

A Tutorial on Quantum Dynamics Simulations on Quantum Computers. Part I: Closed Systems

Brandon Allen,[†] Delmar Guido Azevedo Cabral,[†] Nam P. Vu,^{†,‡} Cameron

Cianci,[¶] Xiaohan Dan,[†] and Victor S. Batista^{*,†,§}

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

[‡]*Department of Chemistry, Lafayette College, Easton, PA 18042, USA*

[¶]*Department of Physics, University of Connecticut, Storrs, CT 06511, USA*

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

E-mail:

Contents

1	Introduction	3
2	Chemical Dynamics Simulations	4
3	Simulations on Digital Computers	6
3.1	Split-Operator Fourier Transform Method	7
3.2	QuTiP Methods	12
4	Qubit-Based Simulations	13
4.1	Setting up a quantum dynamics calculation on the IBM Quantum platform with Qiskit	14
4.2	Encoding an arbitrary Hamiltonian in the basis of Pauli matrices	19

4.3	Quantum Split-Operator Fourier Transform Method	22
4.4	Simulating an Hamiltonian Expressed in the Basis of Pauli Matrices	26
4.4.1	Implementing Real-Time Dynamics with Trotterization	30
4.4.2	More Compact Trotterization Scheme	31
4.4.3	Initializing a Quantum Circuit with Qiskit	36
4.4.4	Qubit-based Quantum Experiments	38
4.5	Using the Hadamard Test for Calculating Expectation Values	41
4.5.1	Hadamard Test Function	42
4.5.2	Processing the Hadamard Test Results	43
4.5.3	How to execute the Hadamard test for our operator?	44
4.6	Variational Quantum Real Time Evolution	47
4.7	Variational Quantum Eigensolver	49
5	Qumode-Based Simulations	51
5.1	Hamiltonian propagation	55
5.2	Variational Quantum Eigensolver	60
6	Conclusion	65
	Acknowledgement	66
	References	66
7	Appendix	68
7.1	Source Code	68
7.2	Software Requirements	68

Abstract

Quantum dynamics simulations of molecular model systems are an invaluable tool to investigate a wide range of processes across multiple fields of research, including energy and charge transport, electronic and nuclear processes involved in bond-breaking

and bond-forming reactions, photochemistry as well as systems of interest in condensed-phase physics, nanoscience, molecular electronics, quantum optics, spectroscopy and quantum information processing. This tutorial provides a user friendly introduction to quantum dynamics simulations on quantum computers, suitable to a diverse audience. It covers the basics with hands-on examples gradually increasing in complexity. The tutorial is split in three parts. Part I focuses on closed quantum systems, using techniques like the Suzuki-Trotter expansion and sum of unitaries to integrate the time-dependent Schrödinger equation. Part II delves into of open quantum systems, using popular methods like the Lindblad equation that rely on the Markovian approximation. Part III focuses on dynamics simulations of non-Markovian open quantum systems based on generalized quantum master equations, using numerical approaches based on the Sz.-Nagy dilation theorem, or parallelization of linear expansions of the non-unitary propagators. We include Python, qiskit, and Strawberryfields codes examples which can be used a starting point for development of more advanced implementations.

1 Introduction

Quantum computing is emerging as a new computational paradigm that could enable simulations of quantum processes with favorable scaling. Significant efforts have been focused on quantum computing for electronic structure calculations including applications to molecular systems and materials. Here, we focus on how to actually simulate quantum dynamics on classical and quantum computers, as necessary for the description of chemical reactivity, and dynamical processes such as energy or charge transfer, and the calculation of time dependent expectation values, or correlation functions often computed for simulations of spectroscopy. This tutorial article aims to provide researchers with no experience in quantum dynamics or quantum computing with computational tools and detailed descriptions of algorithms and methodologies to make use of quantum computers for a wide range of applications.

