30 January 2026

QFlux: Quantum Circuit Implementations of Molecular Dynamics. Part II - Closed Quantum Systems

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

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

Abstract

Quantum computers offer a powerful platform for simulating real-time molecular and materials dynamics, where coherence, entanglement, and many-body correlations govern processes such as charge transport, tunneling, and spin exchange. Building on the foundations introduced in Part I, this tutorial presents practical algorithms for quantum dynamical simulation on qubit-based hardware using the open-source QFlux framework.

We formulate molecular and spin Hamiltonians in the Pauli basis and map their propagators to quantum circuits using parity-based phase constructions and local basis rotations. Expectation values and state overlaps are obtained with the Hadamard test, or by direct measurement of Pauli strings, providing a unified protocol for extracting complex observables and variational quantities.

We introduce the Quantum Split-Operator Fourier Transform (Q-SOFT), the quantum analog of the classical SOFT method for quantum dynamics simulations, which applies kinetic and potential phase operators in conjugate bases connected by the Quantum Fourier Transform to implement second-order Trotterized

evolution. A simulation of proton transfer in an asymmetric double-well potential modeling a DNA base pair reproduces benchmark simulations.

We also present a variational time-evolution scheme based on the McLachlan principle, implemented in QFlux's VarQRTE driver. Applied to a two-spin Heisenberg model, it captures coherent spin exchange with shallow, noise-resilient circuits.

Together, these methods establish a reproducible workflow—from Hamiltonian encoding to circuit execution and measurement—bridging classical and quantum simulation and providing a practical toolkit for real-time quantum dynamics on near-term hardware.

QFlux: Quantum Circuit Implementations of Molecular Dynamics.

Part II – Closed Quantum Systems

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

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

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

Quantum computers offer a powerful platform for simulating real-time molecular and materials dynamics, where coherence, entanglement, and many-body correlations govern processes such as charge transport, tunneling, and spin exchange. Building on the foundations introduced in **Part I**, this tutorial presents practical algorithms for quantum dynamical simulation on qubit-based hardware using the open-source **QFlux** framework. We formulate molecular and spin Hamiltonians in the Pauli basis and map their propagators to quantum circuits using parity-based phase constructions and local basis rotations. Expectation values and state overlaps are obtained with the Hadamard test, or by direct measurement of Pauli strings, providing a unified protocol for extracting complex observables and variational quantities. We introduce the *Quantum Split-Operator Fourier Transform* (**Q-SOFT**), the quantum analog of the classical SOFT method for quantum dynamics simulations, which applies kinetic and potential phase operators in conjugate bases connected by the Quantum Fourier Transform to implement second-order Trotterized evolution. A simulation of proton transfer in an asymmetric double-well potential modeling a DNA base pair reproduces benchmark simulations. We also present a variational time-evolution scheme based on the McLachlan principle, implemented in QFlux's `VarQRTE` driver. Applied to a two-spin Heisenberg model, it captures coherent spin exchange with shallow, noise-resilient circuits. Together, these methods establish a reproducible workflow—from Hamiltonian encoding to circuit execution and measurement—bridging classical and quantum simulation and providing a practical toolkit for real-time quantum dynamics on near-term hardware.

1 Introduction

Quantum computers provide a natural platform for simulating the real-time evolution of molecular and material systems-phenomena such as charge transport, energy transfer, and proton tunneling that are difficult to capture with classical algorithms when quantum coher-

ence and strong correlations play a central role.[1–4] Because quantum hardware manipulates information according to the same mathematical principles that govern physical wavefunctions, it offers, in principle, a direct route to simulating time-dependent quantum dynamics rather than approximating them through increasingly expensive numerical schemes. This potential positions real-time quantum simulation as a promising frontier in computational chemistry, condensed-matter physics, and quantum information science.

This paper presents **Part II** of the QFlux[5] tutorial series, which develops practical methods for implementing closed-system quantum dynamics on qubit-based hardware. Building directly on the conceptual and numerical foundations established in **Part I**-including classical time propagation, operator structure, and cross-validation strategies-this installment advances from physical models to executable quantum algorithms. The emphasis is on showing how familiar objects from quantum dynamics, such as Hamiltonians, propagators, and observables, are translated into quantum circuits in a controlled and reproducible way.

A central task in quantum simulation is the representation of physical Hamiltonians in a form suitable for qubit registers. Accordingly, we begin by showing how molecular and spin Hamiltonians are rewritten in terms of Pauli strings,

$$H = \sum_j a_j P_j, \tag{1.1}$$

and how each term can be implemented using elementary gate sequences. This mapping provides the foundation for constructing time-evolution operators through parity-based phase circuits and basis-rotation techniques that reduce arbitrary Pauli operators to diagonal interactions. Once this representation is established, we introduce the Hadamard test as a general and versatile framework for extracting expectation values and state overlaps of the form $\langle\psi|U|\psi\rangle$, quantities that play a central role in both explicit and variational quantum dynamics.

Building on these components, we present the *Quantum Split-Operator Fourier Transform*

(Q-SOFT) algorithm, the quantum analog of the classical split-operator methods introduced in Part I. In Q-SOFT, potential and kinetic phase operators are applied in complementary bases connected by the Quantum Fourier Transform (QFT), reproducing the familiar second-order Trotterized propagation scheme used in grid-based classical simulations. We describe the construction of diagonal propagators, clarify practical QFT conventions such as endianness and terminal swaps, and discuss numerical accuracy, convergence behavior, and resource scaling. The method is demonstrated through a detailed case study of proton transfer in an asymmetric double-well potential modeling a hydrogen bond in a DNA base pair, where the quantum simulation reproduces tunneling dynamics and population transfer in close agreement with classical SOFT benchmarks.

To complement this explicit product-formula approach, the tutorial also introduces the variational formulation of quantum time evolution. Rather than decomposing the propagator into discrete gates at each time step, the system's state is represented by a parametrized ansatz whose parameters evolve according to the McLachlan variational principle. The resulting equations of motion depend on quantities such as the quantum geometric tensor and Hamiltonian response, which can be evaluated directly on quantum hardware using Hadamard-test circuits. These ideas are implemented within the QFlux `VarQRTE` driver, which stabilizes parameter updates through truncated singular-value decomposition. As an illustrative example, we simulate the real-time dynamics of a two-spin Heisenberg model, demonstrating that the variational approach captures coherent spin exchange using shallow, noise-resilient circuits suitable for near-term devices.

All simulations presented in this work are implemented using the open-source `QFlux` framework,[5] which integrates `Qiskit`,[6] `QuTiP`,[7,8] `Bosonic Qiskit`,[9] and `Strawberry Fields`[10] into a unified environment for quantum dynamics. As introduced in **Part I**, QFlux provides a seamless workflow that begins with defining a model Hamiltonian, proceeds through state preparation and time evolution, and concludes with measurement and analysis. Its modular design allows users to transition smoothly between classical and quan-

tum backends, enabling benchmarking, noise modeling, and visualization of time-dependent observables within a single interface.

This installment plays a pivotal role in the overall tutorial sequence. **Part II** establishes how closed-system dynamics are realized on quantum hardware and introduces the circuit-level building blocks used throughout the remainder of the series. **Part III** builds on these ideas by focusing on state initialization and unitary decomposition, providing systematic methods for translating abstract vectors and matrices into executable circuits. **Part IV** extends the framework to open quantum systems using Lindblad dynamics and dilation techniques. **Part V** introduces adaptive variational algorithms designed for noisy intermediate-scale quantum devices, and **Part VI** generalizes these approaches to non-Markovian dynamics with explicit memory effects.

Together with Part I, this work provides a complete and reproducible entry point into quantum simulation of time-dependent dynamics. It is intended both as a pedagogical resource for students learning how classical intuition carries over to quantum circuits, and as a practical platform for researchers developing and benchmarking quantum algorithms for molecular and materials dynamics.

2 Essentials of Qubit-Based Quantum Simulation

Quantum computers simulate physical systems by exploiting the same mathematical principles that govern quantum mechanics itself. In this section, we review the fundamental building blocks of quantum computing-**qubits**, **superposition**, **entanglement**, and **quantum gates**-and show how they combine in **quantum circuits**. These concepts form the computational foundation for the algorithms implemented in QFlux.

2.1 Qubits and Superposition States

A **qubit** (quantum bit) is the fundamental unit of quantum information: a two-level quantum system that can exist not only in one of two classical states, $|0\rangle$ or $|1\rangle$, but also in any coherent **superposition** of both:

$$|\alpha\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle , \tag{2.1}$$

where the complex coefficients α_0 and α_1, called *probability amplitudes*, satisfy the normalization condition $|\alpha_0|^2 + |\alpha_1|^2 = 1$.

When measured in the *computational basis* $\{|0\rangle , |1\rangle\}$, the qubit collapses to $|0\rangle$ with probability $|\alpha_0|^2$ or to $|1\rangle$ with probability $|\alpha_1|^2$. This probabilistic nature, combined with coherent superposition, allows quantum computers to explore many configurations simultaneously.

The state of a single qubit can be visualized on the **Bloch sphere**, where any pure state is represented as

$$|\alpha\rangle = \cos\left(\frac{\theta}{2}\right) |0\rangle + \mathrm{e}^{\mathrm{i}\phi} \sin\left(\frac{\theta}{2}\right) |1\rangle , \tag{2.2}$$

with (θ, ϕ) defining a point on the sphere. The north and south poles correspond to $|0\rangle$ and $|1\rangle$, respectively, while equatorial points represent balanced superpositions such as $|+\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$ and $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$.

2.2 Multi-Qubit Registers

A register of n qubits spans a Hilbert space of dimension 2^n:

$$|\psi\rangle = \sum_{s \in \{0,1\}^n} \alpha_s |s\rangle , \tag{2.3}$$

where $s = (s_1, \ldots, s_n)$ is a binary string specifying the computational basis state $|s\rangle = |s_1\rangle \otimes \cdots \otimes |s_n\rangle$. The 2^n complex amplitudes $\{\alpha_s\}$ define the full statevector, with corresponding measurement probabilities $P_s = |\alpha_s|^2$.

This exponential scaling of state space-n qubits encoding 2^n amplitudes-is the origin of quantum computational advantage. For example, 30 qubits can represent more amplitudes than can be stored in the memory of a classical supercomputer.

2.3 Entanglement and Quantum Correlations

Unlike classical registers, multi-qubit systems can exhibit **entanglement**, where the measurement outcomes of different qubits are intrinsically correlated. An entangled state cannot be written as a product of individual qubit states. A canonical example is the **Bell state**,

$$\left|\Phi^+\right\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \tag{2.4}$$

in which each qubit has equal probability of being found in $|0\rangle$ or $|1\rangle$, yet the two are perfectly correlated: measuring one in $|0\rangle$ guarantees that the other is $|0\rangle$, and similarly for $|1\rangle$. Entanglement is a uniquely quantum resource that underlies quantum teleportation, quantum error correction, and the speedups achieved in many quantum algorithms.

2.4 Quantum Gates and Circuits

Quantum computations are carried out through sequences of **unitary operations**, or **quantum gates**, that act on qubit registers. Each gate corresponds to a reversible linear transformation preserving normalization, such as the Pauli gates:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}. \tag{2.5}$$

Here, X performs a bit flip (NOT), and Z a phase flip. Other single-qubit gates include the Hadamard gate H that creates superpositions, and single-qubit rotations,[1]

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad R(\theta, \phi, \lambda) = \begin{bmatrix} \cos\frac{\theta}{2} & -e^{i\lambda} \sin\frac{\theta}{2} \\ e^{i\phi} \sin\frac{\theta}{2} & e^{i(\phi+\lambda)} \cos\frac{\theta}{2} \end{bmatrix}. \tag{2.6}$$

Two-qubit gates, including the **controlled-NOT (CNOT)** and **controlled-Z (CZ)**,

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad \text{CZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}, \tag{2.7}$$

generate entanglement by making the target qubit respond to the state of the control qubit: the CNOT flips the target bit, and the CZ flips its phase, but only when the control qubit is in $|1\rangle$.

A sequence of quantum gates defines a **quantum circuit**, as illustrated in Fig. 1 for the preparation of a Bell state:

$$|00\rangle \xrightarrow{H\otimes I} \frac{|00\rangle + |10\rangle}{\sqrt{2}} \xrightarrow{\text{CNOT}} \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \left|\Phi^+\right\rangle. \tag{2.8}$$

Reading from left to right, the register begins in the $|00\rangle$ state. A Hadamard gate applied to the first qubit creates a superposition, which is then entangled with the second qubit via a CNOT gate, where the first qubit acts as the control and the second as the target: This

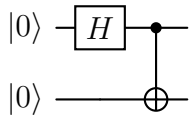

Figure 1: Quantum circuit for preparing the Bell state defined in Eq. (2.8).

simple two-qubit circuit encapsulates the fundamental principles of quantum computation-

superposition and entanglement-which form the basis of more complex algorithms.

For a hands-on start, QFlux uses the functionality of Qiskit,[6] which provides convenient tools to build and execute circuits on IBM superconducting quantum hardware as well as on classical computers. In superconducting quantum computers, qubits are realized with Josephson junctions and controlled by microwave-frequency pulses, which can be programmed, and compiled with Qiskit functionalities integrated in QFlux.

Script S.2.1 shows how to install and import the QFlux in Google Colab.[11] As a minimal example, Script S.2.2 encodes the Bell-state circuit from Fig. 1 and executes it on a simulator, to produce the output shown in Fig. 2.

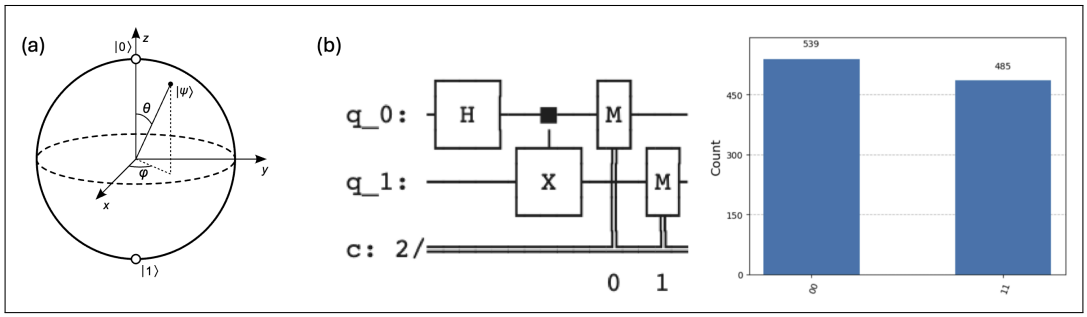

Figure 2: Fundamental concepts of qubit-based simulation. (a) The Bloch sphere illustrates a qubit's state as a point on the sphere, with poles corresponding to $|0\rangle$ and $|1\rangle$. (b) The Bell-state circuit demonstrates how simple gate combinations create superposition and entanglement-essential ingredients for simulating correlated quantum dynamics.

Figure 2 provides a schematic overview of the Bloch-sphere representation and two-qubit gate operations used to generate Bell states, showing that the corresponding measurement outcome distributions display perfect correlations, with both qubits found in the same computational state, $|00\rangle$ or $|11\rangle$, each occurring with equal probability.

With these basics in place, subsequent sections implement circuits that simulate time-dependent dynamics in model systems.

2.5 From Circuits to Simulation

In QFlux, these primitives-qubits, gates, and circuits-form the foundation for simulating molecular and spin dynamics on quantum hardware.

- **State preparation:** defines the initial wavefunction of the system, such as a vibrational or spin state.
- **Time propagation:** applies parameterized gate sequences that emulate the evolution under a Hamiltonian, e.g., via Trotterized or variational algorithms.
- **Measurement:** extracts observables such as energy, correlation functions, or population transfer probabilities.

Mastering these fundamental concepts enables the design and interpretation of all QFlux circuits-from harmonic oscillators to spin chains and proton transfer reactions.

3 Hamiltonian Simulation

3.1 Encoding Hamiltonians with Pauli Strings

To simulate quantum dynamics on a qubit-based quantum computer, the system Hamiltonian must be expressed in terms of operations natively supported by the hardware. Since qubit control is implemented using the Pauli operators I, X, Y, and Z, any Hamiltonian $\mathcal{H}$ acting on n qubits can be expanded as a linear combination of tensor products of these operators:

$$\mathcal{H} = \sum_j a_j P_j. \tag{3.1}$$

Each Pauli string

$$P_j = \bigotimes_{k=1}^{n} \sigma_k^{(j)}, \qquad \sigma_k^{(j)} \in \{I, X, Y, Z\},$$

is a tensor product of n single-qubit Pauli (or identity) operators and therefore acts on the full n-qubit Hilbert space. The amplitudes a_j determine the relative contribution of each Pauli string to the Hamiltonian. This representation-known as the *Pauli decomposition*-casts the Hamiltonian into a form that is directly compatible with quantum hardware and serves as the starting point for circuit-based implementations of time evolution.

The coefficients a_j are obtained using the Hilbert-Schmidt inner product,

$$a_j = \frac{1}{2^n} \operatorname{Tr}(P_j \mathcal{H}). \tag{3.2}$$

Because the Pauli operators form an orthogonal basis under the trace inner product, this decomposition is exact: every Hermitian operator admits a unique expansion in terms of Pauli strings. The formal computational cost of constructing the full decomposition scales as 4^n, reflecting the number of possible Pauli strings on n qubits. In practice, however, physically relevant Hamiltonians are typically sparse in this basis, allowing small coefficients to be truncated with minimal loss of accuracy.

Once the Hamiltonian is expressed in the Pauli basis, the remaining task is to implement the time-evolution operator $\mathrm{e}^{-\mathrm{i}\mathcal{H}t}$, so we can apply it to the initial state $|\psi(0)\rangle$ to obtain the time-evolved statevector $|\psi(t)\rangle$, as follows:

$$|\psi(0)\rangle = \mathrm{e}^{-\mathrm{i}\mathcal{H}t} |\psi(t)\rangle . \tag{3.3}$$

Because the Pauli strings P_j generally do not commute, the exponential of their sum cannot be implemented exactly as a product of exponentials. Instead, we employ the Trotter-Suzuki decomposition,

$$\mathrm{e}^{-\mathrm{i}\mathcal{H}t} \approx \left(\prod_j \mathrm{e}^{-\mathrm{i}a_j P_j t/m} \right)^m , \tag{3.4}$$

which converges to the exact evolution in the limit of large Trotter number m. For finite m, this approximation remains accurate provided the noncommutativity between terms is

sufficiently weak over each Trotter time step $\tau = t/m$.

This factorization reduces the simulation of a general Hamiltonian to the implementation of exponentials of individual Pauli strings. Each unitary $e^{-ia_j P_j t/m}$ can be compiled efficiently into a short quantum circuit using single-qubit rotations and entangling gates. The full time-evolution operator is then obtained by sequentially composing these circuits across all Pauli terms and Trotter steps, yielding a practical and hardware-compatible procedure for quantum simulation.

3.2 Simulating All-Z Hamiltonians

The simplest example of this process is an “all-Z" Hamiltonian, such as

$$\mathcal{H} = Z \otimes Z \otimes Z. \tag{3.5}$$

The corresponding time-evolution operator $e^{-i\tau\, Z\otimes Z\otimes Z}$ is diagonal in the computational basis and applies a phase that depends on the parity of the state. Specifically, for a basis state $|b_1 b_2 b_3\rangle$ with $b_k \in \{0, 1\}$, the operator multiplies the state by $e^{-i\tau}$ if the number of qubits in the state $|1\rangle$ is even (even parity), and by $e^{i\tau}$ if that number is odd (odd parity).

Instead of manually assigning these phases to every basis state, we can compute the parity of the register using an ancilla qubit (Fig. 3). By applying a CNOT gate from each data qubit onto the ancilla, the ancilla flips for every qubit in the $|1\rangle$ state, encoding the total parity of the system. A single rotation $R_z(2\tau)$ on the ancilla then introduces the desired phase shift. Reversing the sequence of CNOTs “uncomputes" the parity and restores the ancilla to $|0\rangle$.

This construction is highly efficient: an n-qubit all-Z Hamiltonian can be implemented using only $2n$ CNOT gates and one R_z gate as implemented in Script S.2.3. It also illustrates a general principle of Hamiltonian simulation-global phase operations can be built from local gates through clever use of parity encoding.

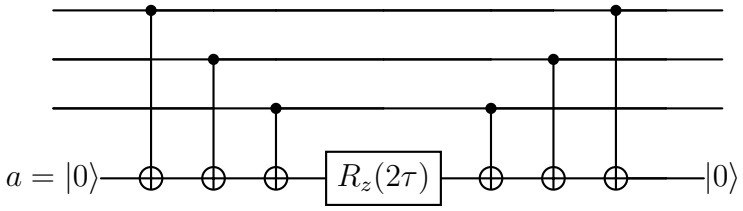

Figure 3: Quantum circuit implementing the propagator $e^{-i\tau Z\otimes Z\otimes Z}$. The ancilla accumulates the parity information through CNOTs, receives a phase rotation $R_z(2\tau)$, and is reset by reversing the CNOTs.

