solver KSL Propagator

The last script implements the generic TDVP-based time-propagation routine `tt_ksl_propagator`. Given an initial TT state and the TT/MPO generator $A = -\mathrm{i}H$, it advances the system for `TIME_STEPS` time steps, accumulating the reduced density operators computed by a user-specified observable function `property_fn`.

Script S.4.20: TT-KSL Runge-Kutta ODE solver

```
def tt_ksl_propagator(
    y0: Any,
    A: Any,
    update_type: str = "rk4",
    rk4slices: int = 1,
    mmax: int = 4,
    RDO_arr_bench: np.ndarray | None = None,
    property_fn: Callable[[Any], np.ndarray] = cal_property,
    verbose: bool = True,
    show_steptime: bool = False,
```

```
    copy_state: bool = False,
) -> tuple[np.ndarray, np.ndarray]:
    """
    Perform TT-TFD time propagation with a given initial state and Hamiltonian.

    Parameters
    ----------
    y0
        Initial TT/MPS state. If ``copy_state`` is False, this object is
        updated in-place by the propagator.
    A
        TT/MPO representing the (possibly non-Hermitian) generator, e.g. -iH.
    update_type : {"rk4", "krylov"}, optional
        Local time-stepper used in tdvp1site. Default is "rk4".
    rk4slices : int, optional
        Number of sub-slices for RK4 integration. Ignored for "krylov".
    mmax : int, optional
        Krylov subspace dimension for "krylov" updates. Default is 4.
    RDO_arr_bench : np.ndarray, optional
        Optional benchmark reduced density operator array of shape
        (TIME_STEPS, DOF_E_SQ). If provided, each step's RDO is compared
        with this reference via compare_diff.
    property_fn : callable, optional
        Function mapping the TT/MPS state to a (flattened) RDO array of shape
        (DOF_E_SQ,). Default is cal_property.
    verbose : bool, optional
        If True, print high-level progress information.
    show_steptime : bool, optional
        If True, print wall-clock time for each TDVP step.
    copy_state : bool, optional
        If True, work on a copy of `y0` instead of modifying it in-place.

    Returns
    -------
    t : np.ndarray
        1D array of simulation times of length pp.TIME_STEPS.
    RDO_arr : np.ndarray
        2D array of reduced density matrices over time with shape
        (pp.TIME_STEPS, pp.DOF_E_SQ).
    """
    n_steps = pp.TIME_STEPS
    dt = pp.DT

    # Optional copy so caller can keep original y0
    if copy_state and hasattr(y0, "copy"):
        y = y0.copy()
    else:
        y = y0

    RDO_arr = np.zeros((n_steps, pp.DOF_E_SQ), dtype=np.complex128)
```

```
    t = np.linspace(0.0, (n_steps - 1) * dt, n_steps, dtype=float)

    start_time = time.time()
    if verbose:
        print("Start propagation")
        print(f"  steps = {n_steps}, dt = {dt}, update_type = {update_type}")

    for ii, ti in enumerate(t):
        if verbose:
            print(f"Step {ii:6d}, t = {ti:.6f}")

        step_t0 = time.time()

        # TDVP one-site update
        y = tdvp1site(
            y,
            A,
            dt,
            update_type=update_type,
            mmax=mmax,
            rk4slices=rk4slices,
        )

        # Reduced density operator (or whatever property_fn returns)
        RDO_arr[ii] = property_fn(y)

        # Optional benchmark comparison
        if RDO_arr_bench is not None:
            compare_diff(RDO_arr[ii], RDO_arr_bench[ii])

        if show_steptime:
            print("  time for tdvp:", time.time() - step_t0)

    if verbose:
        print("\tTotal propagation time:", time.time() - start_time)

    return t, RDO_arr
```