Part I of this tutorial is focused on quantum computing simulations of closed quantum

systems by numerical integration of the time-dependent Schrodinger equation. Closed quantum systems refer to systems in which the energy remains conserved over time. This aligns with the well-known principles of quantum mechanics, where a system’s evolution follows a unitary process. We introduce the methods as applied to the dynamics of simple model systems that gradually increase in complexity, including the dynamics of a harmonic oscillator that allows for validation of the codes by direct comparisons to the analytical solution, and a simple model of chemical reaction described by a double-well potential along the reaction coordinate.

Part II is focused on simulations of the dynamics of Markovian open quantum systems. These methods are based on the integration of quantum master equations to account for dissipation due to the exchange of energy with a surrounding environment. These methods are crucial for simulations of molecular systems interacting with a surrounding solvent environment of material environment since accounting for dissipation mechanisms is essential for accurate descriptions of transport and reaction dynamics.

Part III is focused on simulations of non-Markovian open quantum systems, based on generalized quantum master equations. These methods are essential for simulations of photoinduced reaction dynamics, and ultrafast reaction dynamics, where the time scale of the processes of interest is comparable to the decay time of the reservoir correlation functions.

2 Chemical Dynamics Simulations

The field of chemical dynamics is concerned with describing the time-evolution of chemical systems of physical interest. The dynamics of a closed system is governed by the time-dependent Schrödinger Equation:

$$i\hbar \frac{\partial}{\partial t} |\psi(t)\rangle = \hat{H} |\psi(t)\rangle \tag{1}$$

where \hat{H} is the Hamiltonian for the system of interest. If \hat{H} is time-independent, Equation 1 becomes a linear ordinary differential equation with the following closed-form solution:

$$\begin{aligned}\frac{\partial|\Psi(t)\rangle}{\partial t} &= -\frac{i}{\hbar}\hat{H}|\Psi(t)\rangle \\ \int_0^\tau \frac{\partial|\Psi(t)\rangle}{|\Psi(t)\rangle} &= \int_0^\tau -\frac{i}{\hbar}\hat{H}\partial t \\ \ln|\Psi(\tau)\rangle - \ln|\Psi(0)\rangle &= -\frac{i}{\hbar}\hat{H}\tau \\ \Psi(\tau)\rangle &= e^{-i\hat{H}\tau/\hbar}|\Psi(0)\rangle,\end{aligned}$$

where the atomic unit (au) is used and $\hbar = 1$ is set throughout this series of tutorial.

Quantum dynamics of a closed system is therefore obtained from the solution of Equation 2. This equation can be interpreted as evolving an initial state $|\psi_0\rangle \equiv |\psi(0)\rangle$ according to the time-dependent Schrödinger Equation under the effect of a Hamiltonian \hat{H} describing the system of interest. The initial state is the wavefunction of any quantum particle of interest, such as a collection of electrons or nuclei that define a molecular system. The Hamiltonian $\hat{H} = \hat{T} + \hat{V}$ describes the energetics of the system, where $\hat{T} = \hat{p}^2/2m$ is the kinetic energy operator and \hat{V} is the potential energy operator corresponding to potential energy surfaces (PES).

With the time-evolved state, we can compute quantities of interest, such as the auto-correlation function describing the overlap between the initial state and the time-evolved wavefunction:

$$\xi(t) = \langle\psi_0|\psi(t)\rangle \tag{2}$$

or an expectation value of some observable $\hat{\mathcal{O}}$:

$$\langle\mathcal{O}\rangle = \langle\psi(t)|\hat{\mathcal{O}}|\psi(t)\rangle \tag{3}$$

To illustrate these calculations in a more concrete way, we will first look at some classical methods for computing quantum dynamics and demonstrate how to compute observables so

that we can “benchmark” the calculations run on a quantum computer.