3.3 Extending to Arbitrary Pauli Strings

Most Hamiltonians are not composed solely of Z operators. To simulate a more general Pauli string, such as

$$\mathcal{H} = X \otimes I \otimes Y \otimes Z, \tag{3.6}$$

we make use of basis rotations that convert X and Y into Z. The Hadamard gate H transforms Z into X via conjugation, since $X = HZH$. Similarly, a rotation about the x-axis by $\pi/2$ maps Z into Y, with $Y = R_x(-\pi/2)ZR_x(\pi/2)$. By applying these transformations, we can temporarily express the entire Pauli string in the Z-basis, use the all-Z parity evolution circuit, and then invert the rotations to return to the original basis (Fig. 4).

This procedure can be formally verified. For instance, the operator $e^{i\theta(\sigma_x\otimes\sigma_z)}$ can be written as

$$e^{i\theta(\sigma_x\otimes\sigma_z)} = (H \otimes I)e^{i\theta(\sigma_z\otimes\sigma_z)}(H \otimes I), \tag{3.7}$$

showing that the X-term can be implemented using the all-Z circuit sandwiched between Hadamards on the first qubit. A similar identity holds for Y-operators when replacing H with $R_x(\pm\pi/2)$. Thus, every Pauli string can be simulated by local basis rotations, an all-Z propagation, and the inverse rotations.

Script S.2.4 provides a Python implementation that performs these steps automatically-applying the basis-change gates, invoking the all-Z parity evolution, and restoring the original basis. The design also supports an optional control qubit, though in practice the parity

can be collected on the final data qubit.

3.4 Building General Hamiltonians

Once we can simulate any individual Pauli string, we can handle general Hamiltonians expressed as

$$\mathcal{H} = \sum_k c_k P_k. \tag{3.8}$$

Each term $\mathrm{e}^{-\mathrm{i}c_k P_k t}$ is implemented as described above. Using first-order Trotterization, the combined evolution

$$\mathrm{e}^{-\mathrm{i}\mathcal{H}t} \approx \left(\prod_k \mathrm{e}^{-\mathrm{i}c_k P_k t/m} \right)^m \tag{3.9}$$

is realized by repeating the product sequence (m) times. The number of Trotter steps controls the trade-off between circuit depth and accuracy: larger (m) yields a more faithful approximation at the cost of more gates.

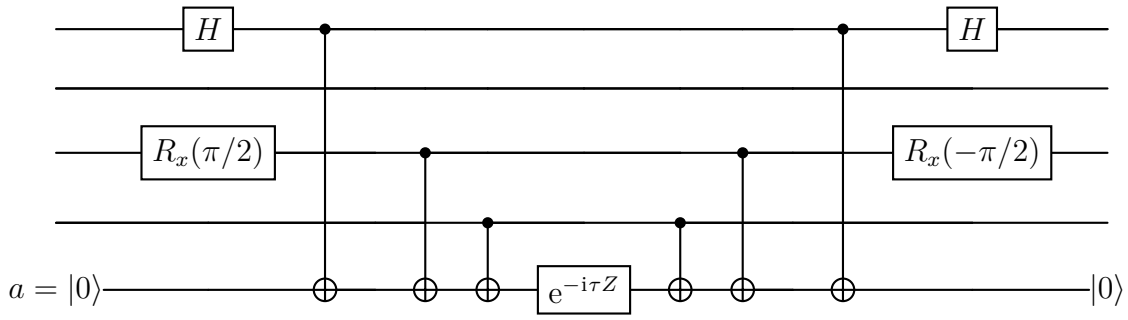

Figure 4: Quantum circuit representing the propagator $\mathrm{e}^{-\mathrm{i}\tau(X\otimes I\otimes Y\otimes Z)}$. The operators X and Y are first mapped to Z via conjugations: $X = HZH$ and $Y = R_x(-\pi/2)ZR_x(\pi/2)$. The resulting effective all-Z propagator is implemented using the parity circuit from Section 3.2.

In practice, a quantum software framework like QFlux automates this process. It first decomposes the Hamiltonian into Pauli strings, computes their coefficients via the Hilbert-Schmidt inner product, and then constructs a composite circuit by sequentially applying the corresponding unitary evolutions, as implemented in Script S.2.5, using the `exp_pauli` function.

Each circuit fragment is hardware-native and composed only of one- and two-qubit gates, making the resulting simulation compatible with NISQ devices.

3.5 Putting It All Together

To illustrate the complete workflow, consider the Hamiltonian

$$\mathcal{H} = 0.5\ Z \otimes Z + 0.3\ Y \otimes Y. \tag{3.10}$$

Starting from an initial superposition state, we can apply the Trotterized propagator generated from this Hamiltonian and then measure the resulting distribution. The first term, $Z \otimes Z$, contributes a phase conditioned on the parity of the qubits, while the $Y \otimes Y$ term introduces coherent oscillations between $|00\rangle$ and $|11\rangle$. When simulated using the Aer backend in Qiskit, the final histogram displays interference patterns characteristic of entangled spin dynamics (Fig. 5). The Script S.2.6 provides a test simulation for the Hamiltonian $H = 0.5\ \mathrm{ZZ} + 0.3\ \mathrm{YY}$, preparing a superposition state, evolving it using the `hamiltonian_simulation` function, and measuring the outcome. The output should look similar to Fig. 5 when using `plot_histogram` (counts).

Through this procedure, QFlux and similar frameworks enable users to move seamlessly from abstract Hamiltonians to executable quantum circuits. The decomposition into Pauli strings, the use of parity-based phase circuits, and the modular treatment of each operator provide a systematic method for simulating a wide variety of physical systems on contemporary quantum hardware. Whether modeling molecular interactions or spin lattices, the same underlying principle applies: any Hamiltonian can be rewritten in terms of Pauli operators, and from that foundation, a circuit-level realization naturally follows. It should thus be clear that quantum circuits do not introduce “new physics" but rather new backends for familiar quantum-dynamical simulation methods.

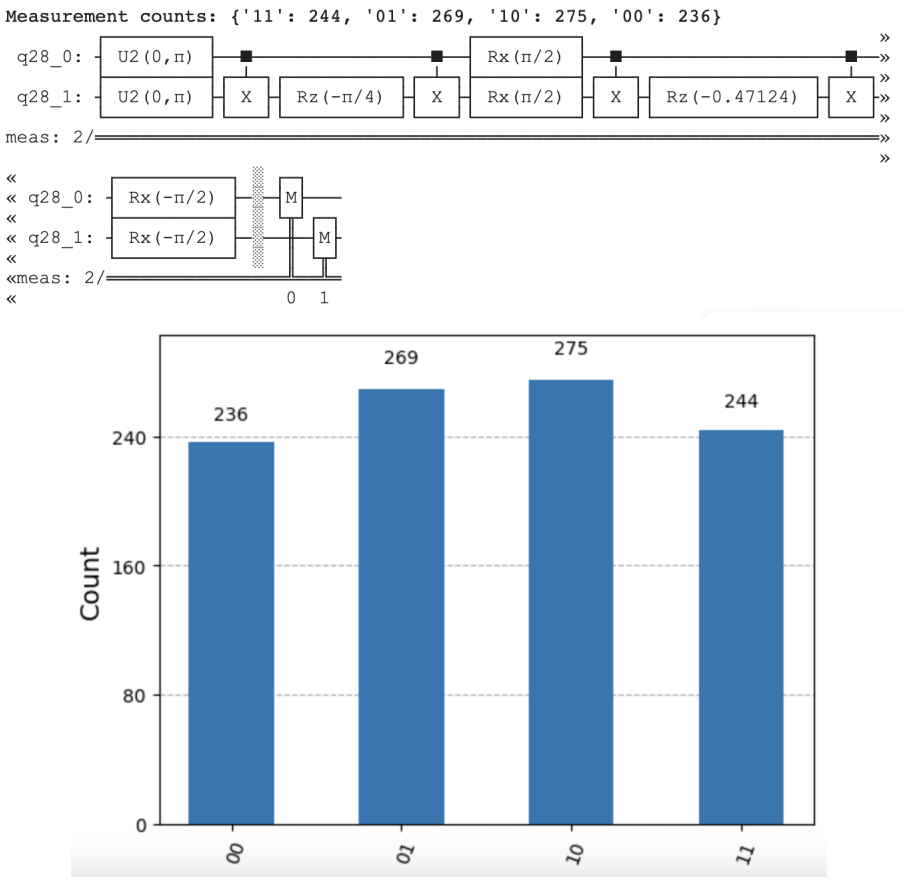

Figure 5: Output of the test code S.2.6 for Hamiltonian simulation.

3.6 Expectation Values

3.6.1 Hadamard Test: Expectation Values from Measurements of an Ancilla

The Hadamard test is a foundational quantum algorithm used to estimate the *real* and *imaginary* components of an expectation value of a unitary operator U with respect to a state $|\psi\rangle$: $\langle U\rangle = \langle\psi|U|\psi\rangle$. It employs a single ancilla qubit to coherently control the application of U and extract phase information through interference. Two circuit variants are typically used: one to measure the *real part* and another to measure the *imaginary part* of the expectation value, as illustrated in Fig. 6.

Circuit Description

In the real-part circuit (Fig. 6a), the ancilla is first placed in an equal superposition $|+\rangle$ via a Hadamard gate. It then controls the application of U on the system prepared in $|\psi\rangle$. After the operation, a second Hadamard gate recombines the interference paths before measuring the ancilla in the computational (Z) basis. To obtain the imaginary part (Fig. 6b), a phase gate $P_{-\pi/2}$ is inserted before the controlled-U, introducing a relative phase that shifts the interference pattern, isolating the imaginary component.

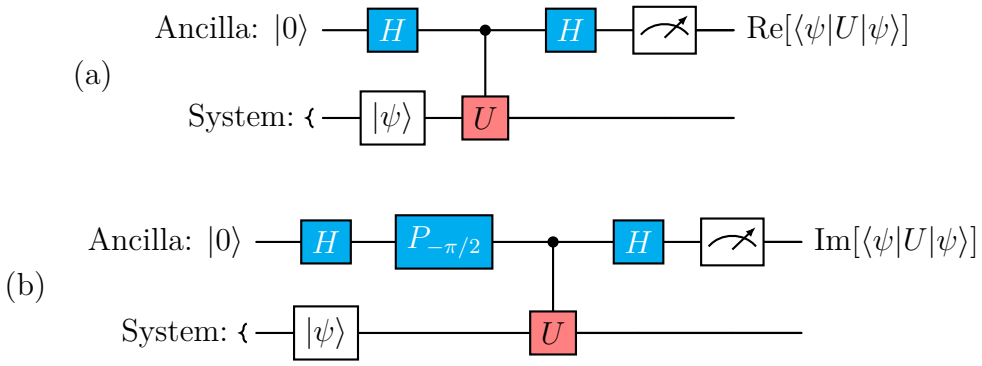

Figure 6: Circuits for measuring (a) the real and (b) the imaginary parts of $\langle\psi|U|\psi\rangle$ using the Hadamard test.

Derivation for the Real Part

1. Initialization. The ancilla starts in $|0\rangle$ and is transformed by a Hadamard gate into

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle). \tag{3.11}$$

The joint system-ancilla state becomes

$$|\Psi_0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |\psi\rangle\,. \tag{3.12}$$

2. Controlled application of U. A controlled-U acts only when the ancilla is $|1\rangle$:

$$|\Psi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle \otimes |\psi\rangle + |1\rangle \otimes U\,|\psi\rangle)\,. \tag{3.13}$$

3. Second Hadamard and measurement. Applying another Hadamard on the ancilla produces

$$|\Psi_2\rangle = \frac{1}{2}[|0\rangle \otimes (I+U)\,|\psi\rangle + |1\rangle \otimes (I-U)\,|\psi\rangle]\,. \tag{3.14}$$

The ancilla's Z-expectation value is then

$$\langle\sigma_z\rangle = \langle\Psi_2|\sigma_z \otimes \mathbb{I}|\Psi_2\rangle = \mathrm{Re}[\langle\psi|U|\psi\rangle]. \tag{3.15}$$

4. Experimental estimation. In practice, this expectation value is obtained by averaging measurement outcomes over many shots:

$$\langle\sigma_z\rangle = \frac{N_{|0\rangle} - N_{|1\rangle}}{N_{|0\rangle} + N_{|1\rangle}}, \tag{3.16}$$

where $N_{|0\rangle}$ and $N_{|1\rangle}$ are the counts of ancilla measurements yielding $|0\rangle$ and $|1\rangle$.

Imaginary Part via Phase Shift

For the imaginary component, the procedure is the same except that the ancilla receives a phase gate

$$P_{-\pi/2} = \begin{bmatrix} 1 & 0 \\ 0 & \mathrm{e}^{-\mathrm{i}\pi/2} \end{bmatrix}, \tag{3.17}$$

which changes the superposition to

$$P_{-\pi/2}\,|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle). \tag{3.18}$$

Repeating the same controlled-U and measurement process, the ancilla's Z-expectation now yields

$$\langle \sigma_z \rangle = \mathrm{Im}[\langle \psi | U | \psi \rangle]. \tag{3.19}$$

Example: Survival Amplitude

A practical example is computing the *survival amplitude*

$$\xi(t) = \langle \psi_0 | \mathrm{e}^{-\mathrm{i}\mathcal{H}t/\hbar} | \psi_0 \rangle, \tag{3.20}$$

which can be obtained via the Hadamard test by setting $U = \mathrm{e}^{-\mathrm{i}\mathcal{H}t/\hbar}$. The ancilla's measurement outcomes yield both $\mathrm{Re}[\xi(t)]$ and $\mathrm{Im}[\xi(t)]$, characterizing how the quantum state evolves under the Hamiltonian $\mathcal{H}$.

In Section 4.5.1, we show how QFlux automates this Hadamard-test-based computation for spin-chain simulations, enabling efficient estimation of survival amplitudes and correlation functions (Script S.2.31 and Script S.2.32).[12]

3.6.2 Expectation Values from Pauli-String Measurements

In many quantum algorithms, particularly variational and Hamiltonian-based simulations, U corresponds to an *observable* (a Hermitian operator) rather than a general unitary. As explained in Section 3.1 for the decomposition of the Hamiltonian as a linear combination of Pauli strings, any observable O can be decomposed as a weighted sum of *Pauli strings*:

$$O = \sum_j c_j P_j, \tag{3.21}$$

Each Pauli string $P_j = \bigotimes_{k=1}^{n} \sigma_k^{(j)}$, with $\sigma_k^{(j)} \in \{I, X, Y, Z\}$, is a tensor product of Pauli operators acting on individual qubits. Here, $c_j = \frac{1}{2^n} \mathrm{Tr}(P_j O)$.

To estimate the expectation value

$$\langle O\rangle = \langle\psi|O|\psi\rangle = \sum_j c_j \langle\psi|P_j|\psi\rangle, \tag{3.22}$$

each Pauli string P_j is measured separately:

1. **Basis rotation.** Rotate qubits so that each P_j can be measured in the computational (Z) basis. For example:

$$HXH = Z,$$
$$S^\dagger HYH^\dagger S = Z.$$

2. **Measurement.** Perform projective measurements in the Z-basis on all qubits, repeating the circuit many times.

3. **Averaging outcomes.** Each measurement shot yields a bitstring $b = (b_1, \ldots, b_n)$, where $b_k \in \{0, 1\}$. Map each bit to an eigenvalue of Z: $z_k = (-1)^{b_k}$. The value of P_j for that shot is the product of the corresponding z_k for the non-identity terms in P_j. Averaging over all shots gives

$$\langle P_j\rangle = \frac{1}{N}\sum_{s=1}^{N} p_j^{(s)}. \tag{3.23}$$

4. **Combining results.** Finally, sum over all terms:

$$\langle O\rangle = \sum_j c_j \langle P_j\rangle. \tag{3.24}$$

This measurement-based approach is equivalent in purpose to the Hadamard test—both compute expectation values—but it avoids the need for an ancilla qubit or controlled-U operations (Appendix S.1). The Hadamard test, however, becomes advantageous when evaluating *complex overlaps*, *time-evolution amplitudes*, or expectation values of *non-Hermitian*

unitaries, where interference between quantum paths directly reveals both real and imaginary components.

4 Simulating Heisenberg Hamiltonians

Building on the general framework introduced in Section 3 for simulating arbitrary Hamiltonians expressed as sums of Pauli strings, we now apply these ideas to a concrete and physically relevant example: the Heisenberg model. This model serves as a compact yet powerful case study for implementing, validating, and benchmarking quantum Hamiltonian dynamics on near-term hardware.

4.1 Two-Spin Heisenberg Dynamics on IBM Quantum Hardware

We begin with the two-spin Heisenberg model, the simplest nontrivial example that captures key features of spin-spin interactions. It provides a clear demonstration of how a model Hamiltonian can be encoded into a quantum circuit and its dynamics simulated on actual quantum devices.

The Hamiltonian for two coupled spin-$\frac{1}{2}$ particles is

$$\mathcal{H} = \frac{1}{2}\left(h_0\sigma_0^z + h_1\sigma_1^z\right) + \frac{J}{4}\left(\sigma_0^x\sigma_1^x + \sigma_0^y\sigma_1^y + \sigma_0^z\sigma_1^z\right), \tag{4.1}$$

where $h_0 = -0.5$ and $h_1 = 0.5$ are on-site energy offsets, and $J = 1$ (in atomic units) is the exchange coupling constant. The first term describes Zeeman splittings on individual spins, while the second term introduces isotropic spin-spin coupling, which exchanges excitations between $|01\rangle$ and $|10\rangle$ and shifts the $\{|00\rangle, |11\rangle\}$ subspace.

In Script S.2.7, we explicitly construct $\mathcal{H}$ and compute its short-time propagator $U(\tau) = \mathrm{e}^{-\mathrm{i}\tau\mathcal{H}}$, which serves as a numerical reference. This same operator will later be decomposed into its constituent Pauli terms and compiled into a quantum circuit following the techniques introduced in Section 3, enabling direct execution on IBM Quantum backends via Qiskit.

Classical Simulation. Before deploying the circuit on quantum hardware, we first perform a classical simulation to establish a ground-truth reference. For an evolution time of $\tau = 1$ a.u., we compute the time-evolution operator $U = \mathrm{e}^{-\mathrm{i}\mathcal{H}\tau/\hbar}$ via direct matrix exponentiation and apply it to the initial state $|00\rangle$, representing both spins in the up configuration. The resulting statevector, denoted `psi_fin` (shown in Script S.2.8), provides a benchmark for validating the quantum simulation results.

Quantum Implementation on IBM Hardware. We now implement the same evolution as a quantum circuit using IBM's Qiskit framework. The process follows three main steps: (1) quantum state initialization, (2) unitary time propagation, and (3) measurement of the final state probabilities.

1. Quantum State Initialization. We allocate a two-qubit quantum register together with a classical register for readout. Since Qiskit initializes all qubits in the $|0\rangle$ state by default, no additional state preparation is necessary (Script S.2.9).

2. Unitary Propagation. The time-evolution operator U is wrapped in a `qiskit.Operator` object and appended to the circuit (Script S.2.10). This step effectively embeds the continuous-time evolution dictated by H into the discrete quantum gate model.

3. Measurement. Finally, measurement operations are applied to both qubits to obtain the output probability distribution (Script S.2.11). The resulting circuit can be visualized with the command `entangler.draw()`.

Execution on IBM Quantum Simulators and Hardware. The constructed circuit can be executed either on simulators or real devices using IBM's cloud infrastructure. Before hardware access, an IBM Quantum account must be authenticated using an API token (Script S.2.12).

We first execute the circuit on the IBM QASM simulator, which provides a noiseless reference reproducing ideal quantum behavior (Script S.2.13). We then repeat the same procedure on a real IBM quantum processor (e.g., `ibm_manila`) or a noisy simulated backend

(e.g., `FakeManilaV2`), as illustrated in Script S.2.14.

Results and Discussion. The probability distributions obtained from the classical computation, the QASM simulator, and real IBM hardware are compared in Fig. 7. As expected, the QASM simulator reproduces the classical results exactly, confirming the correctness of the circuit and the underlying operator decomposition. In contrast, data collected from physical hardware show deviations due to gate infidelities, decoherence, and readout noise. These discrepancies highlight the limitations of current NISQ-era devices and emphasize the importance of error mitigation strategies for achieving quantitatively accurate quantum simulations.

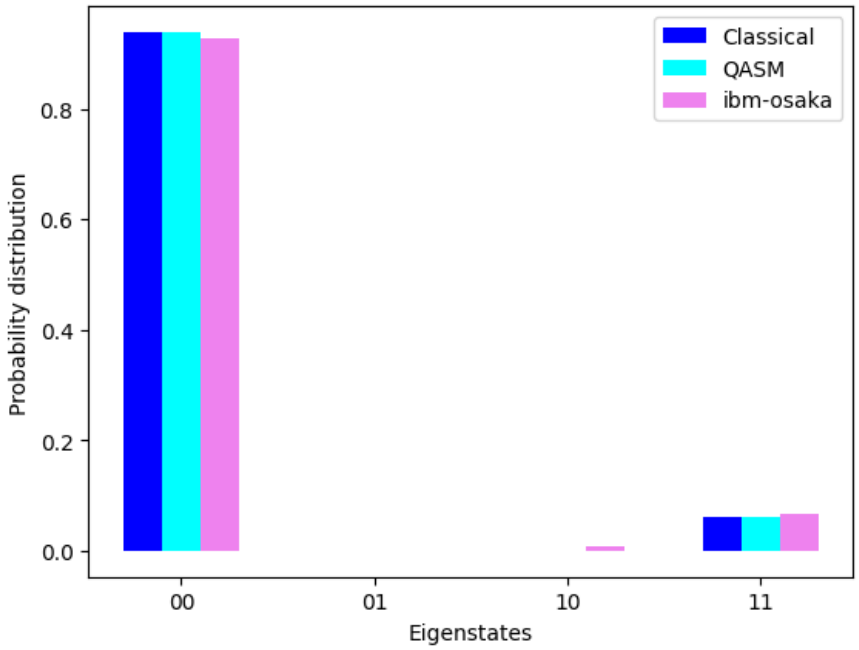

Figure 7: Comparison of probability distributions obtained from classical calculations, QASM simulation, and IBM quantum hardware for the two-spin Heisenberg model. The QASM simulator reproduces the classical benchmark exactly, while real-device data exhibit deviations caused by gate errors, decoherence, and finite sampling.