3 Simulations on Digital Computers

In this section we will explore methods for simulating quantum dynamics on a digital computer (such as a laptop). We will illustrate these methods as applied to a benchmark system, the Harmonic Oscillator, for which an analytical solution is available and can be used to validate the simulation methods. We will then apply these methods to simulate the dynamics of a prototypical double-well model for chemical reactivity, for which an analytical solution does not exist, as is the case for all but the simplest chemical problems.

The sections are structured as follows: we initialize the state within a given basis, then integrate the time-dependent Schrödinger equation to evolve that state in time. With the propagated states, we calculate expectation values of observables and visualize the evolution of these values over time.

We introduce two methods for doing this with digital computers: the Split-Operator Fourier Transform (SOFT) method and a direct matrix-vector multiplication method as implemented in QuTiP.^{1,2}In reality, both of these methods are performing matrix-vector multiplication where the propagator is prepared as a matrix and the wavefunction of complex-valued amplitudes is prepared as a vector. The difference in these methods lies in the basis used for representing the problem. SOFT represents the Hamiltonian and wavefunction in the coordinate basis where the methods utilizing QuTiP^{1,2} represent the Hamiltonian and wavefunction in the basis of eigenvectors of the quantum harmonic oscillator. The methods introduced and demonstrated can serve as a starting point for more sophisticated simulation methods and can also be used to describe other systems by modifying the Hamiltonian.

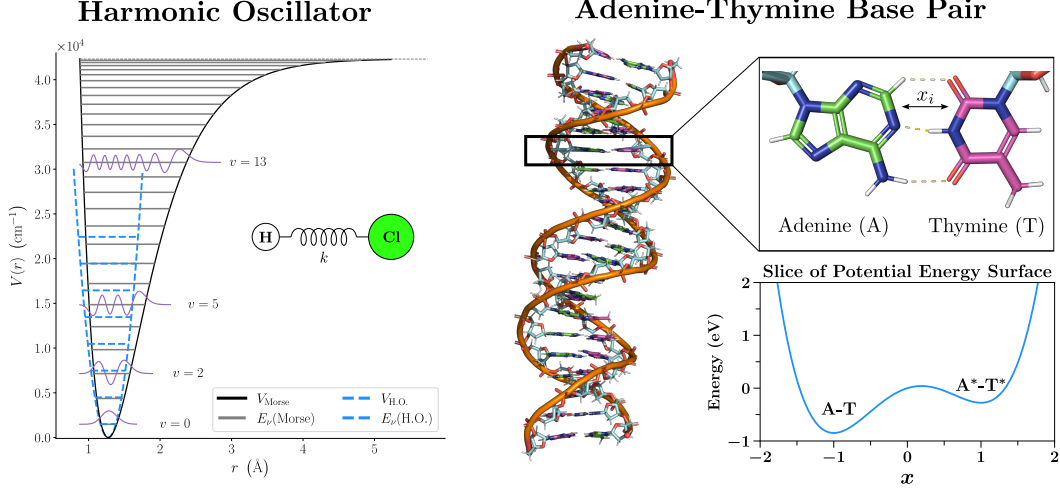


Figure 1: Depiction of two example systems used in Section 3. The Harmonic Oscillator (left) can be used as a rough approximation of the vibrational motion of a diatomic molecule, such as HCl. (Right) The asymmetric double well corresponding to a slice of the potential energy surface of base pairs in DNA representing the base pair between Adenine and Thymine.

3.1 Split-Operator Fourier Transform Method

As an illustrative introduction to simulations on digital computers, we consider a simple example of computing dynamics for the Quantum Harmonic Oscillator using the Split-Operator Fourier Transform (SOFT) method. We must first define a closed range of positions x and momenta p and discretize over some finite number of points N :

$$x_j = (j - N/2) dx$$

$$p_j = p_0 + (j - N/2) dp$$

with $dx = (x_{max} - x_{min.})/N$, initial momentum p_0 , and $dp = 2\pi/(x_{max} - x_{min.})$. We compute the time-evolution of wavepacket defined as a Gaussian in the position basis by initializing amplitudes at the discrete position values x_j :

$$\psi_0(x_j) = \left(\frac{1}{\pi}\right)^{1/4} \exp\left(-\frac{1}{2} [(x_j - x_0)^2 + ip_0 x_j]\right) \quad (4)$$

with initial displacement x_0 and momentum p_0 .

These steps are shown in Script 3.1.

Script 3.1: Wavepacket Initialization

```
1 import numpy as np
2 def psi0(x, p0, x0):
3     #Generate the initial wavefunction (psi0).
4     y = (1/np.pi)**(0.25)*np.exp(-1*(1/2)*((x - x0)**2)+1j*p0*x)
5     return y
6
7 N = 64
8 xmax = 5.0
9 xmin = -5.0
10 x0 = 1.0
11 p0 = 0.0
12 mass = 1.0
13 omega = 1.0
14 hbar = 1.0
15 dx = (xmax - xmin)/N
16 dp = 2*np.pi/(xmax-xmin)
17 pv = np.asarray([p0 + (j - N/2) * dp for j in range(N)])
18 xv = np.asarray([(j - N/2)*dx for j in range(N)])
19 psio = psi0(xv, p0, x0)
```

Now that we have an initial wavepacket, we focus on propagating it in time. The total propagation time t is discretized into N_{steps} time steps: $\tau = t/N_{\text{steps}}$. We obtain the wavepacket at intermediate times t_k by sequential application of the propagator:

$$\psi(x, t_k) = \int dx' \langle x | e^{-iH\tau} | x' \rangle \langle x' | \psi(t_{k-1}) \rangle \quad (5)$$

If a sufficiently small time step is used we can accurately approximate the time-evolution operator to second-order with the Trotter expansion:

$$e^{-iH\tau} = e^{-\frac{i}{\hbar}V(x)\tau/2} e^{-\frac{i}{\hbar}\frac{p^2}{2m}\tau} e^{-\frac{i}{\hbar}V(x)\tau/2} + \mathcal{O}(\tau^3) \quad (6)$$

Note that we have effectively separated the propagator into a product of three operators, each of which has dependence on only position (x) or momentum (p). We now have to represent these operators as arrays. The potential energy propagator can be easily calculated with our array of x -values. The harmonic oscillator potential is defined as:

$$V(x) = \frac{1}{2}m\omega^2x^2 \quad (7)$$

and the kinetic energy is represented in terms of momentum as:

$$T(p) = \frac{p^2}{2m} \quad (8)$$

Script 3.2: SOFT Operators for Harmonic Oscillator



```

1 import numpy as np
2 import scipy.linalg as splA
3 def harmonic(xgrid, mass, omega):
4     # This function generates a 1D Harmonic Oscillator Potential.
5     pot = 0.5*mass*(omega**2)*(xgrid**2)
6     return pot
7
8 Nsteps = 100 # Number of steps
9 total_time = 20.0 # Total propagation time
10 tau = total_time/Nsteps
11 # Potential Energy
12 V_ho = harmonic(xv)
13 V_prop = splA.expm(-1.j / hbar * V_ho * tau / 2.)
14 # Kinetic Energy
15 KE_ho = pv**2 / (2. * mass)
16 KE_prop = splA.expm(-1.j / hbar * KE_ho * tau)