4.2 Hamiltonian Simulation of an N-Spin Heisenberg Model

We now generalize the discussion to an N-spin Heisenberg model to illustrate how Hamiltonians composed of linear combinations of Pauli tensor products can be efficiently encoded and simulated on quantum hardware.

As a representative case, we consider the Heisenberg spin chain,

$$\mathcal{H} = \sum_{n=0}^{N-1} \Omega_n \sigma_n^z - \frac{1}{2} \sum_{n=0}^{N-2} \left(J_{n,n+1}^x \sigma_n^x \sigma_{n+1}^x + J_{n,n+1}^y \sigma_n^y \sigma_{n+1}^y + J_{n,n+1}^z \sigma_n^z \sigma_{n+1}^z \right), \tag{4.2}$$

where $\sigma^{x,y,z}$ are Pauli operators acting on site n, Ω_n are on-site energy offsets, and $J_{n,n+1}^p$ ($p \in \{x, y, z\}$) represent nearest-neighbor coupling strengths.

This Hamiltonian captures a wide range of physical processes, including electron transport and spin transfer along molecular or solid-state chains. For instance, in functionalized graphene nanoribbons studied by Wang *et al.*,[13] alternating sites can host unpaired spins whose stability is modulated by Ω_n, while inter-site couplings $J_{n,n+1}^p$ depend on linker chemistry (e.g., diketone-containing regions) and can be tuned synthetically.

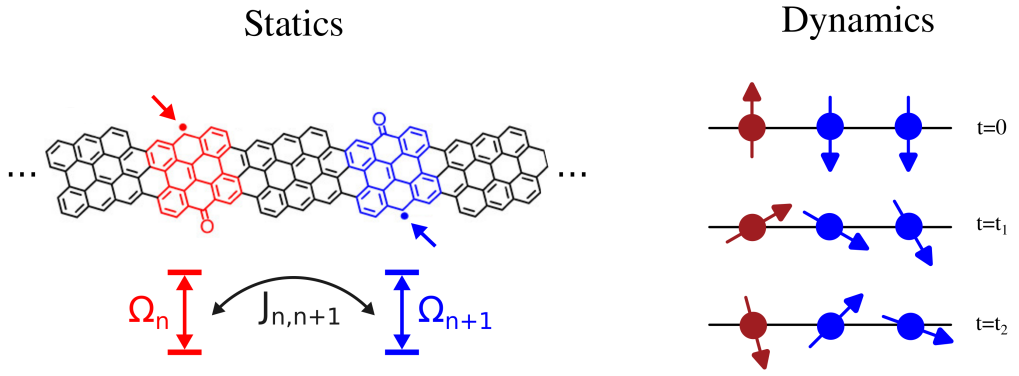

Figure 8: Static versus dynamical views of a Heisenberg spin chain. **Left:** On-site parameters Ω_n set the energy cost of spin flips, while couplings $J_{n,n+1}^p$ mediate correlations and entanglement between neighboring spins. **Right:** Under unitary evolution, the spin configuration evolves in time, while global conserved quantities (e.g., total spin in isotropic models) remain constant.

4.3 Dynamics of a Three-Site Spin Chain

To make things concrete, we adopt parameters from Fiori *et al.*[14] and restrict to a three-site chain (see Fig. 8). The values are summarized in Table 1.

Table 1: Hamiltonian parameters used in the three-site spin-chain simulation.[14]

Parameter	$n = 0$	$n \neq 0$
Ω_n	0.65	1.0
$J^x_{n,n+1}$	0.75	1.0
$J^y_{n,n+1}$	0.75	1.0
$J^z_{n,n+1}$	0.0	0.0

We initialize the system in the same state used by Fiori *et al.*: spin-up at the first site and spin-down elsewhere,

$$|\psi_0\rangle = |\uparrow\downarrow\downarrow\rangle = |\uparrow\rangle \otimes |\downarrow\rangle \otimes |\downarrow\rangle\,, \qquad |\uparrow\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |\downarrow\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{4.3}$$

Because this model is classically tractable, it provides a convenient benchmark for validating quantum-circuit results. In practice, the full time evolution can be simulated with QFlux in just a few lines, as shown in Script S.2.15.

Hamiltonian Encoding in Qiskit: To run the same model on a quantum computer, we encode its Pauli decomposition using Qiskit.[6] Each interaction acts nontrivially only on adjacent sites, e.g.,

$$\sigma^y_n \sigma^y_{n+1} = I \otimes \cdots \otimes Y_n \otimes Y_{n+1} \otimes \cdots \otimes I. \tag{4.4}$$

Qiskit's `SparsePauliOp` offers a compact representation: each Pauli string is specified as a string and scaled by its coefficient; the Hamiltonian is the sum of these terms. Script S.2.16 shows a helper function that assembles the site-local contributions.

4.4 Constructing the Full Heisenberg Hamiltonian

The full Hamiltonian for an N-site spin chain is constructed iteratively for each site, as shown in Script S.2.17. This function generates the Hamiltonian based on user-defined interaction coefficients or assumes uniform coupling if none are provided. The correctness of the Hamiltonian can be verified by calling the function and inspecting its operator representation, as demonstrated in Script S.2.18.

4.4.1 Implementing Real-Time Dynamics with Trotterization

Solving formally the time-dependent Schrödinger equation, we obtain:

$$|\Psi(t)\rangle = \mathrm{e}^{-\mathrm{i}\mathcal{H}t/\hbar}\,|\Psi(0)\rangle\,. \tag{4.5}$$

However, applying the propagator to the initial state is challenging due to the complexity of exponentiating the Hamiltonian which is typically a sum of non-commuting operators (e.g., $\mathcal{H} = A + B$). A practical approach is to approximate the exponential, as follows:

$$\mathrm{e}^{\delta(A+B)} = \mathrm{e}^{\delta A}\cdot\mathrm{e}^{\delta B} + O\left(\delta^2\right). \tag{4.6}$$

For small values of δ, this approximation is sufficiently accurate. The code shown in Script S.2.19 implements the time evolution operator for the Heisenberg Hamiltonian, which is generalizable to any Hamiltonian expressed in terms of Pauli matrices.

4.4.2 Compact Trotterization Scheme

While the propagator can be implemented using built-in quantum evolution methods, a more efficient approach is to manually encode the Trotter decomposition for Hamiltonians with one- and two-qubit Pauli operators. This method reduces circuit depth while maintaining accuracy. The key idea is to represent one-qubit terms as rotation gates (A) and two-qubit terms using an optimal decomposition (B,C). The two-qubit terms are grouped into layers

targeting even- and odd- indexed qubits separately. The objective is to construct circuits corresponding to a single Trotter step:

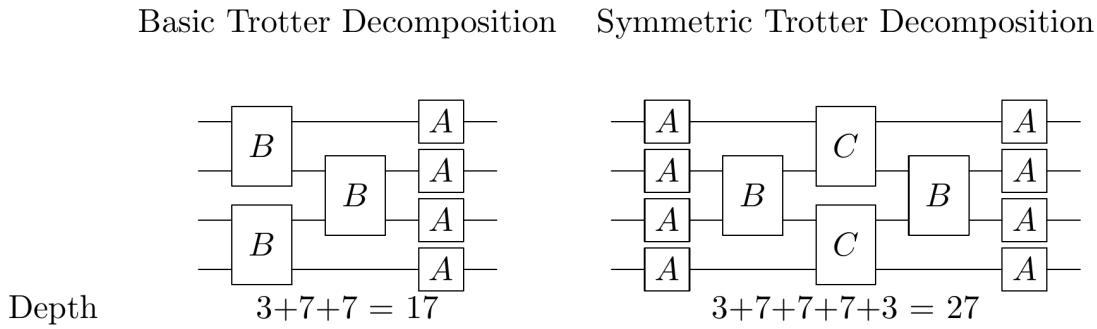

Figure 9: Quantum circuit implementing the basic and symmetric Trotter decomposition for Hamiltonians with two-qubit Pauli operators.

Sorting Hamiltonian Terms by Interaction Order: To systematically organize the Hamiltonian terms, we separate them into one-qubit, even two-qubit, and off two-qubit terms. This is done using functions that process Qiskit `SparsePauliOp` Hamiltonians (Script S.2.20 and Script S.2.21).

Encoding Single-Qubit Terms: Each one-qubit term is implemented using a corresponding rotation gate ($R_X(\theta) = \mathrm{e}^{-\mathrm{i}\frac{\theta}{2}X}$, $R_Y(\theta) = \mathrm{e}^{-\mathrm{i}\frac{\theta}{2}Y}$, or $R_Z(\theta) = \mathrm{e}^{-\mathrm{i}\frac{\theta}{2}Z}$), with rotation angle θ derived from the exponential argument:

$$\mathrm{e}^{-\mathrm{i}\frac{\theta}{2}X} = \begin{bmatrix} \cos\theta/2 & -\mathrm{i}\sin\theta/2 \\ -\mathrm{i}\sin\theta/2 & \cos\theta/2 \end{bmatrix}, \ \mathrm{e}^{-\mathrm{i}\frac{\theta}{2}Y} = \begin{bmatrix} \cos\theta/2 & -\sin\theta/2 \\ \sin\theta/2 & \cos\theta/2 \end{bmatrix}, \ \mathrm{e}^{-\mathrm{i}\frac{\theta}{2}Z} = \begin{bmatrix} \mathrm{e}^{-\mathrm{i}\theta/2} & 0 \\ 0 & \mathrm{e}^{\mathrm{i}\theta/2} \end{bmatrix}, \tag{4.7}$$

where, $\theta = 2h_i\tau$.

Encoding Two-Qubit Terms: For two-qubit terms, we use the optimal $U(4)$ decomposition:

$$U = (A_1 \otimes A_2)N(\alpha, \beta, \gamma)(A_3 \otimes A_4), \tag{4.8}$$

where

$$N(\alpha, \beta, \gamma) = \exp\{\mathrm{i}(\alpha\sigma_x \otimes \sigma_x + \beta\sigma_y \otimes \sigma_y + \gamma\sigma_z \otimes \sigma_z)\}. \tag{4.9}$$

The required parameters are:

$$\theta = \frac{\pi}{2} - 2\gamma, \quad \phi = 2\alpha - \frac{\pi}{2}, \quad \lambda = \frac{\pi}{2} - 2\beta. \tag{4.10}$$

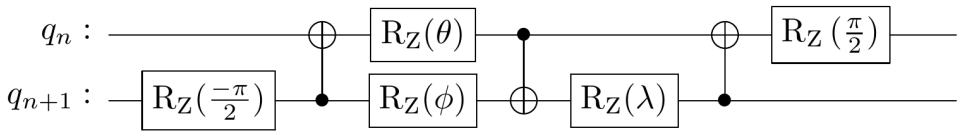

Figure 10: Quantum circuit for implementing a general two-qubit Pauli operator.

Note that $\alpha = J^x_{n,n+1}\tau, \beta = J^y_{n,n+1}\tau, \gamma = J^z_{n,n+1}\tau$ due to the connection to the exponential argument. We note that this circuit can support disconnected 2-qubit operators as long as the first index corresponds to the first wire and the second index to the second wire. The following function in Script S.2.22 implements the most general approach.

Constructing the Full Trotter Circuit: Finally, we assemble the full Trotter circuit, supporting both basic (BCA) and symmetric (ACBCA) decompositions, as shown in Script S.2.23.

This method reduces circuit depth per Trotter layer to 15 gates (basic) and 23 gates (symmetric) while maintaining compatibility with Qiskit's built-in methods (see Script S.2.24).

4.4.3 Initializing a Quantum Circuit with Qiskit

The number of quantum bits (qubits) required for a simulation depends on the problem size, while classical bits are used for recording measurement outcomes. We begin by creating a quantum circuit initialized in the vacuum state $|0\rangle \otimes |0\rangle \otimes |0\rangle = |000\rangle$ (Script S.2.25):

Customizing the Initial State: We can modify the vacuum state by applying bit-flip operations (X gates) to obtain a specific state. For instance, the state $I|0\rangle \otimes X|0\rangle \otimes X|0\rangle = |011\rangle$ can be prepared from vacuum by applying X gates, as shown in Script S.2.26, or by amplitude encoding as shown in Script S.2.27.

Applying the Time Evolution Operator: Once the initial state is prepared, we apply

the time evolution operator and check the overall structure and depth of the quantum circuit (Script S.2.28).

This workflow provides a simple example of how to initialize quantum circuits by setting custom initial states, and how to apply time-evolution operators efficiently. The transpilation step ensures optimal execution on quantum hardware or simulators.

4.5 Qubit-Based Quantum Experiments

Quantum experiments can be executed either on a quantum computer such as an **IBMQ device**, or on a classical simulator, such as the **Statevector Simulator**, which employs linear algebra (e.g., matrix-matrix and matrix vector multiplications) to simulate the evolution of the statevector along the quantum circuit. To ensure computational feasibility, quantum circuits should maintain a relatively shallow depth ($\leq$ 100 linear operations). When executing on quantum hardware, transpilation is necessary to map circuit operations onto the native gate set of the device.

Statevector-Based Quantum Simulation: We start by implementing an iterative statevector simulation to propagate the quantum state over discrete time steps. The Script S.2.29 initializes a quantum state, applies the desired quantum evolution, and updates the state iteratively. Since the `statevector_simulator` backend supports direct state initialization, reinitialization at each iteration remains computationally feasible. However, this approach would not be practical for implementation on quantum hardware, since it would require quantum state tomography which scales exponentially like 4^N, with N the number of qubits.

Computing the Survival Amplitude: We now execute the iterative propagation using a defined initial state and time evolution operator to compute the *absolute value of the survival amplitude*:

$$|\langle\psi_0|\psi_t\rangle| = \left|\langle\psi_0| \, \mathrm{e}^{-\mathrm{i}\mathcal{H}t/\hbar} \, |\psi_0\rangle\right|. \tag{4.11}$$

The corresponding simulation code is shown in Script S.2.30. This implementation en-

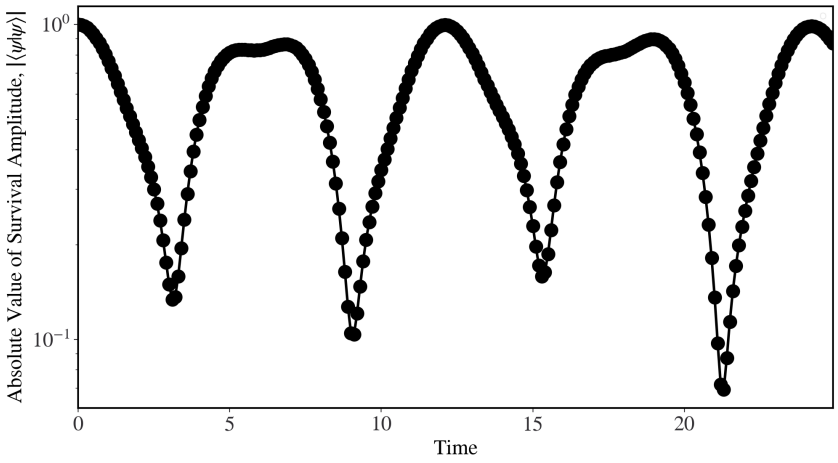

Figure 11: Absolute value of the survival amplitude calculated using the statevector method, in agreement with the classical benchmark.

ables efficient simulation of quantum state evolution using an iterative statevector approach, allowing direct computation of dynamical observables for validation against classical benchmarks.

In summary, the Hadamard test allows us to estimate the real and imaginary parts of $\langle\psi|\,U\,|\psi\rangle$ by using an ancilla qubit which is measured in the computational basis. By averaging over multiple measurements, we can accurately estimate expectation values, as well as correlation functions such as the survival amplitude $\xi(t) = \langle\psi_0|\psi_t\rangle$ with $|\psi_t\rangle = \mathrm{e}^{-\mathrm{i}\mathcal{H}t/\hbar}\,|\psi_0\rangle$. Section 4.5.1 shows how to use QFlux to implement the Hadamard test and obtain the survival amplitude of a spin chain, based on the dynamics simulation as implemented in Script S.2.31.[12]

4.5.1 Hadamard Test Function

The Hadamard test circuit is constructed by setting up the ancilla in an equal superposition, initializing the wavefunction, and applying a controlled unitary corresponding to the expectation value of interest. Measurement of the ancilla in the computational basis provides the counts $N_{|0\rangle}$ and $N_{|1\rangle}$ (i.e., numbers of ancilla measurements with outcomes $+1$ and -1, respectively) necessary to compute the real and imaginary components of the expectation

value, according to Eq. (7.2), as shown in Script S.2.32.

4.5.2 Processing the Hadamard Test Results

The expectation values are obtained from the average of ancilla measurements, according to Eq. (7.2), as shown in the post-processing procedure given in Script S.2.33.

4.6 Executing the Hadamard test for an Operator

Using the `time_evo_op` for a small propagation time step, we generate the controlled unitary. We then execute the Hadamard test for all times by propagating the initial state using the controlled unitary, and we compute the real and imaginary components of the expectation value using the Scripts S.2.34 and S.2.35. For comparison, the initial state and time evolution operators are the same as those used for the spin chain statevector simulation.

Fig. 12 shows that the results of the Hadamard test agree with the calculations based on statevector simulations. The statistical noise in these noiseless simulations can be eliminated by increasing the number of shots at the expense of additional execution time.

5 The Q-SOFT Algorithm

The split-operator Fourier transform (SOFT) algorithm[15–18] advances a wavefunction over a short time step τ by alternating between potential- and kinetic-energy propagators that are diagonal in complementary bases. Classically, fast Fourier transforms (FFTs) shuttle between position and momentum grids. Its quantum analog, *Q-SOFT*, uses the quantum Fourier transform (QFT) to move between computational (position) and momentum bases on n qubits representing $N = 2^n$ grid points.

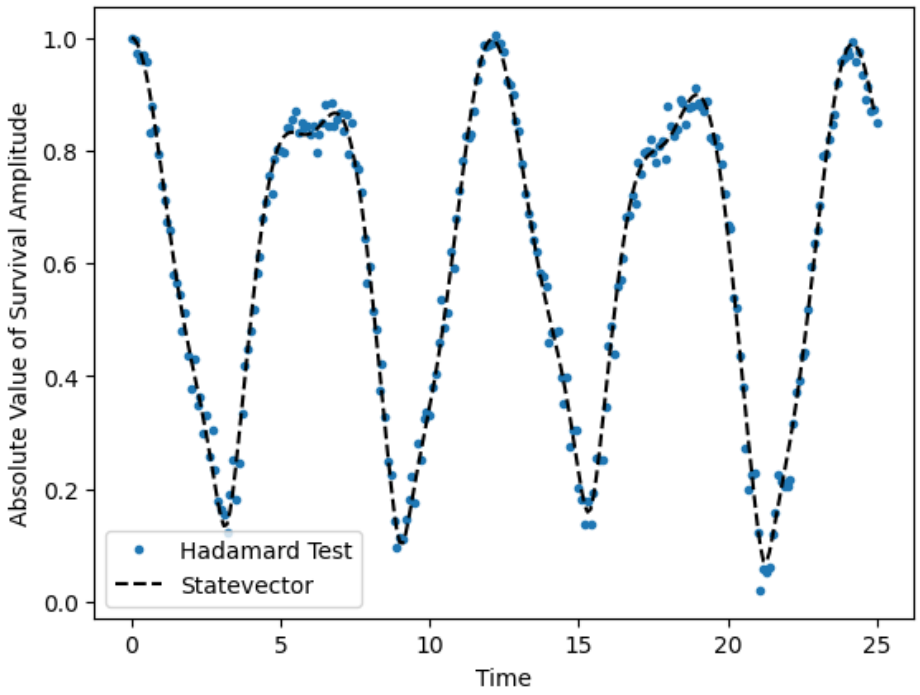

Figure 12: Absolute value of the survival amplitude for spin-chain dynamics, calculated using the Hadamard test (blue circles) and statevector simulations (dashed line).