```

Conveniently, there is a Fourier “partner” relationship between position and momentum (as they are conjugate variables) that allows us to convert between position space and momentum space, so that we are in the correct basis when applying the potential energy and kinetic energy propagators. This allows us to utilize the Fourier Transform (FT) and Inverse Fourier Transform (IFT) to convert between the position and momentum bases. Propagation for a single timestep is then:

$$\psi\left(x, \frac{t_{i+1}}{N}\right) = \overbrace{e^{\frac{-iV(x)\tau}{2\hbar N}}}^{\text{P.E. Prop.}} \cdot \overbrace{\int \frac{dp}{\sqrt{2\pi\hbar}} e^{\frac{-ipx}{\hbar}}}_{\text{IFT}} \cdot \overbrace{e^{\frac{-ip^2\tau}{2m\hbar N}}}^{\text{K.E. Prop.}} \cdot \overbrace{\int \frac{dx}{\sqrt{2\pi\hbar}} e^{\frac{ipx}{\hbar}}}_{\text{FT}} \cdot \overbrace{e^{\frac{-iV(x)\tau}{2\hbar N}}}^{\text{P.E. Prop.}} \cdot \psi(x, t_i) \quad (9)$$

To summarize the formula above, the algorithm will consist of 5 steps per iteration:

1. Apply a half step of the potential energy propagator to the initial state.
2. Fourier transform into the momentum basis.
3. Apply a full step of the kinetic energy propagator on the momentum basis.
4. Inverse Fourier transform back into the coordinate basis.
5. Apply the second half step of the potential energy propagator.

The SOFT routine for a single time step can be implemented with the following code.

Script 3.3: SOFT Propagation for a Single Time Step



```
1 # SOFT propagation
2 def soft(psi,pot_prop,kin_prop):
3     out=pot_prop*psi           # Apply P.E. propagator
4     out=kin_prop*np.fft.fft(out) # FT and apply K.E. propagator
5     out=pot_prop*np.fft.ifft(out) # IFT and Apply P.E. propagator
6     return out
```

With snapshots of the time-evolved wavepacket, we can compute the expectation values of observables as a function of the propagation time. Two natural observables for this system are the position and momentum operators, for which the expectation values can be calculated as:

$$\langle x \rangle(t) = \sum_j^N \psi^*(x_j, t) x_j \psi(x_j, t) dx$$
$$\langle p \rangle(t) = \sum_j^N \psi^*(p_j, t) p_j \psi(p_j, t) dp$$

With snapshots of the time-evolved wavepacket, we can compute the expectation values of observables as a function of the propagation time. Two natural observables for this system

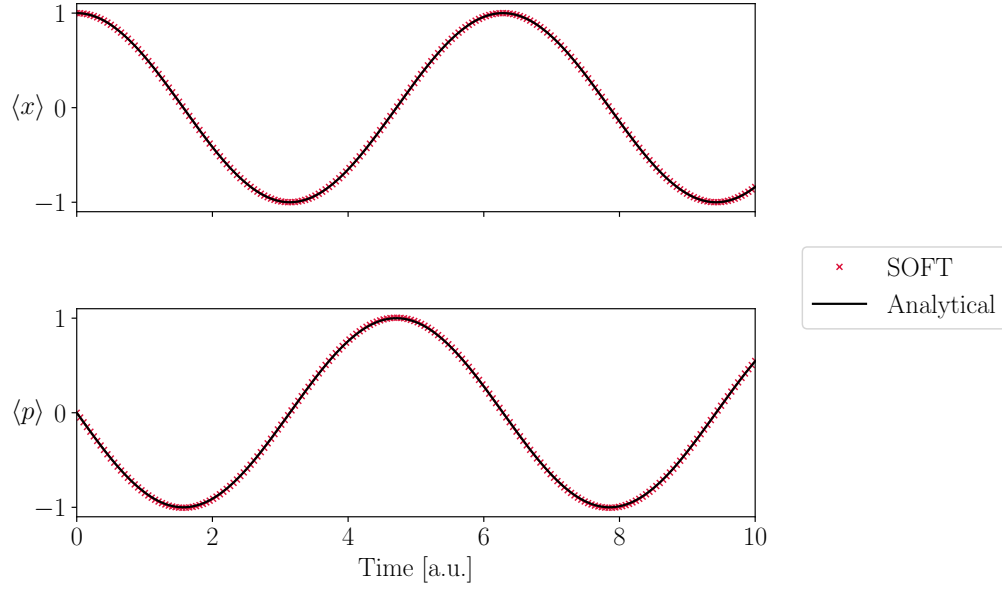


Figure 2: Time-dependent expectation values as computed with the propagated wavepacket using SOFT and the analytical values.

are the position and momentum operators, for which the expectation values can be calculated as:

$$\langle x \rangle(t) = \sum_j^N \psi^*(x_j, t) x_j \psi(x_j, t) dx$$

$$\langle p \rangle(t) = \sum_j^N \psi^*(p_j, t) p_j \psi(p_j, t) dp$$

Figure 2 shows the result of $\langle x \rangle(t)$ and $\langle p \rangle(t)$ for a harmonic oscillator with $V(x) = \frac{1}{2}m\omega^2x^2$, where the analytical result is

$$\langle x \rangle_{ana}(t) = x_0 \cos \omega t + \frac{p_0}{m\omega} \sin \omega t$$

$$\langle p \rangle_{ana}(t) = -m\omega x_0 \sin \omega t + p_0 \cos \omega t .$$

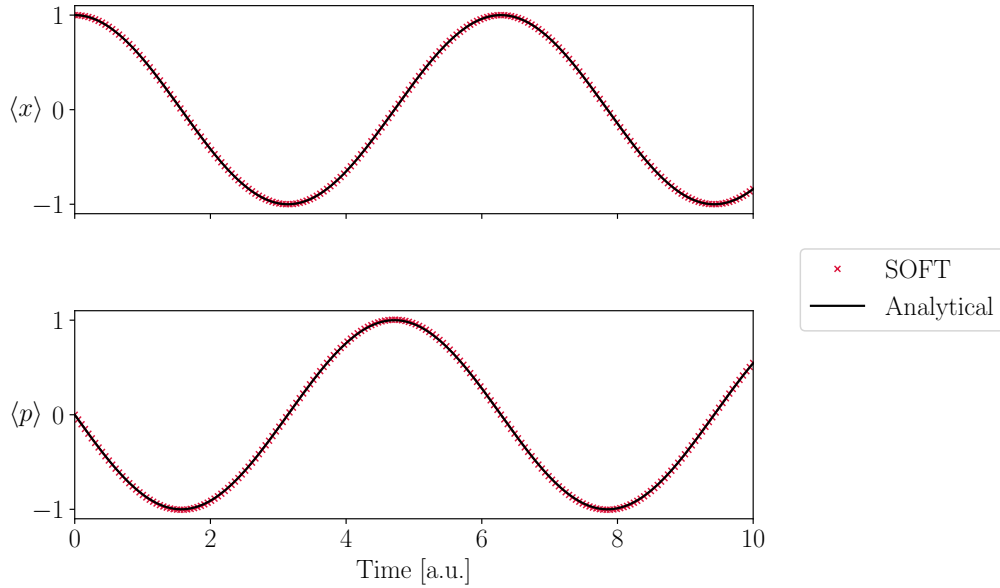


Figure 3: Time-dependent expectation values as computed with the propagated wavepacket using SOFT and the analytical values.

3.2 QuTiP Methods

We can now compute the same dynamics in second-quantized form using QuTiP's^{1,2} `mesolve` function. When we run dynamics, we must do the following:

1. **Define the initial state.** In this example, the initial state is defined as a coherent state with complex amplitude α , which can be expressed in the Fock Basis as:

$$|\alpha\rangle = e^{-\frac{1}{2}|\alpha|^2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle$$

2. **Define the Hamiltonian.** In this example, the Hamiltonian is the familiar quantum harmonic oscillator Hamiltonian, defined in terms of creation and annihilation operators as:

$$H = \hbar\omega \left(\hat{a}^\dagger \hat{a} + \frac{1}{2} \right)$$

3. **Define simulation parameters.** Define the propagation time step t and the number of time steps n .