5.1 Discretization and Encoding

We discretize a 1D coordinate $x \in [x_{\min}, x_{\max})$ into N points $x_j = x_{\min} + j, \Delta x$ with spacing $\Delta x = (x_{\max} - x_{\min})/N$. The continuous state $\psi(x)$ is mapped to an N-component vector

$$|\psi\rangle = \sum_{j=0}^{N-1} \psi(x_j) \, |j\rangle \, , \qquad \sum_{j=0}^{N-1} |\psi(x_j)|^2 \Delta x = 1, \tag{5.1}$$

encoded in the computational basis of n qubits. The conjugate momentum grid is $p_k = \frac{2\pi}{N,\Delta x}\left(k - \frac{N}{2}\right)$ with spacing $\Delta p = \frac{2\pi}{N\Delta x}$ (other phase conventions are possible, Section 5.3).

5.2 Time Evolution via Second-Order Trotterization

For a Hamiltonian $\mathcal{H} = T + V$ with $T = \hat{p}^2/2m$ and $V = V(\hat{x})$, the order-$\mathcal{O}(\tau^3)$ Strang splitting advances the state as[15–18]

$$|\psi(t+\tau)\rangle \approx \mathrm{e}^{-\mathrm{i}V\tau/2}\mathrm{e}^{-\mathrm{i}T\tau}\mathrm{e}^{-\mathrm{i}V\tau/2}\,|\psi(t)\rangle\,. \tag{5.2}$$

In Q-SOFT, the kinetic step is performed in the momentum basis, reached by the QFT. One elementary time-step therefore implements

$$|\psi(t+\tau)\rangle = U_V\; U_{\mathrm{QFT}}\; U_T\; U_{\mathrm{QFT}}^{\dagger}\; U_V\;\; |\psi(t)\rangle\,, \qquad U_V \equiv \mathrm{e}^{-\mathrm{i}V(\hat{x}),\tau/2}, \quad U_T \equiv \mathrm{e}^{-\mathrm{i}\hat{p}^2\tau/2m}. \tag{5.3}$$

Repeating Eq. (5.3) for $N_{\mathrm{steps}} = t_{\max}/\tau$ mirrors classical SOFT propagation.[15–18]

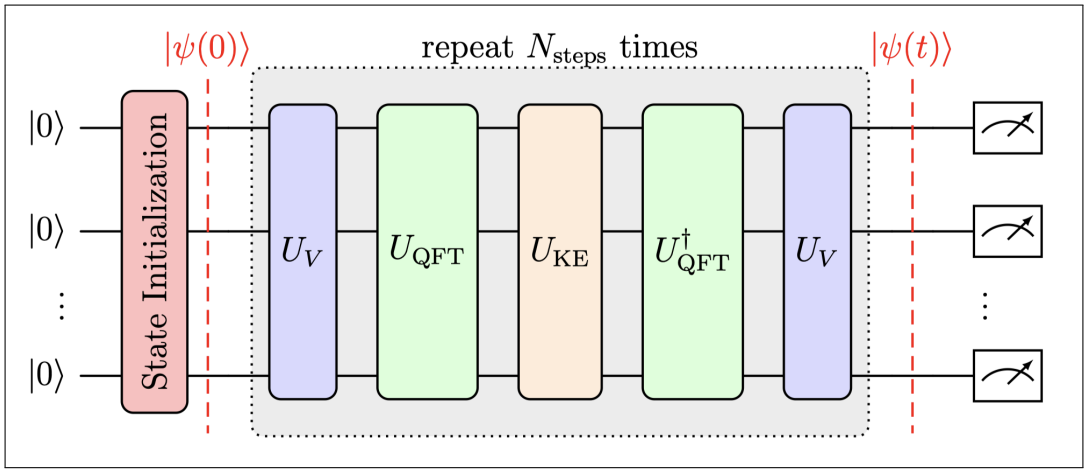

Figure 13: Circuit schematic of one Q-SOFT time step. Diagonal potential and kinetic propagators are implemented as phase operators in the position and momentum bases, respectively. The QFT (and its inverse) mediate between the two representations.

5.3 QFT Conventions and Momentum Ordering

We adopt the N-point QFT on n qubits,

$$U_{\mathrm{QFT}}\,|j\rangle = \frac{1}{\sqrt{N}}\sum_{k=0}^{N-1}\mathrm{e}^{2\pi\mathrm{i}\,jk/N}\,|k\rangle\,. \tag{5.4}$$

With `do_swaps=True`, Qiskit's QFT includes terminal swaps that restore big-endian ordering. Consistency of (i) endianness, (ii) whether swaps are present, and (iii) the definition of p_k is essential so that U_T remains diagonal in the QFT basis. If terminal swaps are used, construct $U_T = \mathrm{diag}\left(\mathrm{e}^{-\mathrm{i}p_k^2\tau/2m}\right)$ in that ordering.

5.4 Constructing Diagonal Propagators

Both U_V and U_T are diagonal matrices of phases. They can be implemented three equivalent ways:

1. **Direct operator embedding** (as in the code snippet): provide the full $2^n \times 2^n$ matrix to an `Operator` and append it. This is convenient for small n and exact in simulation.

2. **Walsh / Pauli decomposition**: expand the diagonal phases in a Walsh-Hadamard or Z-Pauli basis and realize them as products of controlled phase rotations. Depth scales with the number of significant coefficients (Appendix C).

3. **Programmable phase gradient**: for smooth $V(x)$, approximate U_V by a low-degree polynomial in x mapped to powers of a binary-encoded position register, yielding $\mathcal{O}(n)$ multi-controlled R_Z gates.

In all cases, ensure that U_V encodes the *half-step* phase $\tau/2$, while U_T encodes the *full* step τ, per Eq. (5.2). Using a full-step U_V twice per step would overdamp the dynamics.

5.5 Resource Estimates

A naive implementation incurs:

- QFT (and inverse): depth $\mathcal{O}(n^2)$ with controlled-phase ladders; $\mathcal{O}(n \log n)$ with approximate-QFT.[19]

- Diagonal phases: at most $\mathcal{O}(2^n)$ one-qubit R_Z if embedded; $\mathcal{O}(n2^n)$ two-qubit controls in worst-case Pauli/Walsh synthesis, but often far fewer due to structure.

- Per time step: $2 \times \mathrm{QFT} + 2 \times U_V + 1 \times U_T$.

For statevector simulation (no noise), operator embedding is perfectly acceptable up to $n \sim 10$–12. On hardware, approximate-QFT and sparse phase synthesis reduce depth.

5.6 Numerical Accuracy and Diagnostics

Second-order splitting yields a global error $\mathcal{O}(t_{\max}\tau^2)$; reduce τ to check convergence. Additional best practices:

1. **Aliasing**: choose x-domain wide enough that probability at boundaries remains negligible over $t_{\max}$; otherwise include absorbing layers.

2. **Normalization**: monitor $\sum_j |\psi_j|^2 \Delta x$; statevector simulators preserve norm exactly, but resampling/interpolation in analysis must respect Δx.

3. **Momentum convention check**: propagate a narrow Gaussian and confirm that free-particle spreading matches analytics with your p_k ordering.

5.7 Implementation Notes

For clarity and reproducibility:

1. Initialize with `qc.initialize(psi_0, q[:], normalize=True)`.

2. Prebuild U_V, U_T, QFT, and $\mathrm{QFT}^\dagger$ once and reuse them in the loop.

3. Use the statevector backend for benchmarking; `shots` are ignored.

4. If you later add measurements, include a classical register explicitly.

6 Case Study: Proton Transfer in a Double-Well

We illustrate Q-SOFT by simulating proton transfer in an asymmetric double-well potential modeling the hydrogen bond in a DNA A-T base pair. The potential is

$$V(x) = \alpha \left[0.429 \left(\frac{x}{x_0} \right) - 1.126 \left(\frac{x}{x_0} \right)^2 - 0.143 \left(\frac{x}{x_0} \right)^3 + 0.563 \left(\frac{x}{x_0} \right)^4 \right], \tag{6.1}$$

where $x_0 = 1.9592$ represents half the separation between the two minima of the double-well potential $V(x)$, and $\alpha = 0.0367493$ in atomic units. In this model, the coordinate x describes the position of the proton in an individual A-T base pair as it undergoes tautomerization from the energetically favored **amino-keto A-T form** to the isomeric **imino-enol A*-T* form**. This process plays a key role in proton transfer mechanisms responsible for quantum tunneling effects in biological systems.

6.1 Discretization, Initial State, and Units

We work in atomic units with the proton mass $m = m_p$. The position grid spans $[x_{\min}, x_{\max}]$ symmetrically around zero to include both wells, with $N = 2^n$ points. Let $\Delta x = (x_{\max} - x_{\min})/N$ and $\Delta p = 2\pi/(N\Delta x)$. The initial state is a minimum-uncertainty Gaussian centered in the right well,

$$\psi_0(x) = \frac{1}{(\pi\sigma_x^2)^{1/4}} \exp\left[-\frac{(x - x_c)^2}{2\sigma_x^2} + ip_c(x - x_c) \right], \tag{6.2}$$

sampled on the grid and normalized so that $\sum_j |\psi_0(x_j)|^2 \Delta x = 1$. We choose a time step τ (e.g., $\sim$ 0.5–1.0 fs) small enough to resolve the highest local curvature of $V(x)$ and the Nyquist frequency implied by Δx.

6.2 Classical SOFT Baseline

The classical reference uses the same grid and time step as Q-SOFT:

$$\psi(t+\tau) = \mathrm{e}^{-\mathrm{i}V\tau/2}; \mathrm{FFT}^{-1}\mathrm{e}^{-\mathrm{i}p^2\tau/2m}\mathrm{FFT}; \mathrm{e}^{-\mathrm{i}V\tau/2}\psi(t). \tag{6.3}$$

We benchmark at $t = 30$ fs and record the probability density $\rho(x,t) = |\psi(x,t)|^2$.

6.3 Quantum Circuit Realization

Q-SOFT mirrors the classical steps with QFTs:

1. Prepare $|\psi_0\rangle$ on n qubits via `initialize`.

2. For each of the N_{steps} time steps, apply
   $U_V; U_{\text{QFT}}; U_T; U_{\text{QFT}}^\dagger; U_V$
   where $U_V = \mathrm{diag}\left(\mathrm{e}^{-\mathrm{i}V(x_j)\tau/2}\right)$ and $U_T = \mathrm{diag}\left(\mathrm{e}^{-\mathrm{i}p_k^2\tau/2m}\right)$.

3. Simulate with the statevector backend and extract $\rho(x,t) = |\psi(x,t)|^2$; normalize by Δx when comparing to continuous densities.

U_V and U_T are appended as `Operator` objects; QFT and QFT† are `QFT` circuit objects with `do_swaps=True`. This matches the classical FFT momentum ordering used to define p_k.

The corresponding codes for implementing this model potential are provided in Script S.2.36. Q-SOFT simulations of adenine–thymine tautomerization can be efficiently executed using the `QFlux` framework, as demonstrated in Script S.2.37. This includes direct performance and accuracy comparisons with classical SOFT and Runge–Kutta simulations.[20–22] By default, the initial quantum state is a Gaussian wavepacket representing a coherent state centered at $x = 1.5$, $x_0 = 2.939$, located near the right minimum of the double-well potential $\omega = \left[\frac{1}{m_p}\frac{\partial^2 V(x;x_0)}{\partial x^2}\right]^{1/2}$, as shown in Script S.2.38. The position and momentum operators, together with their exponential propagators, are defined as described in Script S.2.39.

Real-time propagation is then carried out for $t = 30$ fs using the quantum circuit shown in Fig. 13. The circuit, constructed as detailed in Script S.2.40 and executed by Script S.2.41, is initialized in the state $|\psi_0\rangle$ and evolves through a series of unitary transformations representing the kinetic and potential propagators. Transitions between position and momentum representations are achieved using the Quantum Fourier Transform (QFT) operator from `qiskit`, corresponding to the inverse Fourier transform implementation in `numpy`. The propagation loop performs a fixed number of iterations, updating the quantum state at each time step.

The Q-SOFT results are benchmarked against classical SOFT simulations performed on a conventional computer, as shown in Script S.2.42. Finally, Fig. 14 presents a direct comparison between Q-SOFT and SOFT outcomes, generated using the plotting routines in Script S.2.43. The strong agreement between the two approaches validates the accuracy of Q-SOFT in capturing the quantum dynamics of adenine–thymine tautomerization, governed by proton transfer across the model double-well potential.

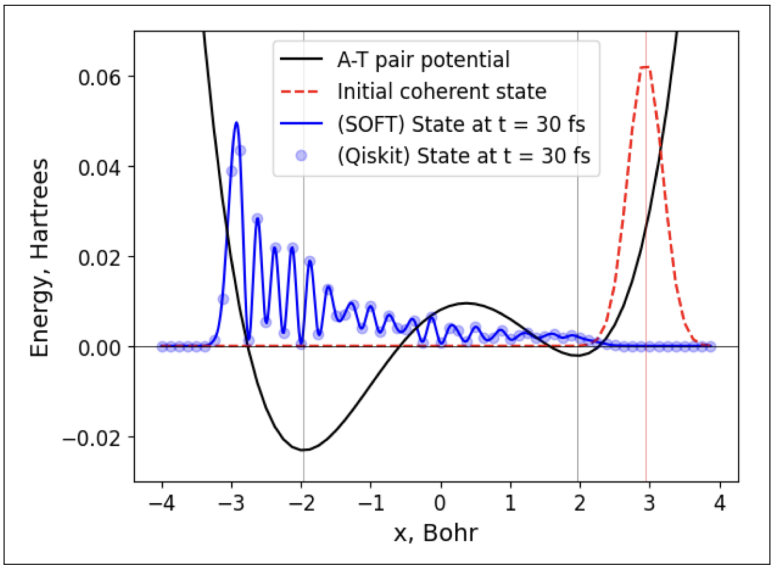

Figure 14: Application of Q-SOFT to proton transfer in a DNA A-T base pair model, described by the asymmetric double-well potential $V(x)$ of Eq. (6.1). Probability densities at the initial time (red) and after $t = 30$ fs (blue) obtained by classical SOFT (solid line) and Q-SOFT (points). The close agreement validates the circuit construction and basis conventions.

6.4 Validation and Best Practices

To ensure correctness:

- **Half-/full-step phases:** verify that the potential operator is a half-step $\tau/2$ and the kinetic operator a full step τ. Applying a full-step U_V twice per step doubles the potential kick.
- **QFT ordering:** keep `do_swaps` and endianness consistent between the circuit and how p_k are enumerated when constructing U_T.
- **Grid effects:** confirm negligible density at $x_{\min\,/\,\max}$ at $t = 30$ fs or add absorbing boundaries.
- **Convergence:** halve τ and/or increase n to check that observables (e.g., well population) are stable within tolerance.

6.5 What the Comparison Shows

Across identical grids and time steps, the classical and quantum implementations produce indistinguishable densities at $t = 30$ fs within plotting resolution. Small pointwise differences reflect (i) discrete sampling of $\rho(x)$ from the statevector vs. cubic interpolation used for the classical curve in post-processing and (ii) finite Trotter error controlled by τ. Agreement confirms that (a) phase operators are correctly parameterized, (b) QFT conventions match the FFT baseline, and (c) the circuit realizes the intended Strang splitting.

6.6 Scalability and Hardware Outlook

On fault-tolerant hardware, diagonal phase operators admit efficient synthesis leveraging problem structure (Walsh/Pauli decompositions with coefficient truncation), and approximate-QFT reduces depth with bounded error. On NISQ devices, noise-sensitive QFT layers typically limit usable n and N_{steps}. Nonetheless, shallow instances (few qubits,

few steps) can serve as calibration and verification targets when paired with high-fidelity simulators.

Reproducibility checklist (matching the code):

1. Use identical x/p grids and time step τ for classical and quantum runs.
2. Build $U_V = \exp[-\mathrm{i}V(x)\tau/2]$ and $U_T = \exp[-\mathrm{i}p^2\tau/2m]$ with the same ordering used by QFT/FFT.
3. For plotting on the same axes as $V(x)$, scale densities by a constant and normalize with Δx.

7 Variational Quantum Time Evolution

Variational algorithms provide a resource-efficient alternative to explicit Trotterization for simulating real-time quantum dynamics. Instead of decomposing the evolution operator into small time steps of unitary gates, the system's state is approximated by a parametrized ansatz $|\psi(\boldsymbol{\theta})\rangle$, where the parameters $\boldsymbol{\theta}$ evolve according to the McLachlan variational principle.

This principle minimizes the distance between the true time-evolved state and the variational manifold, leading to the Euler equation of motion,

$$M_{ij}\,\dot{\theta}_j = V_i, \tag{7.1}$$

with

$$M_{ij} = \mathrm{Re}\left[\langle\partial_i\psi|\partial_j\psi\rangle\right], \qquad V_i = \mathrm{Im}\left[\langle\partial_i\psi|H|\psi\rangle\right]. \tag{7.2}$$

Here, M is the quantum geometric tensor (or overlap matrix), and V encodes the system's response to the Hamiltonian. These quantities are estimated on a quantum device by preparing and measuring appropriately constructed circuits.

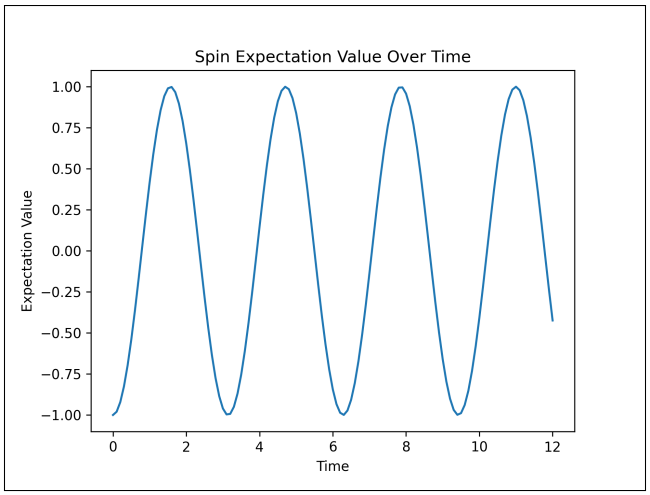

Figure 15: Workflow of the Variational Quantum Time Evolution (VQTE) algorithm. A parametrized circuit generates trial states; the measured observables are used to compute the matrices M and V, which feed into a classical solver updating the parameters $\boldsymbol{\theta}$ to follow the quantum dynamics. Figure adapted from Ref.[23]

7.1 Implementation in QFlux

QFlux automates the variational real-time evolution procedure through its modular `VarQRTE` driver. The driver constructs the variational ansatz, performs the required quantum measurements (`Measure_A` and `Measure_C`), and updates the parameters iteratively. The update rule follows an explicit Euler step,

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - (M^{-1}V)\,\Delta t, \tag{7.3}$$

where M^{-1} is computed using a truncated singular-value decomposition (SVD) to stabilize the inversion. The `VarQRTE` routine returns the full trajectory of parameters over the specified time interval, allowing reconstruction of observables such as expectation values or state fidelities.

A reference implementation of this algorithm is provided in Script S.2.44, using QFlux to simulate the real-time evolution of a two-qubit system described in Sec. 7.2. The simulation applies the McLachlan variational principle through the `VarQRTE` function, demonstrating

how variational techniques can efficiently capture quantum dynamics within shallow, noise-resilient circuits (Figure 16).

The ansatz circuit is defined using the `Construct_Ansatz` function, defined by a generator $G = \sum_j \theta_j G_j$, such that $|\psi(\theta)\rangle = \mathrm{e}^{-\mathrm{i}G(\theta)}|0\rangle$. This modular design enables rapid adaptation to different Hamiltonians and system sizes while maintaining low circuit depth and physical interpretability. The derivatives of the quantum state with respect to the variational parameters are linked to the generators G_i of the ansatz:

$$\frac{\partial |\psi(\theta(t))\rangle}{\partial \theta_i} = -\mathrm{i}G_i |\psi(\theta(t))\rangle . \tag{7.4}$$

7.1.1 Example: Single-Qubit Z-Rotation

As a simple illustration, consider the case of a single-qubit Z-rotation:

$$\frac{\partial}{\partial \theta} \mathrm{e}^{-\mathrm{i}\theta\sigma_Z} |\psi\rangle = -\mathrm{i}\,\sigma_Z \mathrm{e}^{-\mathrm{i}\theta\sigma_Z} |\psi\rangle . \tag{7.5}$$

This expression shows that taking the derivative of a parameterized unitary operation with respect to its angle introduces the corresponding generator-in this case, the Pauli operator σ_Z-multiplied by $-\mathrm{i}$. In general, for any parameterized gate of the form $U(\theta) = \mathrm{e}^{-\mathrm{i}\theta G}$, the generator G naturally appears when differentiating with respect to θ.

Therefore, the expectation values defined in Eqs. (7.2) can be evaluated in two main ways (Section 3.6):

1. by directly measuring the relevant Pauli operators on the quantum state, or

2. by performing Hadamard tests on the corresponding generators of the ansatz circuit.

Example: Evaluating matrix elements A_{ij}. To determine the elements of the matrix A_{ij}, one must evaluate the expectation values involving both the Hermitian conjugate of the generator associated with parameter θ_i, denoted $G_i^\dagger$, and the generator corresponding to

θ_j, denoted G_j. These quantities can be estimated using a single ancilla qubit within the Hadamard test framework.

The procedure is as follows:

1. Initialize the ancilla qubit in the $|0\rangle$ state.

2. To measure $G_i^\dagger$, apply a pair of NOT (X) gates to the ancilla-one before and one after the controlled application of G_i. This effectively inverts the control, allowing access to the Hermitian conjugate.