4. Compute the time-evolved wavefunction at each timestep as:

$$|\alpha(t_{i+1})\rangle = e^{-\frac{i}{\hbar}Ht} |\alpha(t_i)\rangle$$

The process for running this simulation is shown in the following code box.

Script 3.4: Harmonic Oscillator Dynamics with QuTiP



```
1 import qutip as qt
2 # Define the Number of States in Fock Basis
3 N = 128
4 a = qt.destroy(N) # Define annihilation operator
5 # Define the Initial State:
6 xo = 1.00 # Initial Position
7 po = 0.00 # Initial Momentum
8 # Define a coherent state with amplitude alpha
9 psi0 = qt.coherent(N, alpha=(xo+1.j*po)/np.sqrt(2))
10 # Define the Hamiltonian:
11 mass = 1 # mass = 1.0
12 hbar = 1 # hbar = 1.0
13 omga = 1.0 # Oscillator frequency
14 H_ho = hbar*omga*(a.dag()*a + 1./2.) # Harmonic Oscillator Hamiltonian
15
16 # Define the propagation time array with n_steps from 0 to total_time
17 n_steps = 200 # Number of time steps
18 total_time = 10 # Total Propagation time
19 # Define the list of times for which we calculate dynamics.
20 tlist = np.linspace(0, total_time, 200)
21
22 # Run dynamics!
23 result = qt.mesolve(H_ho, psi0, tlist, [], [], progress_bar=True,
  → options=qt.solver.Options(nsteps=len(tlist)))
```

4 Qubit-Based Simulations

This section aims to explain qubit-based quantum computation, the most widely applied form of quantum computing. A qubit is a 2-level quantum state generally defined with the following statevector $|\alpha\rangle$:

$$|\alpha\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle, \tag{10}$$

where α_0 and α_1 are complex-valued expansion coefficients with $|\alpha_0|^2 + |\alpha_1|^2 = 1$. These coefficients store the information for computation. For a multi-qubit quantum state, the number of expansion coefficients scale exponentially with the number of qubits, enabling an exponentially large storage space that forms a foundation for quantum advantage.

In this section, we first demonstrate how a qubit-based quantum simulation is carried out, with a simulator and with an IBM quantum computer. Utilizing the IBM Quantum platform and qiskit package, we are going to show the implementation of the dynamics of a simple Heisenberg spin chain model and compare with classical benchmarking calculations. Then, we cover more advanced methods for simulations of complex systems. This includes encoding the arbitrary Hamiltonians into a sum of unitaries, exponentiation of Hamiltonian with Trotterization, and measuring the expectation values via the Hadamard test.

4.1 Setting up a quantum dynamics calculation on the IBM Quantum platform with Qiskit

Consider the following 2-site Hamiltonian:

$$\hat{H} = \frac{1}{2}(h_0\sigma_0^z + h_1\sigma_1^z) + \frac{J}{4}(\sigma_0^x\sigma_1^x + \sigma_0^y\sigma_1^y + \sigma_0^z\sigma_1^z), \quad (11)$$

where $h_0 = -0.5$, $h_1 = 0.5$ are the site energies and $J = 1$ is the coupling constant.

Script 4.1: Installing and importing necessary packages



```

1 !pip install qiskit --quiet
2 !pip install qiskit_ibm_runtime --quite
3 !pip install qiskit-ibmq-provider --quite
4 import numpy as np
5 import matplotlib.pyplot as plt
6 import scipy.linalg as LA
7 from qiskit import *
8 from qiskit.quantum_info.operators import Operator
9 from qiskit_ibm_runtime import QiskitRuntimeService, Options, Sampler

```

Script 4.2: 2-site Hisenberg spin chain Hamiltonian and propagator

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

In this subsection, we consider the time evolution for total time $\tau = 1a.u.$ governed by this Hamiltonian. To provide a benchmark for the upcoming quantum computations, we first run classical simulations following Sec. 2. As in the code above, the exponential propagator for this Hamiltonian, U , is prepared by simply exponentiating $-i\hat{H}\tau$.