3. Perform a standard Hadamard test on the same ancilla qubit to measure G_j.

By combining these measurements, both $G_i^\dagger$ and G_j can be evaluated consistently using the same ancilla, providing the quantities required for constructing the matrix A_{ij}.

This approach generalizes naturally to multi-parameter variational circuits, where each generator G_k corresponds to a controllable rotation or interaction term in the ansatz.

7.2 Heisenberg Spin Chain

Spin systems provide compact yet nontrivial examples for testing quantum time-evolution algorithms. The Heisenberg Hamiltonian for a two-spin system is

$$\mathcal{H} = J(X_1X_2 + Y_1Y_2 + Z_1Z_2), \tag{7.6}$$

where J determines the interaction strength between neighboring spins. Starting from an initial superposition or product state, real-time evolution under this Hamiltonian generates oscillatory spin correlations, observable through quantities such as

$$\langle Z_i(t)\rangle = \langle\psi(t)|Z_i|\psi(t)\rangle. \tag{7.7}$$

When implemented with the `VarQRTE` driver, the algorithm iteratively measures the M

and V matrices for this two-qubit system and updates the circuit parameters to reproduce the spin dynamics. The resulting trajectories of $\langle Z_1(t)\rangle$ and $\langle Z_2(t)\rangle$ match closely with classical reference solutions, confirming the ability of the variational approach to capture coherent spin-exchange processes even with shallow quantum circuits.

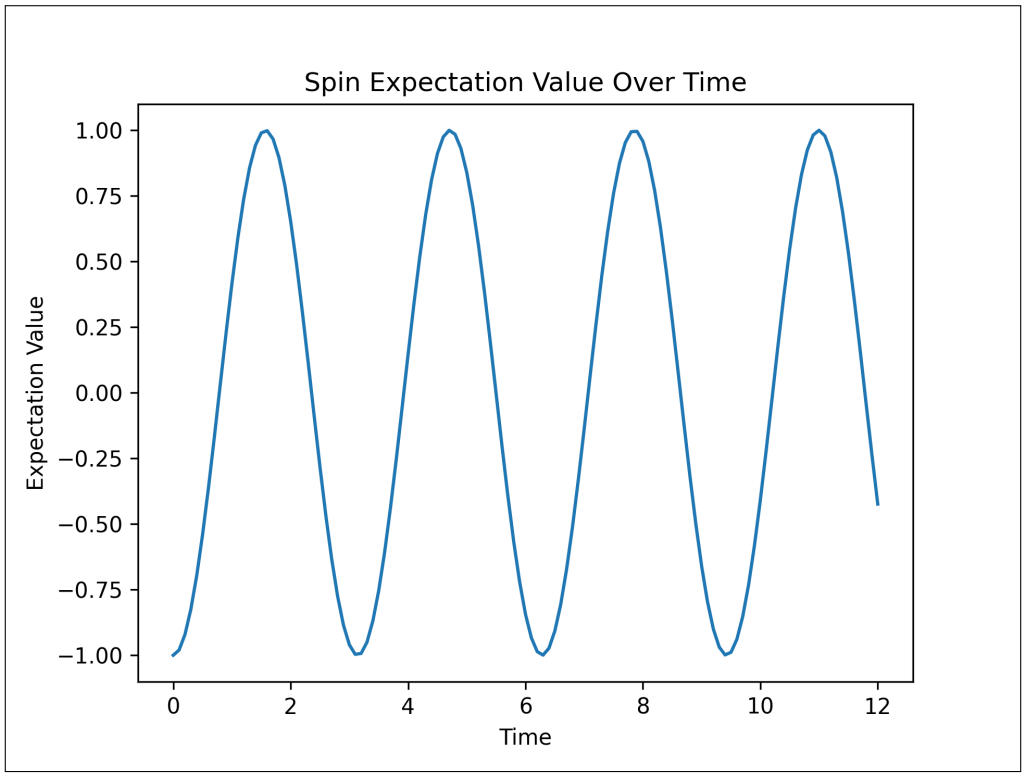

Figure 16: VarQRTE simulation of a two-spin Heisenberg model. The measured spin polarization $\langle Z_i(t)\rangle$ follows the expected sinusoidal pattern of coherent quantum evolution demonstrating the capabilities of QFlux's variational circuits for modeling interacting quantum systems.

Implementation Overview. The QFlux implementation of VarQRTE follows a modular design where each stage of the McLachlan update is separated into code components that measure and propagate variational parameters. The three central routines are:

- Utility functions that apply parameters and measure circuit derivatives in a given basis (`apply_param`, `measure_der`, `pauli_measure`),
- Construction of A_{ij} and C_i measurement circuits (`A_Circuit`, `C_Circuit`), and

- The main variational update routines (`VarQRTE` and `VarQITE`).

This separation opens the black box of the variational loop and allows users to inspect, modify, or extend the low-level circuit structure governing the variational principle. The code snippets detail this structure (Script S.2.45–Script S.2.48). The first defines `apply_param`, `measure_der` and `pauli_measure` functions, the foundational routines that apply parameters and probe their generators in the ansatz (Script S.2.45). Next, the construction of the A_{ij} and C_i circuits is modularized in `A_Circuit` and `C_Circuit` functions, each coupled with corresponding measurement wrappers `Measure_A` and `Measure_C` (Script S.2.46, Script S.2.47). Finally, the top-level `VarQRTE` these lower-level routines are integrated to evolve parameters in time (Script S.2.48).

7.3 Imaginary-Time Evolution

Analogous to the problem of dynamics is the problem of finding eigenstates of a Hamiltonian, which can also be approached with the Variational Quantum Imaginary-Time Evolution (VarQITE). This algorithm follows a protocol analogous to VarQRTE and can be invoked similarly within QFlux.

An initial state, which has a non-zero overlap with the ground state, can be evolved in imaginary time to reach the ground state. During the real time evolution, the eigenstates accumulate a phase $|\psi(t)\rangle = \mathrm{e}^{-\mathrm{i}Et}\,|\psi(t=0)\rangle$, while the evolution in imaginary time $\tau = \mathrm{i}t$ suppresses states with higher energies, $|\psi(\tau)\rangle = \mathrm{e}^{-E\tau}\,|\psi(\tau=0)\rangle$.

The imaginary time evolution can be performed very similarly to VarQRTE described above, except with one change in calculating the elements of C_i,

$$C_i = -\,\mathrm{Re}\left[\frac{\partial\,\langle\psi(\theta(t))|}{\partial\theta_i}\,\mathcal{H}\,|\psi(\theta(t))\rangle\right]. \tag{7.8}$$

Building upon the same framework as VarQRTE, one can measure the A_{ij} and C_i matrices on a quantum computer, and use them to change the parameters $\theta(t+dt) = \theta(t) + \dot{\theta}dt$.

Within QFlux, `VarQITE` function reuses the same modular components while invoking `Measure_C` function with the argument `evolution_type='imaginary'`. This structural parallel highlights how both real- and imaginary-time algorithms are implemented through identical circuit primitives.

In Script S.2.49, we demonstrate the module-based QFlux implementation of the VarQITE algorithm for a representative example of a model Hamiltonian for 3 interacting spins defined as

$$\mathcal{H} = 0.65\, Z_1 + Z_2 + Z_3 + 0.75(X_1X_2 + Y_1Y_2) + X_2X_3 + Y_2Y_3 \tag{7.9}$$

Thus, we can find the ground state for this spin-chain (Figure 17), which has the same parameters as outlined in Table 1.

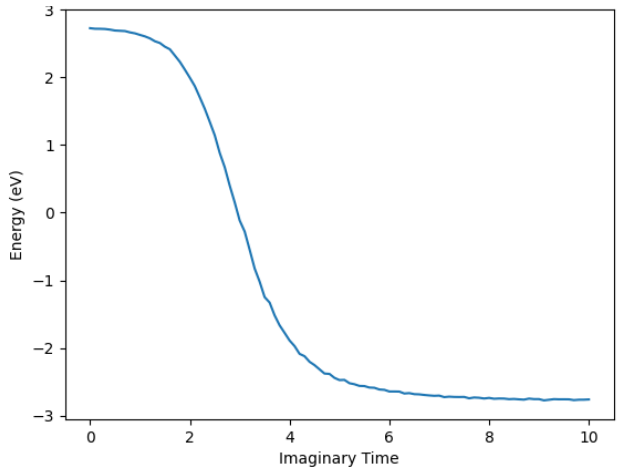

Figure 17: VarQITE drives the system toward the ground state as imaginary time increases, provided the ansatz is expressive enough. This enables estimation of the ground-state energy by sampling the long-time behavior.

7.4 Variational Quantum Eigensolver

An alternative approach to the QITE method is the Variational Quantum Eigensolver (VQE) algorithm, which is also a hybrid quantum-classical algorithm designed to approximate the ground state energy of a quantum system.[24,25] It is particularly well-suited for noisy intermediate-scale quantum (NISQ) devices since it limits the use of the quantum computer

to generation of ansatz for measurements of Pauli operators, while the classical computer adjusts the parameters of the ansatz to minimize the energy (Fig. 18).

VQE is grounded in the variational principle, which guarantees that for any normalized trial state $|\psi(\theta)\rangle$, the expectation value of the Hamiltonian $\mathcal{H}$ is an upper bound of the true ground state energy E_0:

$$E_0 \leq \frac{\langle\psi(\theta)|\,\mathcal{H}\,|\psi(\theta)\rangle}{\langle\psi(\theta)|\psi(\theta)\rangle} \tag{7.10}$$

The algorithm proceeds by preparing a parameterized quantum state $|\psi(\theta)\rangle$ using an ansatz circuit. The quantum computer measures the expectation value of the Pauli terms in the system Hamiltonian, and a classical computer updates the parameters θ using an optimizer (e.g., COBYLA or SPSA) to iteratively lower the estimated energy.

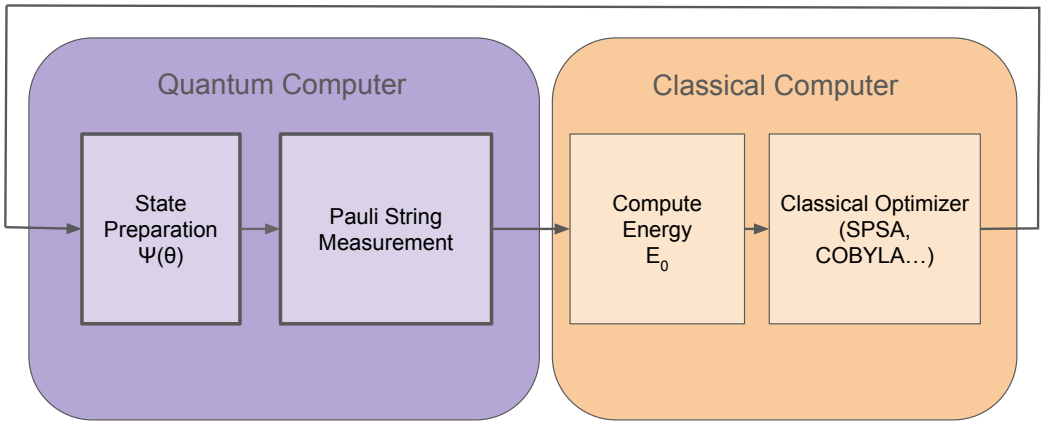

Figure 18: Overview of the Variational Quantum Eigensolver (VQE) workflow.

Upon convergence to optimal parameters θ_f, the quantum state $|\psi(\theta_f)\rangle$ approximates the ground state $|\psi_0\rangle$, and its energy $\langle\mathcal{H}\rangle_{\theta_f}$ approximates the ground state energy E_0.

A critical component of VQE is the choice of ansatz, which defines the expressivity and optimization landscape. Physically motivated ansätze that respect symmetries of the

Hamiltonian (e.g., particle number, spin parity) can reduce the parameter space and improve convergence. For example, selecting an ansatz that conserves excitation number for fermionic systems ensures the search remains within a physically relevant subspace.

The implementation in Script S.2.50 demonstrates VQE applied to the two-qubit Hamiltonian:

$$\mathcal{H} = \frac{1}{2}Z_1 + \frac{1}{2}Z_2 + \frac{1}{5}X_1X_2, \tag{7.11}$$

where Z_i and X_i are the Pauli operators for the i-th qubit. The script utilizes the Qiskit function `SparsePauliOp` to define the Hamiltonian, `EfficientSU2` as the ansatz, and the `Estimator` primitive for calculation of the expectation value. The classical optimizer used is COBYLA from the SciPy library.

8 Conclusions

This tutorial has demonstrated how quantum computers can be used to simulate real-time molecular and material dynamics by constructing executable circuits that mirror the fundamental operations of quantum mechanics. Building on the theoretical foundations established in **Part I**, we advanced from basic concepts-qubits, gates, and circuits-to full algorithms for quantum time evolution implemented in the open-source **QFlux** framework. Through this progression, we established a unified workflow that connects Hamiltonian encoding, circuit construction, time propagation, and measurement within a single reproducible environment.

Two complementary methods for quantum dynamical simulation were developed and benchmarked. The first, the *Quantum Split-Operator Fourier Transform* (**Q-SOFT**) algorithm, provides an explicit and physically transparent approach to propagating wavefunctions on qubit registers. By alternating potential and kinetic phase operators in conjugate bases connected by the Quantum Fourier Transform, Q-SOFT reproduces the structure and accuracy of the classical SOFT method while remaining compatible with quantum hardware. Its successful application to proton transfer in a DNA base-pair potential illustrates

that meaningful molecular dynamics can already be modeled using modest circuit depths on contemporary quantum devices.

The second approach, the *variational quantum time-evolution* method based on the McLachlan variational principle, offers a resource-efficient alternative to explicit propagation. Implemented in QFlux through the `VarQRTE` driver, it enables real-time evolution within low-depth, parameterized circuits that are robust to noise and well suited to near-term processors. Its accurate description of coherent spin exchange in the two-spin Heisenberg model highlights the versatility of variational strategies for capturing correlated quantum motion.

Together, these techniques establish a coherent bridge between classical simulation methods and their quantum counterparts, demonstrating that qubit-based hardware can now perform nontrivial dynamical simulations with quantitative fidelity. The QFlux framework provides both a pedagogical entry point and a flexible research platform for extending these capabilities to larger and more complex systems.

Looking ahead, **Part III** of the QFlux tutorial series will introduce general-purpose algorithms for quantum state initialization and arbitrary unitary decomposition. These developments will complete the foundation for constructing fully programmable quantum simulations, enabling users to prepare custom initial states, implement complex operators, and explore a broader class of quantum dynamical phenomena across chemistry, physics, and materials science.

Supporting Information

Detailed Jupyter notebooks implementing Q-SOFT and VQTE algorithms, code for Hamiltonian decomposition, and benchmark data are available in the Supporting Information and corresponding Google Colab notebook as well as through the QFlux Documentation site.

Acknowledgements

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) Nielsen, M. A.; Chuang, I. L. *Quantum computation and quantum information*; American Mathematical Society, 2010.

(2) 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.

(3) 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.

(4) 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.

(5) 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).

(6) Aleksandrowicz, G.; Alexander, T.; Barkoutsos, P.; Bello, L.; Ben-Haim, Y.; Bucher, D.; Cabrera-Hernández, F. J.; Carballo-Franquis, J.; Chen, A.; Chen, C.-F.; Chow, J. M.; Córcoles-Gonzales, A. D.; Cross, A. J.; Cross, A.; Cruz-Benito, J.; Culver, C.; González, S. D. L. P.; Torre, E. D. L.; Ding, D.; Dumitrescu, E.; Duran, I.; Eendebak, P.; Everitt, M.; Sertage, I. F.; Frisch, A.; Fuhrer, A.; Gambetta, J.; Gago, B. G.; Gomez-Mosquera, J.; Greenberg, D.; Hamamura, I.; Havlicek, V.; Hellmers, J.; Łukasz Herok; Horii, H.; Hu, S.; Imamichi, T.; Itoko, T.; Javadi-Abhari, A.; Kanazawa, N.; Karazeev, A.; Krsulich, K.; Liu, P.; Luh, Y.; Maeng, Y.; Marques, M.; Martín-Fernández, F. J.; McClure, D. T.; McKay, D.; Meesala, S.; Mezzacapo, A.; Moll, N.; Rodríguez, D. M.; Nannicini, G.; Nation, P.; Ollitrault, P.; O'Riordan, L. J.; Paik, H.; Pérez, J.; Phan, A.; Pistoia, M.; Prutyanov, V.; Reuter, M.; Rice, J.; Davila, A. R.; Rudy, R. H. P.; Ryu, M.; Sathaye, N.; Schnabel, C.; Schoute, E.; Setia, K.; Shi, Y.; Silva, A.; Siraichi, Y.; Sivarajah, S.; Smolin, J. A.; Soeken, M.; Takahashi, H.; Tavernelli, I.; Taylor, C.; Taylour, P.; Trabing, K.; Treinish, M.; Turner, W.; Vogt-Lee, D.; Vuillot, C.; Wildstrom, J. A.; Wilson, J.; Winston, E.; Wood, C.; Wood, S.; Wörner, S.; Akhalwaya, I. Y.; Zoufal, C. Qiskit: An Open-source Framework for Quantum Computing. 2019.

(7) 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.

(8) 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.

(9) Stavenger, T. J.; Crane, E.; Smith, K. C.; Kang, C. T.; Girvin, S. M.; Wiebe, N. C2qa-bosonic qiskit. 2022 IEEE High Performance Extreme Computing Conference (HPEC). 2022; pp 1–8.

(10) Killoran, N.; Izaac, J.; Quesada, N.; Bergholm, V.; Amy, M.; Weedbrook, C. Strawberry Fields: A Software Platform for Photonic Quantum Computing. *Quantum* **2019**, *3*, 129.

(11) Bisong, E. *Building machine learning and deep learning models on google cloud platform: a comprehensive guide for beginners*; Springer, 2019; pp 59–64.

(12) Cabral, D. Simulation of the Heisenberg Model Dynamics Tutorial. https://www.youtube.com/watch?v=fm4bH9B0Ikw, year = 2024, note = (accessed: 2024-03-05).

(13) Wang, T.; Sanz, S.; Castro-Esteban, J.; Lawrence, J.; Berdonces-Layunta, A.; Mohammed, M. S. G.; Vilas-Varela, M.; Corso, M.; Peña, D.; Frederiksen, T.; de Oteyza, D. G. Magnetic Interactions Between Radical Pairs in Chiral Graphene Nanoribbons. *Nano Letters* **2022**, *22*, 164–171, PMID: 34936370.

(14) Fiori, E. R.; Pastawski, H. M. Non-Markovian decay beyond the Fermi Golden Rule: Survival collapse of the polarization in spin chains. *Chemical Physics Letters* **2006**, *420*, 35–41.

(15) 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.

(16) 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.

(17) 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.

(18) 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.

(19) Coppersmith, D.; Feig, E.; Linzer, E. Hadamard transforms on multiply/add architectures. *IEEE Transactions on Signal Processing* **1994**, *42*, 969–970.

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

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

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

(23) Kyaw, T. H.; Soley, M. B.; Allen, B.; Bergold, P.; Sun, C.; Batista, V. S.; Aspuru-Guzik, A. Boosting quantum amplitude exponentially in variational quantum algorithms. *Quantum Science and Technology* **2023**, *9*, 01LT01.

(24) Peruzzo, A.; McClean, J.; Shadbolt, P.; Yung, M.-H.; Zhou, X.-Q.; Love, P. J.; Aspuru-Guzik, A.; O'Brien, J. L. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications* **2014**, *5*, 4213.

(25) Tilly, J.; Chen, H.; Cao, S.; Picozzi, D.; Setia, K.; Li, Y.; Grant, E.; Wossnig, L.; Rungger, I.; Booth, G. H.; Tennyson, J. The Variational Quantum Eigensolver: A review of methods and best practices. *Physics Reports* **2022**, *986*, 1–128.

Supporting Information for

QFlux: Quantum Circuit Implementations of Molecular Dynamics.

Part II - Closed Quantum Systems

Delmar G. A. Cabral[†], Brandon C. Allen[†], Cameron Cianci[‡], Alexander V. Soudackov[†], Xiaohan Dan[†], Nam P. Vu[†], 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 Pauli Average and the Hadamard Test Equivalency

This section collects the derivations establishing the equivalence between Pauli-string averaging and the Hadamard test applied to $e^{i\pi P/2}$, and extends the discussion to general Hermitian observables via Pauli decompositions and smooth-angle variants.

S.1.1 Pauli strings P with $P^2 = \mathbb{I}$

We first consider Pauli strings P that are Hermitian, unitary, and square to the identity. For such operators, the exponential $e^{i\pi P/2}$ reduces to a simple multiple of P, which leads to a direct identification between Pauli averages and the imaginary-part output of the Hadamard test.

Let P be a Pauli string (hence Hermitian and unitary) so that $P^2 = \mathbb{I}$ and the spectrum is $\{\pm 1\}$. Then

$$e^{i\frac{\pi}{2}P} \;=\; \cos\!\left(\frac{\pi}{2}\right)\mathbb{I} + i\sin\!\left(\frac{\pi}{2}\right)P \;=\; iP.$$

For any state $|\psi\rangle$,

$$\langle\psi|e^{i\frac{\pi}{2}P}|\psi\rangle = i\,\langle\psi|P|\psi\rangle.$$

Applying the Hadamard test with $U = e^{i\pi P/2}$ therefore yields

$$\mathrm{Re}\Big[\langle\psi|U|\psi\rangle\Big] = 0, \qquad \mathrm{Im}\Big[\langle\psi|U|\psi\rangle\Big] = \langle\psi|P|\psi\rangle.$$

Hence, for Pauli strings, the imaginary-part Hadamard test with $U = e^{i\pi P/2}$ returns exactly the Pauli expectation value $\langle P\rangle$.

On the other hand, direct Pauli-string measurement proceeds by rotating each local factor of P to Z, measuring in the computational basis, mapping outcomes to eigenvalues ± 1, and averaging over shots:

$$\langle P\rangle = \frac{1}{N}\sum_{s=1}^{N} p^{(s)}, \qquad p^{(s)} \in \{\pm 1\}.$$

Thus, for any Pauli string P, *Pauli averaging* and the *Hadamard test on* $\mathrm{e}^{\mathrm{i}\pi P/2}$ are operationally different but mathematically equivalent estimators of the same real number $\langle P\rangle$.

S.1.2 General Hermitian observables via Pauli decomposition

Next, we extend the equivalence to general Hermitian observables by expressing them as real linear combinations of Pauli strings and using linearity of expectation values.

Let a Hermitian observable O be expanded as a linear combination of Pauli strings,

$$O = \sum_j c_j P_j, \qquad P_j^2 = \mathbb{I}, \quad c_j \in \mathbb{R}.$$

Linearity of expectation values gives

$$\langle O\rangle = \sum_j c_j \langle P_j\rangle.$$

Each $\langle P_j\rangle$ can be obtained either by (i) direct Pauli averaging (basis rotations to Z, measure, and average) or by (ii) the Hadamard test on $U_j = \mathrm{e}^{\mathrm{i}\pi P_j/2} = \mathrm{i}P_j$, extracting the imaginary part:

$$\langle P_j\rangle = \mathrm{Im}\Big[\langle\psi|\mathrm{e}^{\mathrm{i}\frac{\pi}{2}P_j}|\psi\rangle\Big].$$

Therefore

$$\langle O\rangle \;=\; \sum_j c_j \;\mathrm{Im}\Big[\langle\psi|\mathrm{e}^{\mathrm{i}\frac{\pi}{2}P_j}|\psi\rangle\Big] \;=\; \sum_j c_j\langle P_j\rangle$$

establishing the equivalence between (a) averaging measurement outcomes of Pauli strings and (b) applying the Hadamard test to $\mathrm{e}^{\mathrm{i}\pi P_j/2}$ term-by-term.

S.1.3 Optional: smooth-angle variant for general Hermitian O

Finally, we comment on a smooth-angle (derivative-based) variant that, in principle, allows extraction of $\langle O\rangle$ from small-angle unitaries $\mathrm{e}^{\mathrm{i}\theta O}$, even when O does not square to the identity.

For a general Hermitian O (not necessarily unitary), define

$$f(\theta) \;=\; \langle\psi|\mathrm{e}^{\mathrm{i}\theta O}|\psi\rangle.$$

Then $f'(0) = \mathrm{i}\langle O\rangle$, so

$$\langle O\rangle \;=\; \lim_{\theta\to 0}\frac{\mathrm{Im}[f(\theta)]}{\theta}.$$

Thus, in principle, $\langle O\rangle$ can be extracted from Hadamard tests of $\mathrm{e}^{\mathrm{i}\theta O}$ at small θ (via a derivative/parameter-shift evaluation). In practice, the decomposition $O = \sum_j c_j P_j$ with term-wise estimation (Section 3.6.1, "Expectation Values from Pauli-String Measurements") is preferred on contemporary hardware, while exact single-shot extraction from $\mathrm{e}^{\mathrm{i}\pi O/2}$ holds *exactly* whenever $O^2 = \mathbb{I}$ (e.g., Pauli strings).

S.2 Tutorial Scripts

This section organizes the code listings as short, task-oriented subsections. Each snippet is self-contained and tied to the narrative in the main text (e.g., Figs. 1 and 2) and now includes a brief explanation of its purpose and usage.

S.2.1 Installing Qiskit and Importing Packages

This script sets up the Python environment in a Jupyter/Colab notebook, installs `qflux`, and imports Qiskit, Aer, and other utilities used throughout the quantum-circuit examples.

Script S.2.1 shows a minimal Colab setup.

Script S.2.1: Installing Qiskit and Importing Packages

```
!pip install qflux

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import axes
```

```python
import scipy.linalg as LA

from qiskit.circuit.library import QFT
from qiskit_aer import Aer
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit.quantum_info.operators import Operator
from qiskit_ibm_runtime import QiskitRuntimeService, Options, SamplerV2
```

S.2.2 Bell State: Circuit and Simulation

The next script constructs a two-qubit Bell circuit, runs it on the Aer simulator, and plots the measurement histogram, reproducing the Bell state example in Figs. 1 and 2.

Script S.2.2 implements the Bell circuit from Fig. 1 and produces the histogram in Fig. 2.

Script S.2.2: Quantum Circuit of a Bell State

```python
from qiskit_aer import Aer
from qiskit import QuantumCircuit, transpile
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt
from IPython.display import display

qc = QuantumCircuit(2, 2)
qc.h(0)
qc.cx(0, 1)
qc.measure([0, 1], [0, 1])
display(qc.draw('mpl'))

simulator = Aer.get_backend('aer_simulator')
compiled_circuit = transpile(qc, simulator)
result = simulator.run(compiled_circuit).result()
counts = result.get_counts()
plot_histogram(counts)
```

S.2.3 Spin-1/2 Heisenberg Model

This group of scripts builds and propagates a two-site Heisenberg spin chain, constructs the corresponding quantum circuit, and demonstrates how to run it on simulators and (mock) IBM hardware.

S.2.4 Circuit for $e^{-it\,Z\otimes\cdots\otimes Z}$

This helper constructs a Qiskit circuit implementing the unitary $e^{-it\,Z\otimes\cdots\otimes Z}$ (optionally controlled by an ancilla) on a subset of qubits, using a chain of CNOTs and a single R_z or CR_z rotation.

Script S.2.3: Quantum Circuit for $e^{-itZ\otimes\cdots\otimes Z}$

```
from qiskit import QuantumCircuit, QuantumRegister

def exp_all_z(circuit, quantum_register, pauli_indexes, control_qubit=None, t=1):
    if control_qubit and control_qubit.register not in circuit.qregs:
        circuit.add_register(control_qubit.register)
    if not pauli_indexes:
        if control_qubit:
            circuit.p(t, control_qubit)
        return circuit
    for i in range(len(pauli_indexes) - 1):
        circuit.cx(quantum_register[pauli_indexes[i]],
     quantum_register[pauli_indexes[i + 1]])
    target = quantum_register[pauli_indexes[-1]]
    angle = -2 * t
    if control_qubit:
        circuit.crz(angle, control_qubit, target)
    else:
        circuit.rz(angle, target)
    for i in reversed(range(len(pauli_indexes) - 1)):
        circuit.cx(quantum_register[pauli_indexes[i]],
     quantum_register[pauli_indexes[i + 1]])
    return circuit
```

S.2.5 Circuit for e^{-itP}

This function `exp_pauli` builds a circuit for e^{-itP} where P is a general Pauli string, by conjugating to a multi-Z operator, calling `exp_all_z`, and then undoing the basis change.

Script S.2.4: Quantum Circuit for e^{-itP}

```python
import numpy as np
from qiskit import QuantumCircuit, QuantumRegister

def exp_pauli(pauli, quantum_register, control_qubit=None, t=1):
    if len(pauli) != len(quantum_register):
        raise ValueError("Pauli string length must match register size.")
    pauli_indexes = []
    pre_circuit = QuantumCircuit(quantum_register)
    for i, op in enumerate(pauli):
        if op == 'I':
            continue
        elif op == 'X':
            pre_circuit.h(i); pauli_indexes.append(i)
        elif op == 'Y':
            pre_circuit.rx(np.pi/2, i); pauli_indexes.append(i)
        elif op == 'Z':
            pauli_indexes.append(i)
        else:
            raise ValueError(f"Invalid Pauli operator '{op}' at position {i}.")
    circuit = QuantumCircuit(quantum_register)
    circuit.compose(pre_circuit, inplace=True)
    circuit = exp_all_z(circuit, quantum_register, pauli_indexes, control_qubit, t)
    circuit.compose(pre_circuit.inverse(), inplace=True)
    return circuit
```

S.2.6 First-Order Trotterization of e^{-iHt}

This routine builds a first-order Trotterized circuit for e^{-iHt} given a dictionary of Pauli strings and coefficients, optionally with a control qubit and multiple Trotter steps.

Script S.2.5: Quantum Circuit for e^{-iHt}

```
from qiskit import QuantumCircuit, QuantumRegister

def hamiltonian_simulation(hamiltonian, quantum_register=None, control_qubit=None,
    t=1, trotter_number=1):
   if not hamiltonian:
      raise ValueError("Hamiltonian must contain at least one term.")
   n_qubits = len(next(iter(hamiltonian)))
   if quantum_register is None:
      quantum_register = QuantumRegister(n_qubits)
   delta_t = t / trotter_number
   circuit = QuantumCircuit(quantum_register)
   for pauli_str, coeff in hamiltonian.items():
      term_circuit = exp_pauli(pauli_str, quantum_register, control_qubit, coeff *
    delta_t)
      circuit.compose(term_circuit, inplace=True)
   full_circuit = QuantumCircuit(quantum_register)
   for _ in range(trotter_number):
      full_circuit.compose(circuit, inplace=True)
   return full_circuit
```

S.2.7 Test: Hamiltonian Simulation

This test script applies the Hamiltonian simulation routine to a simple two-qubit Hamiltonian, prepares an initial superposition, performs the time evolution, and measures the output distribution on a simulator.

Script S.2.6: Test Code for Hamiltonian Simulation

```
from qiskit_aer import Aer
from qiskit import QuantumCircuit, transpile
from qiskit.visualization import plot_histogram
qr=QuantumRegister(2)
qc = QuantumCircuit(qr)
hamiltonian = {"ZZ": 0.5, "YY": 0.3}
t = np.pi / 4
trotter_steps = 1
U = hamiltonian_simulation(hamiltonian, quantum_register=qr, t=t,
    trotter_number=trotter_steps)
qc.h(qr)
qc.append(U,qr)
qc.measure_all()
```

```
sim = Aer.get_backend("aer_simulator")
qobj = transpile(qc, sim)
result = sim.run(qobj).result()
counts = result.get_counts()
print("Measurement counts:", counts)
qc.decompose().draw()
```

S.2.7.1 2-site Hamiltonian and Propagator

The first cell defines Pauli matrices, constructs the 2-site Heisenberg Hamiltonian, and computes the corresponding unitary propagator $U = \mathrm{e}^{-\mathrm{i}H}$ via a matrix exponential.

Script S.2.7: 2-site Heisenberg spin chain Hamiltonian and propagator

```
J = 1
h0 = -0.5
h1 = 0.5
X = np.array([[0,1],[1,0]], dtype = complex)
Y = np.array([[0,1j],[-1j,0]], dtype = complex)
Z = np.array([[1,0],[0,-1]], dtype = complex)
I = np.eye(2, dtype = complex)
H = 0.5*(h0*np.kron(Z, I) + h1*np.kron(I, Z)) + J/4*(np.kron(X, X) + np.kron(Y, Y) +
    np.kron(Z, Z))
U = LA.expm(-1j * H)
```

The next snippet applies the propagator to an initial basis state, performing a purely classical propagation of the two-spin system.

Script S.2.8: Classical propagation of 2-site Heisenberg chain

```
psi_init = np.array([1,0,0,0],dtype = complex)
psi_fin = U @ psi_init
```

S.2.7.2 Quantum Circuit Construction and Measurement

The following cell initializes the quantum circuit that will implement the two-site Heisenberg time evolution on a quantum backend (or simulator).

Script S.2.9: Quantum circuit initialization

```
qreg = QuantumRegister(2)
creg = ClassicalRegister(2, 'creg')
entangler = QuantumCircuit(qreg, creg)
# Qiskit initializes qubits in |00> by default
```

Here, the precomputed unitary U is wrapped as a Qiskit `Operator` and appended to the circuit as a custom gate acting on the two qubits.

Script S.2.10: Quantum circuit unitary propagation

```
U_gate = Operator(U)
entangler.append(U_gate, [0,1])
```

The next cell adds measurements on both qubits, preparing the circuit for sampling on a simulator or real device.

Script S.2.11: Quantum circuit measurement

```
entangler.measure(0,0)
entangler.measure(1,1)
```

S.2.7.3 IBM Runtime Setup and Execution

This script configures access to IBM Quantum services via an API token and defines a convenience function `run_IBM_session` that transpiles, runs, and collects results for a given circuit and backend.

Script S.2.12: Accessing IBM account with API token

```python
from qiskit_ibm_runtime import QiskitRuntimeService

MY_API_TOKEN = "INSERT_YOUR_API_TOKEN_HERE"

service = QiskitRuntimeService(channel="local",
                        token=MY_API_TOKEN)

from qiskit_ibm_runtime import SamplerV2 as Sampler
from qiskit_ibm_runtime import Session
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager

def run_IBM_session(circuit, backend, nshots=2048, opt_level=1):
    # Transpilation to device gate-set/architecture
    pm = generate_preset_pass_manager(backend=backend,
                              optimization_level=opt_level)
    transpiled_circuit = pm.run(circuit)

    # Circuit execution
    with Session(backend=backend) as session:
        sampler = Sampler(mode=session)
        sampler.options.default_shots = nshots
        job = sampler.run([transpiled_circuit])

    # Result retrieval
    print(f"Job ID is {job.job_id()}")
    pub_result = job.result()[0]
    result_dict = pub_result.data.creg.get_counts()

    return result_dict
```

The next cell demonstrates how to execute the entangler circuit on a QASM simulator using the IBM Runtime helper defined above, returning the measurement statistics.

Script S.2.13: Execute Quantum Circuit on QASM simulator

```python
from qiskit_aer import AerSimulator
result_dict = run_IBM_session(entangler, backend=AerSimulator())
print("Counts per state:", result_dict)
```

The final two-site example shows how to run the same circuit on a noisy backend (a fake device model) to emulate realistic hardware noise.

Script S.2.14: Execute Quantum Circuit on Quantum Hardware

```python
from qiskit_ibm_runtime.fake_provider import FakeManilaV2
noisy_backend = FakeManilaV2()
noisy_result_dict = run_IBM_session(entangler, backend=noisy_backend)
print("Counts per state (noisy backend):", noisy_result_dict)
```

S.2.7.4 QFlux Spin-Chain Simulation (Statevector)

This script uses the `SpinDynamicsS` class from `qflux` to simulate a three-site Heisenberg spin chain using Trotterized statevector propagation and to save and plot the resulting dynamics.

Script S.2.15: QFlux Simulation for Spin Chain using Statevector

```python
from qflux.closed_systems.spin_dynamics_oo import SpinDynamicsS

num_q = 3
evolution_timestep = 0.1
n_trotter_steps = 1
hamiltonian_coefficients = [[0.75 / 2, 0.75 / 2, 0.0, 0.65]] + [
    [0.5, 0.5, 0.0, 1.0] for _ in range(num_q - 1)
]
initial_state = "011"  # Specify the initial state as a binary string

csimulation = SpinDynamicsS(
    num_q,
    evolution_timestep,
```

```
    n_trotter_steps,
    hamiltonian_coefficients,
)
csimulation.run_dynamics(nsteps=250, state_string=initial_state)
csimulation.save_results(f"{num_q}_spin_chain")
csimulation.plot_results(f"{num_q}_spin_chain_statevector")
```

S.2.7.5 Heisenberg Hamiltonian Assembly

The next helper builds the local Hamiltonian terms for a given bond in an N-site Heisenberg chain, using Qiskit's `SparsePauliOp` representation.

Script S.2.16: Heisenberg Hamiltonian for site n

```
from qiskit.quantum_info import SparsePauliOp

def get_hamiltonian_n_site_terms(n, coeff, n_qubits):
    '''
        Assemble each term in the Hamiltonian using its Pauli-string
        representation and multiply by the corresponding coefficient.
        coeff = [Jxx, Jyy, Jzz, Oz]
    '''
    XX_coeff, YY_coeff, ZZ_coeff, Z_coeff = coeff

    XX_term = SparsePauliOp("I"*n + "XX" + "I"*(n_qubits - 2 - n)) * XX_coeff
    YY_term = SparsePauliOp("I"*n + "YY" + "I"*(n_qubits - 2 - n)) * YY_coeff
    ZZ_term = SparsePauliOp("I"*n + "ZZ" + "I"*(n_qubits - 2 - n)) * ZZ_coeff
    Z_term  = SparsePauliOp("I"*n + "Z"  + "I"*(n_qubits - 1 - n)) * Z_coeff

    return XX_term + YY_term + ZZ_term + Z_term
```

Using the per-site builder, the following function assembles the full even and odd components of an N-site Heisenberg Hamiltonian, returning them as separate `SparsePauliOp` objects for use in Trotter decompositions.

Script S.2.17: Heisenberg Hamiltonian for N Sites

```python
def get_heisenberg_hamiltonian(n_qubits, coeff=None):
    '''
    Constructs the Heisenberg Hamiltonian for an N-site spin chain.

    H = \sum _i ^N h_z Z_i
        + \sum _i ^{N-1} (h_xx X_iX_{i+1}
            + h_yy Y_iY_{i+1}
            + h_zz Z_iZ_{i+1}
            )

    Parameters:
        n_qubits (int): Number of spins/qubits.
        coeff (list of lists, optional): A list of sublists containing the coefficients
            [XX, YY, ZZ, Z] for each site. The last sublist contains only the Z
     component.
            Defaults to uniform coefficients if not provided.

    Returns:
        list: Two components of the Hamiltonian (even and odd terms).
    '''

    # Three qubits because for 2 we get H_O = 0
    assert n_qubits >= 3

    if coeff == None:
        'Setting default values for the coefficients'
        coeff = [[1.0, 1.0, 1.0, 1.0] for i in range(n_qubits)]

    # Even terms of the Hamiltonian
    # (summing over individual pair-wise elements)
    H_E = sum((get_hamiltonian_n_site_terms(i, coeff[i], n_qubits)
            for i in range(0, n_qubits-1, 2)))

    # Odd terms of the Hamiltonian
    # (summing over individual pair-wise elements)
    H_O = sum((get_hamiltonian_n_site_terms(i, coeff[i], n_qubits)
            for i in range(1, n_qubits-1, 2)))

    # adding final Z term at the Nth site
    final_term = SparsePauliOp("I" * (n_qubits - 1) + "Z")
    final_term *= coeff[n_qubits-1][3]
    if (n_qubits % 2) == 0:
        H_E += final_term
    else:
        H_O += final_term

    # Returns the list of the two sets of terms
    return [H_E, H_O]
```

The following diagnostic cell constructs a 3-site Heisenberg Hamiltonian using the helper above and prints its even/odd components, as well as the combined Hamiltonian.

Script S.2.18: Heisenberg Hamiltonian 3 Sites

```
num_q = 3
# XX YY ZZ, Z
ham_coeffs = ([[0.75/2, 0.75/2, 0.0, 0.65]]+
          [[0.5, 0.5, 0.0, 1.0] for _ in range(num_q-1)])

spin_chain_hamiltonian = get_heisenberg_hamiltonian(num_q, ham_coeffs)

print('Hamiltonian (even and odd components):',spin_chain_hamiltonian)
print('Combined Hamiltonian:', sum(spin_chain_hamiltonian))
```

S.2.7.6 Trotterized Time Evolution

This script builds the Trotterized time-evolution operator $e^{-iH\tau}$ using Qiskit's `PauliEvolutionGate` with a Suzuki–Trotter synthesis strategy.

Script S.2.19: Trotterized Time Evolution Operator

```
from qiskit.circuit.library import PauliEvolutionGate
from qiskit.synthesis import SuzukiTrotter
from qiskit import QuantumCircuit, QuantumRegister
import numpy as np
from itertools import groupby
import re

def get_time_evolution_operator(num_qubits, tau, trotter_steps, coeff=None):
    '''
    Generates the Trotterized time-evolution operator for a Heisenberg spin chain

        Inputs:
            num_qubits (int): number of qubits, which should be equal to the
                number of spins in the chain
            evo_time (float): time parameter in time-evolution operator
            trotter_steps (int): number of time steps for the Suzuki-Trotter
                decomposition
```

```
        coeff (list of lists): parameters for each term in the Hamiltonian
          for each site ie ([[XX0, YY0, ZZ0, Z0], [XX1, YY1, ZZ1, Z1], ...])
      Returns:
        evo_op.definition: Trotterized time-evolution operator
   '''
   # Heisenberg_hamiltonian = [H_E, H_O]
   heisenberg_hamiltonian = get_heisenberg_hamiltonian(num_qubits, coeff)

   # e^(-i*H*evo_time), with Trotter decomposition
   # exp[(i*evo_time)*(IIIIXXIIII + IIIIYYIIII + IIIIZZIIII + IIIIZIIIII)]
   evo_op = PauliEvolutionGate(heisenberg_hamiltonian, tau,
                        synthesis=SuzukiTrotter(order=2,
                        reps=trotter_steps))
   return evo_op.definition

num_shots = 100
num_q = 3
evolution_timestep = 0.1
n_trotter_steps = 1
# XX YY ZZ, Z
ham_coeffs = ([[0.75/2, 0.75/2, 0.0, 0.65]]
            + [[0.5, 0.5, 0.0, 1.0]
            for i in range(num_q-1)])
time_evo_op = get_time_evolution_operator(
   num_qubits=num_q, tau=evolution_timestep,
   trotter_steps=n_trotter_steps, coeff=ham_coeffs)
```

S.2.7.7 Manual Circuit Patterns for Pauli Exponentials

The next helper generates the circuit pattern implementing $\exp(-\mathrm{i}\,\Delta t\, h\, \sigma_\alpha)$ for single-qubit Pauli terms, using appropriate rotation gates.

Script S.2.20: Circuit for Exponential of 1-Qubit Pauli Term

```
def generate_circ_pattern_1qubit(circ, term, delta_t):
   coeff = 2 * term[1] * delta_t
   if term[3] == 'X':
      circ.rx(coeff, term[2])
   elif term[3] == 'Y':
      circ.ry(coeff, term[2])
   elif term[3] == 'Z':
      circ.rz(coeff, term[2])
```

```
    return circ
```

The following helper identifies the positions of Pauli operators in a string label and organizes Hamiltonian terms by interaction order and qubit index, which is useful for structured Trotter orderings.

Script S.2.21: Sorting Terms by Interaction order

```
def find_string_pattern(pattern, string):
    match_list = []
    for m in re.finditer(pattern, string):
        match_list.append(m.start())
    return match_list

def sort_Pauli_by_symmetry(ham):
    # Separates a qiskit PauliOp object terms into 1 and 2-qubit
    # operators. Furthermore, 2-qubit operators are separated according
    # to the parity of the index first non-identity operation.
    one_qubit_terms = []
    two_qubit_terms = []
    # separating the one-qubit from two-qubit terms
    for term in ham:
        matches = find_string_pattern('X|Y|Z', str(term.paulis[0]))
        pauli_string = term.paulis[0]
        coeff = np.real(term.coeffs[0])
        str_tag = pauli_string.to_label().replace('I', '')
        if len(matches) == 2:
            two_qubit_terms.append((pauli_string, coeff, matches, str_tag))
        elif len(matches) == 1:
            one_qubit_terms.append((pauli_string, coeff, matches, str_tag))

    # sorting the two-qubit terms according to index on which they act
    two_qubit_terms = sorted(two_qubit_terms, key=lambda x: x[2])
    # separating the even from the odd two-qubit terms
    even_two_qubit_terms = list(filter(lambda x: not x[2][0]%2, two_qubit_terms))
    odd_two_qubit_terms = list(filter(lambda x: x[2][0]%2, two_qubit_terms))

    even_two_qubit_terms = [list(v) for i, v in groupby(even_two_qubit_terms, lambda
     x: x[2][0])]
    odd_two_qubit_terms = [list(v) for i, v in groupby(odd_two_qubit_terms, lambda
     x: x[2][0])]

    return one_qubit_terms, even_two_qubit_terms, odd_two_qubit_terms
```

The following function constructs the circuit block implementing the exponential of a two-qubit Pauli interaction, parameterized by effective angles for XX, YY, and ZZ contributions.

Script S.2.22: Circuit for Exponential of 2-Qubit Pauli Term

```python
def generate_circ_pattern_2qubit(circ, term, delta_t):

    # wires to which to apply the operation
    wires = term[0][2]

    # angles to parameterize the circuit,
    # based on exponential argument
    if any('XX' in sublist for sublist in term):
        g_phi = ( 2 * (-1) * term[0][1] * delta_t - np.pi / 2)
    else:
        g_phi = - np.pi / 2
    if any('YY' in sublist for sublist in term):
        g_lambda = (np.pi/2 - 2 * (-1) * term[1][1] * delta_t)
    else:
        g_lambda = np.pi/2
    if any('ZZ' in sublist for sublist in term):
        g_theta = (np.pi/2 - 2 * (-1) * term[2][1] * delta_t)
    else:
        g_theta = np.pi/2

    # circuit
    circ.rz(-np.pi/2, wires[1])
    circ.cx(wires[1], wires[0])
    circ.rz(g_theta, wires[0])
    circ.ry(g_phi, wires[1])
    circ.cx(wires[0], wires[1])
    circ.ry(g_lambda, wires[1])
    circ.cx(wires[1], wires[0])
    circ.rz(np.pi/2, wires[0])
    return circ
```

S.2.7.8 Manual Trotterization of the Propagator

This function builds a full Trotter step circuit from the sorted Pauli terms, supporting both basic and symmetric (Strang-like) orderings and optional repetition of the Trotter step.

Script S.2.23: Manual Trotterization of Propagator

```
def get_manual_Trotter(num_q, pauli_ops, timestep, n_trotter=1,
                       trotter_type='basic', reverse_bits=True):
    # sorts the Pauli strings according to qubit number they affect and symmetry
    one_q, even_two_q, odd_two_q = sort_Pauli_by_symmetry(pauli_ops)
    # scales the timestep according to the number of trotter steps
    timestep_even_two_q = timestep / n_trotter
    timestep_odd_two_q = timestep / n_trotter
    timestep_one_q = timestep / n_trotter
    # symmetric places 1/2 of one_q and odd_two_q before and after even_two_q
    if trotter_type == 'symmetric':
        timestep_odd_two_q /= 2
        timestep_one_q /= 2
    # constructs circuits for each segment of the operators
    qc_odd_two_q, qc_even_two_q, qc_one_q = QuantumCircuit(num_q),
     QuantumCircuit(num_q), QuantumCircuit(num_q)
    for i in even_two_q:
        qc_even_two_q = generate_circ_pattern_2qubit(qc_even_two_q, i,
     timestep_even_two_q)
    for i in odd_two_q:
        qc_odd_two_q = generate_circ_pattern_2qubit(qc_odd_two_q, i,
     timestep_odd_two_q)
    for i in one_q:
        qc_one_q = generate_circ_pattern_1qubit(qc_one_q, i, timestep_one_q)
    # assembles the circuit for Trotter decomposition of exponential
    qr = QuantumRegister(num_q)
    qc = QuantumCircuit(qr)
    if trotter_type == 'basic':
        qc = qc.compose(qc_even_two_q)
        qc = qc.compose(qc_odd_two_q)
        qc = qc.compose(qc_one_q)
    elif trotter_type == 'symmetric':
        qc = qc.compose(qc_one_q)
        qc = qc.compose(qc_odd_two_q)
        qc = qc.compose(qc_even_two_q)
        qc = qc.compose(qc_odd_two_q)
        qc = qc.compose(qc_one_q)
    # repeats the single_trotter circuit several times to match n_trotter
    for i in range(n_trotter-1):
        qc = qc.compose(qc)
    if reverse_bits:
        return qc.reverse_bits()
    else:
        return qc
```

The next cell prints example manual Trotter circuits for different numbers of steps and for basic vs symmetric Trotter schemes, illustrating the circuit depth trade-offs.

Script S.2.24: Manual Trotter Circuits

```
spin_chain_hamiltonian = get_heisenberg_hamiltonian(num_q, ham_coeffs)

spin_chain_hamiltonian = sum(spin_chain_hamiltonian)
print(get_manual_Trotter(num_q, spin_chain_hamiltonian, 0.1).draw())
print(get_manual_Trotter(num_q, spin_chain_hamiltonian, 0.1, n_trotter=2).draw())
print(get_manual_Trotter(num_q, spin_chain_hamiltonian, 0.1,
    trotter_type='symmetric').draw())
print(get_manual_Trotter(num_q, spin_chain_hamiltonian, 0.1, n_trotter=2,
    trotter_type='symmetric').draw())
```

S.2.7.9 Circuit Initialization and State Preparation

This script sets up a generic N-qubit circuit with matching quantum and classical registers, which will be used for spin-chain dynamics.

Script S.2.25: Quantum Circuit Initialization

```
from qiskit import QuantumCircuit
from qiskit import QuantumRegister, ClassicalRegister
from qiskit import transpile

# specifying a quantum register with specific number of qubits
qr = QuantumRegister(num_q)
# classical register used for measurement of qubits
cr = ClassicalRegister(num_q)
# quantum circuit combining quantum and classical registers
qc = QuantumCircuit(qr, cr) # instantiated here
qc.draw(style='iqp')
print(qc)
```

The next cell prepares a specific spin configuration (one spin-up and the rest spin-down) by applying X gates selectively, then verifies the initialized state on a statevector simulator.

Script S.2.26: Quantum Circuit for Vacuum State Initialization

```
# specifying initial state by flipping qubit states
for qubit_idx in range(num_q):
    if qubit_idx == 0:
        # generate only one spin-up at first qubit
        qc.id(qubit_idx)
    else:
        # make all other spins have the spin-down state
        qc.x(qubit_idx)
qc.barrier()
qc.draw(style='iqp')
print(qc)

# checking the initial state
device = Aer.get_backend('statevector_simulator')
qc_init_state = execute(qc, backend=device).result()
qc_init_state = qc_init_state.get_statevector()
print(qc_init_state)
```

This helper uses Qiskit's `initialize` instruction to encode a computational basis state (here, "011") as an amplitude-encoded initial state and appends it to the main circuit.

Script S.2.27: State Initialization: Amplitude Encoding

```
qr_init = QuantumRegister(num_q)
qc_init = QuantumCircuit(qr_init)
qc_init.initialize('011')
qc.append(qc_init, qc.qubits)
```

The next block appends the time-evolution operator to the prepared circuit, shows the circuit depth before and after transpilation, and illustrates the structure of the Trotterized dynamics circuit.

Script S.2.28: Applying Time Evolution Operator to Circuit

```
# generating the time evolution operator for a specific set of
# hamiltonian parameters and timestep
time_evo_op = get_time_evolution_operator(num_qubits=num_q,
      tau=evolution_timestep,
      trotter_steps=n_trotter_steps,
      coeff=ham_coeffs)

# appending the Hamiltonian evolution to the circuit
qc.append(time_evo_op, list(range(num_q)))
qc.barrier()
qc.draw(style='iqp')
print(qc)

# Depth check
print('Depth of the circuit is', qc.depth())
# transpiled circuit to statevector simulator
qct = transpile(qc, device, optimization_level=2)
qct.decompose().decompose()
qct.draw(style='iqp')
print(qct)

print('Depth of the circuit after transpilation is '
      f'{qct.depth()}')
```

S.2.7.10 Statevector-Based Quantum Simulation Helpers

This helper function `qsolve_statevector` encapsulates a single time step of statevector propagation by appending a given evolution circuit to the current state and returning the updated statevector.

Script S.2.29: Execution of Quantum Experiment

```
import numpy as np
from qiskit import QuantumCircuit, QuantumRegister
from qiskit_aer import Aer

def qsolve_statevector(psin, qc):
   r'''
      Performs iterative quantum state propagation using a statevector simulator.
```

```
    The initial state is the statevector from the prior iteration:

      | \psi _t \rangle  = e^{i*\tau*H/hbar} e^{i*\tau*H/hbar} ... | \psi _0 \rangle
      -> | \psi _t \rangle  = e^{i*\tau*H/hbar} | \psi _{t-\tau} \rangle


      Args:
         psin (array): Initial quantum state.
         qc (QuantumCircuit): Circuit representing the time evolution operator.

      Returns:
         psin (statevector): final statevector after execution
   '''
   # Determining number of qubits from the length of the state vector
   n=np.size(psin)
   num_qubits=int(np.log2(np.size(psin)))
   # Circuit preparation
   qreg = QuantumRegister(num_qubits)
   circ = QuantumCircuit(qreg)

   circ.initialize(psin,qreg)
   circ.barrier()
   circ.append(qc, qreg)
   circ.barrier()

   # Circuit execution
   device = Aer.get_backend('statevector_simulator')
   psin = execute(circ, backend=device).result()
   return psin.get_statevector()
```

The next script performs a full survival-amplitude experiment: it initializes a basis state, repeatedly applies the time-evolution operator, records overlaps with the initial state, and stores and plots the resulting survival amplitude vs time.

Script S.2.30: Statevector Experiment

```
# Qubit basis states
zero_state = np.array([[1],[0]])
one_state = np.array([[0],[1]])

# Prepare an initial state (e.g., |011>), as follows
psin = zero_state # for the first spin
# iterates over the remaining spins, by performing
```

```
# Kronecker Product
for i in range(num_q-1):
   psin = np.kron(psin, one_state)
psin0 = psin.flatten()
print(psin0)

# time evolution operator
time_evo_op = get_time_evolution_operator(num_qubits=num_q,
      tau=evolution_timestep,
      trotter_steps=n_trotter_steps,
      coeff=ham_coeffs)
# number of steps for which to propagate
# (totaling 25 units of time)
nsteps = 250
psin_list = []
psin_list.append(psin0)
correlation_list = []

# Perform propagation by statevector re-initialization
for k in range(nsteps):
   #print(f'Running dynamics step {k}')
   if k > 0:
      psin = qsolve_statevector(psin_list[-1], time_evo_op)
      # removes the last initial state to save memory
      psin_list.pop()
      # stores the new initial state
      psin_list.append(psin)
   correlation_list.append(np.vdot(psin_list[-1],psin0))

time = np.arange(0, evolution_timestep*(nsteps),
            evolution_timestep)
np.save(f'{num_q}_spin_chain_time', time)
sa_observable = np.abs(correlation_list)
np.save(f'{num_q}_spin_chain_SA_obs', sa_observable)

# Plot survival amplitude
plt.plot(time, sa_observable, '-o')
plt.xlabel('Time')
plt.ylabel('Absolute Value of Survival Amplitude, '
        r'$\left|\langle \psi | \psi \rangle \right|$')
plt.xlim((min(time), max(time)))
plt.yscale('log')
plt.legend()
plt.show()
```

S.2.7.11 QFlux Spin-Chain Simulation (Hadamard Test)

This script uses the `SpinDynamicsH` class from `qflux` to run a Hadamard-test-based simulation of the same spin chain, saving and plotting observables extracted from ancilla measurements.

Script S.2.31: QFlux Simulation for Spin Chain using Hadamard Test

```
from qflux.closed_systems.spin_dynamics_oo import SpinDynamicsH

num_q = 3
evolution_timestep = 0.1
n_trotter_steps = 1
hamiltonian_coefficients = [[0.75 / 2, 0.75 / 2, 0.0, 0.65]] + [
                    [0.5, 0.5, 0.0, 1.0] for _ in range(num_q - 1)
                    ]
initial_state = "011" # Specify the initial state as a binary string

qsimulation = SpinDynamicsH(
                num_q,
                evolution_timestep,
                n_trotter_steps,
                hamiltonian_coefficients,
                )
qsimulation.run_simulation(state_string=initial_state, total_time=25,
    num_shots=100)
qsimulation.save_results('hadamard_test')
qsimulation.plot_results('hadamard_test')
```

S.2.7.12 Explicit Hadamard Test Construction and Analysis

The following function constructs a Hadamard-test circuit on `num_q` system qubits and one ancilla, optionally including a phase gate on the ancilla to extract imaginary parts of expectation values.

Script S.2.32: Hadamard Test Function

```python
import numpy as np
from qiskit_aer import Aer
from qiskit import QuantumCircuit
from qiskit import QuantumRegister, ClassicalRegister

def get_hadamard_test(num_q, initial_state, control_operation,
                      control_repeats=0, imag_expectation=False):

    # Create circuit with quantum and classical registers
    qr_hadamard = QuantumRegister(num_q+1)
    cr_hadamard = ClassicalRegister(1)
    qc_hadamard = QuantumCircuit(qr_hadamard, cr_hadamard) # instantiated here

    # Initialize the computation qubits
    qc_hadamard.append(initial_state, qr_hadamard[1:]) # initial psi
    qc_hadamard.barrier()

    # Hadamard test on the ancilla qubit
    qc_hadamard.h(0)
    if imag_expectation:
        qc_hadamard.p(-np.pi/2, 0) # qc_hadamard.s(0).inverse() may be equivalent

    # iterates over the number of times the control operation should be added
    for i in range(control_repeats):
        qc_hadamard.append(control_operation, qr_hadamard[:])
    qc_hadamard.h(0)
    qc_hadamard.barrier()

    # Measuring the ancilla
    qc_hadamard.measure(0,0)

    return qc_hadamard
```

This helper converts single-bit measurement counts into an average spin value $\langle \sigma_z \rangle$, mapping classical outcomes $\{0, 1\} \mapsto \{+1, -1\}$ and averaging over shots.

Script S.2.33: Hadamard Test Post-Processing

```python
def get_spin_correlation(counts):
    qubit_to_spin_map = {
        '0': 1,
        '1': -1,
```

```
    }
    total_counts = 0
    values_list = []
    for k,v in counts.items():
        values_list.append(qubit_to_spin_map[k] * v)
        total_counts += v
    # print(values_list)
    average_spin = (sum(values_list)) / total_counts
    return average_spin
```

The following script sets up the Hadamard-test dynamics: it defines the Hamiltonian, builds the controlled time-evolution operator, initializes the system, and prepares for the time loop.

Script S.2.34: Hadamard Test Propagation

```
# IMPORTANT: Use qasm_simulator to obtain meaningful statistics
simulator = Aer.get_backend('qasm_simulator')

num_q = 3
n_trotter_steps = 1
# XX YY ZZ, Z
hamiltonian_coefficients = ([[0.75/2, 0.75/2, 0.0, 0.65]]
                      + [[0.5, 0.5, 0.0, 1.0]
                         for i in range(num_q-1)])

num_shots = 100 # increase to check for convergence

evolution_timestep = 0.1
total_time = 25
time_range = np.arange(0, total_time+evolution_timestep,
                  evolution_timestep)

# time evolution operator
time_evo_op = get_time_evolution_operator(num_qubits=num_q,
      tau=evolution_timestep,
      trotter_steps=n_trotter_steps,
      coeff=hamiltonian_coefficients)

controlled_time_evo_op = time_evo_op.control()
print(controlled_time_evo_op.decompose())

init_state_list = '1' + '0' * (num_q-1)
```

```
init_circ = get_initialization(num_q, init_state_list)
init_circ.draw(style='iqp')
print(init_circ)
```

Finally, this loop runs the Hadamard test over a range of times to reconstruct the complex survival amplitude, compares it to the statevector benchmark, and plots the absolute value vs time.

Script S.2.35: Hadamard Test Execution

```
# it takes >1hr for 3 spins, with the parameters defined above
# lists t store observables
real_amp_list = []
imag_amp_list = []
for idx,time in enumerate(time_range):
   print(f'Running dynamics step {idx}')
   # Real component ------------------------------
   qc_had_real = get_hadamard_test(num_q, init_circ,
                           controlled_time_evo_op,
                           control_repeats=idx,
                           imag_expectation=False)
   had_real_counts = get_circuit_execution_counts(
         qc_had_real, simulator, n_shots=num_shots)
   real_amplitude = get_spin_correlation(had_real_counts)
   real_amp_list.append(real_amplitude)

   # Imag component ------------------------------
   qc_had_imag = get_hadamard_test(num_q, init_circ,
                           controlled_time_evo_op,
                           control_repeats=idx,
                           imag_expectation=True)
   had_imag_counts = get_circuit_execution_counts(
         qc_had_imag, simulator, n_shots=num_shots)
   imag_amplitude = get_spin_correlation(had_imag_counts)
   imag_amp_list.append(imag_amplitude)
   print(f'Finished step {idx}, where '
         f'Re = {real_amplitude:.3f} '
         f'Im = {imag_amplitude:.3f}')

   real_amp_array = np.array(real_amp_list)
   imag_amp_array = np.array(imag_amp_list)

np_abs_correlation_with_hadamard_test = np.abs(real_amp_array + 1j*imag_amp_array)
```

```
# plotting the data
plt.plot(time_range, np_abs_correlation_with_hadamard_test,
         '.', label='Hadamard Test')

sa_statevector = np.load(f'data/Part_I_SpinChain/{num_q}_spin_chain_SA_obs.npy')
time = np.load(f'{num_q}_spin_chain_time.npy')
plt.plot(time, sa_statevector, '-', label='Statevector')

plt.xlabel('Time')
plt.ylabel('Absolute Value of Survival Amplitude')
plt.legend()
plt.show()
```

S.2.8 1D Potential for A–T Tautomerization

This script defines a one-dimensional double-well potential model for A–T tautomerization, along with its second derivative, in atomic units. These functions serve as the potential energy surface in subsequent SOFT/QSOFT simulations.

Script S.2.36: 1D PES for A-T tautomerization

```
from qflux.closed_systems.utils import convert_fs_to_au, convert_eV_to_au,
    convert_au_to_fs
ev2au = convert_eV_to_au(1.0)

def get_doublewell_potential(x, x0=1.9592, f=ev2au, a0=0.0, a1=0.429, a2=-1.126,
    a3=-0.143, a4=0.563):
   xi = x/x0
   return f*(a0 + a1*xi + a2*xi**2 + a3*xi**3 + a4*xi**4)

def get_doublewell_potential_second_deriv(x, x0=1.9592, f=ev2au, a0=0.0, a1=0.429,
    a2=-1.126, a3=-0.143, a4=0.563):
   return f*(2*a2/x0**2 + 6*a3*x/x0**3 + 12*a4*x**2/x0**4)
```

S.2.9 QFlux Simulation Using QSOFT

This cell sets up and runs a QSOFT-based QFlux simulation of A–T tautomerization using a discretized coordinate grid, automatically initializing operators, states, and propagation

parameters.

Script S.2.37: QFlux Simulation using QSOFT

```
from qflux.closed_systems import QubitDynamicsCS
from qflux.closed_systems.utils import get_proton_mass
from qiskit_aer import Aer

proton_mass = get_proton_mass()
x0     = 1.9592
N_steps = 3000

omega = np.sqrt(get_doublewell_potential_second_deriv(x0)/proton_mass)
AT_dyn_obj  = QubitDynamicsCS(n_basis=64, xo=1.5*x0, mass=proton_mass, omega=omega)

AT_dyn_obj.set_coordinate_operators(x_min=-4.0, x_max=4.0)
AT_dyn_obj.initialize_operators()
AT_dyn_obj.set_initial_state(wfn_omega=omega)

total_time = 30.0 * convert_fs_to_au(1.0)
AT_dyn_obj.set_propagation_time(total_time, N_steps)
AT_dyn_obj.set_hamiltonian(potential_type='quartic')

AT_dyn_obj.propagate_SOFT()
AT_dyn_obj.propagate_qt()

backend = Aer.get_backend('statevector_simulator')
AT_dyn_obj.propagate_qSOFT(backend=backend)
```

S.2.10 Gaussian Initial Wavepacket

This script builds a Gaussian (coherent-state) initial wavepacket for the double-well potential using the same grid conventions as the SOFT helper functions.

Script S.2.38: Gaussian initial wavepacket

```
mass_proton = 1836.15
x0 = 1.9592
x_0 = 1.5*x0
p_0 = 0.0
xmin, xmax = -4., 4.
Nq = 6
N_xpts = 2**Nq
```

```
xgrid = get_xgrid(xmin, xmax, N_xpts)
omega = np.sqrt(get_doublewell_potential_second_deriv(x0)/proton_mass)

psi_0 = get_coherent_state(xgrid, p_0, x_0, proton_mass, omega)
```

S.2.11 Split-Operator Propagators (V and K)

This cell constructs diagonal potential and kinetic energy propagators for the double-well system, sets up the time grid, and defines the effective shifted potential for use in SOFT and QSOFT propagation.

Script S.2.39: Preparation of potential and kinetic split propagators

```
from qflux.closed_systems.utils import convert_fs_to_au, convert_au_to_fs

au2fs = convert_au_to_fs(1.0)
fs2au = convert_fs_to_au(1.0)

pgrid = get_pgrid(xmin, xmax, N_xpts, reorder=True)
dx = xgrid[1] - xgrid[0]
dp = pgrid[1] - pgrid[0]

Vx_DW = get_doublewell_potential(xgrid)
VV = Vx_DW - get_doublewell_potential(x0) - omega/2

tmin, tmax = 0.0, 30.0*fs2au
iterations = 3000
mass = mass_proton

tgrid = np.linspace(tmin, tmax, iterations)
time_step = tgrid[1] - tgrid[0]

VVd_prop = np.diag(np.exp(-1j*Vx_DW/2*time_step))
KEd_prop = np.diag(np.exp(-1j*pgrid**2/2/mass*time_step))
```

S.2.12 Quantum SOFT Circuit Construction

This script builds the QSOFT quantum circuit for the double-well dynamics, initializing the wavepacket and then iteratively applying the split-operator sequence with QFT and inverse

QFT at each time step.

Script S.2.40: Quantum SOFT Circuit Preparation

```
import scipy.linalg as LA
from qiskit.circuit.library import QFT
from qiskit_aer import Aer
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit.quantum_info.operators import Operator
from qiskit_ibm_runtime import QiskitRuntimeService, Options, SamplerV2

nqubits = Nq
q_reg = QuantumRegister(nqubits)
c_reg = ClassicalRegister(nqubits)
qc = QuantumCircuit(q_reg)

qc.initialize(psi_0, q_reg[:],normalize=True)

for k in range(iterations):
   V_op = Operator(VVd_prop)
   qc.append(V_op, q_reg)
   qc.append(QFT(nqubits,do_swaps=True,inverse=False),q_reg)
   K_op = Operator(KEd_prop)
   qc.append(K_op, q_reg)
   qc.append(QFT(nqubits,do_swaps=True,inverse=True),q_reg)
   qc.append(V_op, q_reg)
```

S.2.13 Quantum SOFT: Execution

This cell runs the QSOFT circuit on a statevector simulator and extracts the final wavefunction amplitudes for later comparison with the classical SOFT benchmark.

Script S.2.41: Quantum SOFT circuit execution

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import Aer
from qflux.closed_systems.custom_execute import execute

backend = Aer.get_backend('statevector_simulator')
executed_circuit = execute(qc, backend=backend, shots=1024)
psin = executed_circuit.result().get_statevector().data
```

S.2.14 Classical SOFT Benchmark

This benchmark script performs a standard SOFT propagation in coordinate and momentum space to generate a classical reference trajectory for the double-well wavepacket.

Script S.2.42: Classical SOFT Benchmark

```
V_prop = np.exp(-1j*Vx_DW/2*time_step)
K_prop = np.exp(-1j*pgrid**2/2/mass*time_step)

propagated_states = [psi_0]
psi_t = psi_0
print("For ",tmax*au2fs," fs using a timestep of ",time_step*au2fs," fs =
    ",time_step," a.u.")

for tstep_idx in range(len(tgrid)):
   psi_t = do_SOFT_propagation(psi_t, K_prop, V_prop)
   propagated_states.append(psi_t)

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

S.2.15 Plotting Wavefunctions and Potential

This final A–T tautomerization script compares the potential profile, initial wavepacket, classical SOFT final state, and QSOFT final state on a common plot, providing a visual benchmark between quantum and classical propagation.

Script S.2.43: Plotting initial and final wavefunctions

```
from scipy.interpolate import interp1d
def get_prob_density(psi):
   return np.real(np.conjugate(psi) * psi)

x_dense = np.linspace(xgrid[0], xgrid[-1], 512)
f_interp = interp1d(xgrid, get_prob_density(propagated_states[-1]), kind='cubic')
rho_interp = f_interp(x_dense)

fig, ax = plt.subplots()
ax.plot(xgrid, VV, '-',color='black',label='A-T pair potential')
```

```
ax.plot(xgrid,
    0.04*np.real(get_prob_density(psi_0)),'--',color='red',label='Initial coherent
    state')
ax.plot(x_dense, 0.04*rho_interp,'-',color='blue',label='(SOFT) State at t = 30
    fs',zorder=0,markeredgecolor='blue',fillstyle='full',markerfacecolor='white')
ax.plot(xgrid, 0.04*np.real(psin.conj()*psin/dx),'o',color='blue',label='(Qiskit)
    State at t = 30 fs', markevery=1, alpha=0.25)
ax.axhline(0, lw=0.5, color='black', alpha=1.0)
ax.axvline(-x0, lw=0.5, color='black', alpha=0.5)
ax.axvline(x0, lw=0.5, color='black', alpha=0.5)
ax.axvline(x0*1.5, lw=0.5, color='red', alpha=0.5)
ax.set_xlabel('x, Bohr',fontsize=14)
ax.set_ylabel('Energy, Hartrees',fontsize=14)
ax.tick_params(labelsize=12, grid_alpha=0.5)
plt.ylim(-0.03,0.07)
plt.legend(fontsize=12,loc='upper center')
plt.show()
```

S.2.16 VarQRTE Driver Example

This example uses the `VarQRTE` driver to perform variational real-time evolution of a single-qubit system under a simple Hamiltonian, measuring the spin expectation value as a function of time.

Script S.2.44: Variation Quantum Real Time Evolution

```
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit
from qiskit.quantum_info import SparsePauliOp
from qiskit_aer.primitives import EstimatorV2 as Estimator
from qflux.closed_systems.VarQTE import VarQRTE, Construct_Ansatz

H = SparsePauliOp.from_list([("X", 1.0)])
qc = QuantumCircuit(1)
qc.x(0)

layers = 1
total_time = 12
timestep = 0.1
params = VarQRTE(layers, H, total_time, timestep, init_circ=qc)


estimator = Estimator()
```

```
observable = SparsePauliOp.from_list([("Z", 1.0)])
spin_values = []

for i in range(len(params)):
   ansatz = Construct_Ansatz(qc, params[i], H.num_qubits)
   result = estimator.run([(ansatz, observable)]).result()
   spin_values.append(result[0].data.evs)

plt.title("Spin Expectation Value Over Time")
plt.plot([i*timestep for i in range(int(total_time/timestep)+1)], spin_values)
plt.xlabel("Time")
plt.ylabel("Expectation Value")
plt.show()
```

S.2.16.1 VQRTE Utilities

The following utilities implement parameter application and derivative measurement patterns used internally by the VarQTE routines to build the A and C matrices.

Script S.2.45: Utility Functions for Variation Quantum Time Evolution

```
import numpy as np
from qiskit import QuantumCircuit
from qiskit_aer.primitives import EstimatorV2 as Estimator
from qiskit.quantum_info import SparsePauliOp
import numpy.typing as npt

def apply_param(params: npt.NDArray[np.float64], i: int, qc: QuantumCircuit, N:
    int) -> None:
   qc.rx(params[i], i % N)
   if i % N == N - 1 and i != len(params) - 1:
      for i in range(N - 1):
         qc.cz(i, i + 1)

def measure_der(i: int, qc: QuantumCircuit, N: int) -> None:
   qc.cx(N, i % N)

def pauli_measure(qc: QuantumCircuit, pauli_string: str) -> None:
   N = len(pauli_string)
   for i in range(len(pauli_string)):
      if str(pauli_string[i]) == "X": qc.cx(N, i)
      if str(pauli_string[i]) == "Y": qc.cy(N, i)
```

```
        if str(pauli_string[i]) == "Z": qc.cz(N, i)
```

S.2.16.2 VQRTE: Construct and Measure A

These routines construct and measure the entries of the A matrix in the McLachlan variational principle, using an ancilla-based measurement scheme and Qiskit's `EstimatorV2` primitive.

Script S.2.46: Construction and Measurement of A Matrix

```
def A_Circuit(params: npt.NDArray[np.float64], i: int, j: int, N: int) ->
    QuantumCircuit:
   qc = QuantumCircuit(N + 1, 1)
   qc.h(N)
   for parameter in range(len(params)):
      if parameter == i:
         qc.x(N); measure_der(parameter, qc, N); qc.x(N)
      if parameter == j:
         measure_der(parameter, qc, N)
      apply_param(params, parameter, qc, N)
   qc.h(N)
   return qc

def Measure_A(init_circ: QuantumCircuit, params: npt.NDArray[np.float64], N: int,
    shots: int = 2**10, noisy: bool = False) -> npt.NDArray[np.float64]:
   A = [[0.0 for i in range(len(params))] for j in range(len(params))]
   for i in range(len(params)):
      for j in range(len(params) - i):
         qc = QuantumCircuit(N + 1, 1)
         ansatz = A_Circuit(params, i, i + j, N)
         qc = qc.compose(init_circ, [k for k in range(N)])
         qc = qc.compose(ansatz, [k for k in range(N + 1)])
         observable = SparsePauliOp.from_list([("Z" + "I" * N, 1.0)])
         if noisy:
            device_backend = FakeSherbrooke()
            noise_model = NoiseModel.from_backend(device_backend)
            estimator = Estimator(options={"backend_options":{"noise_model":
    noise_model},
                                           "run_options":{"shots": shots}})
         else:
            estimator = Estimator(options={"run_options":{"shots": shots}})
         result = estimator.run([(qc, observable)]).result()
         A[i][i + j] = result[0].data.evs
```

```
    return np.array(A)
```

S.2.16.3 VQRTE: Construct and Measure C

These routines build and measure the C vector associated with the Hamiltonian, again using ancilla-based circuits and Pauli measurements to obtain the contributions of each term in the Pauli expansion of H.

Script S.2.47: Construction and Measurement of C Matrix

```
def C_Circuit(params: npt.NDArray[np.float64], i: int, pauli_string: str, N: int,
    evolution_type: str = "real") -> QuantumCircuit:
   qc = QuantumCircuit(N + 1, 1)
   qc.h(N)
   if evolution_type == "imaginary":
      qc.s(N)
   else:
      qc.z(N)
   for parameter in range(len(params)):
      if parameter == i:
         qc.x(N); measure_der(parameter, qc, N); qc.x(N)
      apply_param(params, parameter, qc, N)
   pauli_measure(qc, pauli_string)
   qc.h(N)
   return qc

def Measure_C(init_circ: QuantumCircuit, params: npt.NDArray[np.float64], H:
    SparsePauliOp, N: int, shots: int = 2**10, evolution_type: str = "real", noisy:
    bool = False) -> npt.NDArray[np.float64]:
   C = [0.0 for i in range(len(params))]
   for i in range(len(params)):
      for pauli_string in range(len(H.paulis)):
         qc = QuantumCircuit(N + 1, 1)
         ansatz = C_Circuit(params, i, H.paulis[pauli_string], N,
    evolution_type=evolution_type)
         qc = qc.compose(init_circ, [k for k in range(N)])
         qc = qc.compose(ansatz, [k for k in range(N + 1)])
         observable = SparsePauliOp.from_list([("Z" + "I" * N, 1.0)])
         if noisy:
            device_backend = FakeSherbrooke()
            noise_model = NoiseModel.from_backend(device_backend)
            estimator = Estimator(options={"backend_options":{"noise_model":
    noise_model},
```

```
                                    "run_options":{"shots": shots}})
            else:
                estimator = Estimator(options={"run_options":{"shots": shots}})
            result = estimator.run([(qc, observable)]).result()
            C[i] -= 0.5 * H.coeffs[pauli_string].real * result[0].data.evs
    return np.array(C)
```

S.2.16.4 VQRTE Driver and Ansatz Construction

The final VarQRTE routines implement the full time-stepping procedure: at each time step, they measure A and C, compute the parameter updates via a pseudo-inverse, and build the corresponding ansatz circuit for subsequent estimation or visualization.

Script S.2.48: Variational Quantum Real-Time Evolution Driver

```
from typing import Optional, List

def VarQRTE(n_reps_ansatz: int, hamiltonian: SparsePauliOp, total_time: float =
    1.0, timestep: float = 0.1, init_circ: Optional[QuantumCircuit] = None, shots:
    int = 2**10, noisy: bool = False) -> List[npt.NDArray[np.float64]]:
    if init_circ is None:
        init_circ = QuantumCircuit(hamiltonian.num_qubits)
    initial_params = np.zeros(hamiltonian.num_qubits * (n_reps_ansatz + 1))
    num_timesteps = int(total_time / timestep)
    all_params = [np.copy(initial_params)]
    my_params = np.copy(initial_params)
    for i in range(num_timesteps):
        print(f"Simulating Time={str(timestep*(i+1))}                    ", end="\r")
        A = Measure_A(init_circ, my_params, hamiltonian.num_qubits, shots=shots,
     noisy=noisy)
        C = Measure_C(init_circ, my_params, hamiltonian, hamiltonian.num_qubits,
     shots=shots, evolution_type="real", noisy=noisy)
        u, s, v = np.linalg.svd(A)
        for j in range(len(s)):
            if s[j] < 1e-2:
                s[j] = 1e8
        A_inv = (v.T) @ np.diag(s**-1) @ (u.T)
        theta_dot = A_inv @ C
        my_params -= theta_dot * timestep
        all_params.append(np.copy(my_params))
    return all_params
```

```python
def Construct_Ansatz(init_circ: QuantumCircuit, params: npt.NDArray[np.float64],
    N: int) -> QuantumCircuit:
   qc = QuantumCircuit(N, 0)
   qc = qc.compose(init_circ, [k for k in range(N)])
   ansatz = QuantumCircuit(N, 0)
   for parameter in range(len(params)):
      apply_param(params, parameter, ansatz, N)
   qc = qc.compose(ansatz, [k for k in range(N)])
   return qc
```

S.2.17 Quantum Imaginary Time Evolution

This example applies the VarQITE driver to a three-qubit Hamiltonian, starting from a simple product state, and tracks the energy expectation value as a function of (imaginary) time, illustrating convergence toward the ground state.

Script S.2.49: Variation Quantum Imaginary Time Evolution

```python
import matplotlib.pyplot as plt
from qiskit import QuantumCircuit
from qiskit.quantum_info import SparsePauliOp

from qflux.closed_systems.VarQTE import VarQITE, ansatz_energy

# --- Define the Hamiltonian ---
H = SparsePauliOp.from_list([("IIZ", 1.0), ("IZI", 1.0), ("ZII", 0.65), ("IXX",
    1.0), ("IYY", 1.0), ("XXI", 0.75), ("YYI", 0.75)])

# --- Define the Initial State ---
qc = QuantumCircuit(3)
qc.rx(0.5, 0)
qc.rx(0.5, 1)
qc.rx(0.5, 2)

# --- Perform Variational Real-Time Evolution ---
layers = 0
total_time = 10
timestep = 0.1
params = VarQITE(layers, H, total_time, timestep, init_circ=qc)
# Params now holds the parameter values for the ansatz at each timestep for
    Imaginary-Time Evolution
```

```python
# --- Get the Expectation Value of the Energy ---
all_energies = []
for i in range(len(params)):
   print(f"Timestep {i} Energy: {ansatz_energy(qc, params[i], H)}")
   all_energies.append(ansatz_energy(qc, params[i], H)[0])

# --- Plot Expectation Values Over Time ---
plt.title("VarQITE Energy Over Imaginary Time")
plt.plot([i*timestep for i in range(int(total_time/timestep)+1)], all_energies)
plt.xlabel("Imaginary Time")
plt.ylabel("Energy (eV)")
plt.show()
```

S.2.18 Variational Quantum Eigensolver

This final script demonstrates a basic VQE workflow using Qiskit's `EfficientSU2` ansatz and `StatevectorEstimator` to find the ground-state energy of a simple two-qubit Hamiltonian.

Script S.2.50: Variational Quantum Eigensolver

```python
# -- Imports --
# - EfficientSU2: A parameterized quantum circuit (ansatz) often used in VQE.
# - SparsePauliOp: Efficient representation of Hamiltonians in terms of Pauli
    strings.
# - StatevectorEstimator: Estimates expectation values
import numpy as np
from scipy.optimize import minimize
from qiskit.circuit.library import EfficientSU2
from qiskit.primitives import StatevectorEstimator
from qiskit.quantum_info import SparsePauliOp

# --- Define the Hamiltonian ---
# H = 0.5 * Z_0 + 0.5 * Z_1 + 0.2 * X_0 * X_1
hamiltonian = SparsePauliOp.from_list([("ZI", 0.5), ("IZ", 0.5), ("XX", 0.2)])

# --- Initialize the Estimator Primitive ---
# StatevectorEstimator: Ideal, noiseless estimator
# using statevectors (no sampling noise).
# Computes expectation value exactly.
estimator = StatevectorEstimator()
```

```
# --- Define the Ansatz ---
# Use EfficientSU2 as a general-purpose parameterized ansatz.
# EfficientSU2 is expressive and hardware-efficient, using layers
# of single-qubit rotations and entangling gates.
ansatz = EfficientSU2(num_qubits=hamiltonian.num_qubits)

# --- Define the Energy Evaluation Function ---
# Given parameters params, assigns them to the ansatz circuit and evaluates
# the expectation value of the Hamiltonian. Returns the energy to the optimizer.
# Includes basic exception handling for robustness.
def energy(params, ansatz, hamiltonian, estimator):
    """Evaluate energy for given ansatz parameters."""
    try:
        result = estimator.run([(ansatz, hamiltonian, params)]).result()
        energy_estimate = result[0].data.evs
        print(f"Energy: {energy_estimate}")
        return energy_estimate
    except Exception as e:
        print(f"Estimator failed: {e}")
        return np.inf

# --- Initialize Parameters ---
# Random initialization of ansatz parameters in the full 0-2 pi range.
# Good starting point for global exploration of energy landscape.
initial_params = np.random.uniform(0, 2 * np.pi, size=ansatz.num_parameters)

# --- Classical Optimization ---
# Minimize energy using COBYLA, a derivative-free classical optimizer.
# Minimizes the energy function over the variational parameters.
# Hybrid quantum-classical loop: quantum subroutine evaluates energy,
# classical subroutine updates parameters.
opt_result = minimize(
    energy,
    initial_params,
    args=(ansatz, hamiltonian, estimator),
    method="COBYLA",
    options={"maxiter": 200, "disp": True}
)

# --- Output Results ---
# Outputs the final optimized parameters and estimated ground state energy.
final_params = opt_result.x
final_energy = energy(final_params, ansatz, hamiltonian, estimator)

print(f"\nFinal Optimized Energy: {final_energy}")
print(f"Optimized Parameters: {final_params}")
```