The initial state is set at $|00\rangle$, which means both sites are in the spin-up state. The propagation is carried out by acting the exponential propagator on the initial state:

Script 4.3: Classical propagation of 2-site Heisenberg chain

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

The results can be viewed by simply printing out `psi_fin`.

Next we perform the corresponding IBM quantum setup. In the following, we initialize the qubits, prepare the unitary operator as a quantum circuit, and measuring the probability density by sampling the outcome of the circuit.

We first setup the initial state. The statevector in this case is a 4-vector, which corresponds to a 2-qubit state. Therefore we create a `QuantumRegister` with 2 wires. For each wire, the measured outcome needs to be recorded classically. The initial state is then created by simply calling the classical initial state. However, note that `qiskit` would set the vacuum state $|00\rangle$ as default, which is just the initial state that we want to set. Therefore, there is no need for explicitly calling the function for initialization.

Script 4.4: Quantum circuit for the 2-site Heisenberg chain propagation– initiation



```
1 qreg=QuantumRegister(2) # qreg is filled with two qubits
2 creg=ClassicalRegister(2) # creg is filled with two classical bits
3 entangler=QuantumCircuit(qreg,creg) # we put together our qreg and creg to make our
  ↳ Quantum Circuit, called entangler here.
4 #entangler.initialize(psi_init) #Not necessary since we start at |00>
```

Then, the circuit is prepared as a qiskit Operator object and appended to entangler:

Script 4.5: Quantum circuit for the 2-site Heisenberg chain propagation– gate construction



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

Finally, measurement is added to the end of the circuit.

Script 4.6: Quantum circuit for the 2-site Heisenberg chain propagation– measurement



```
1 entangler.measure(0,0) # measure the first qubit and record it in the first classical
  ↳ bit
2 entangler.measure(1,1)
```

You can then visualize the circuit with `entangler.draw()`. Running this circuit requires the use of the IBM quantum cloud service. To do that, first we need to register a IBM quantum account. It is convenient to access the account directly through the API token, as below:

Script 4.7: Accessing IBM account from API token



```
1 from qiskit import IBMQ
2 IBMQ.save_account('Your API token')
3 IBMQ.load_account()
```

First, we run the circuit with the QASM simulator. To do that, the `ibmq_qasm_simulator` backend is selected.

Script 4.8: Run dynamics with QASM simulator



```
1 service = QiskitRuntimeService()
2 options = Options()
3 backend = service.backend("ibmq_qasm_simulator")
4 sampler = Sampler(options=options, backend=backend)
5 job = sampler.run(circuits=entangler, shots=2000)
```

The sampled results are directly obtained as quasi probability distribution:

Script 4.9: Measure results from QASM simulator



```
1 qasm_result = np.zeros((4,))
2 qasm_result[0] = job.result().quasi_dists[0][0]
3 qasm_result[3] = job.result().quasi_dists[0][3]
```

Then we use the actual IBM machine. This is done by simply selecting another backend that corresponds to an available IBM machine (in this case, `ibmq_osaka`):

Script 4.10: Measure results from QASM simulator



```
1 backend = service.backend("ibmq_osaka")
2 sampler = Sampler(options=options, backend=backend)
3 job = sampler.run(circuits=entangler, shots=2000)
4 real_result = np.zeros((4,))
5 real_result[0] = job.result().quasi_dists[0][0]
6 real_result[1] = job.result().quasi_dists[0][1]
7 real_result[2] = job.result().quasi_dists[0][2]
8 real_result[3] = job.result().quasi_dists[0][3]
```

Now, the comparison between classical, QASM and real machine is visualized. You can see that the QASM results agrees exactly with the classical since no noise is introduced into the simulator. The real machine generates a moderate amount of noise.

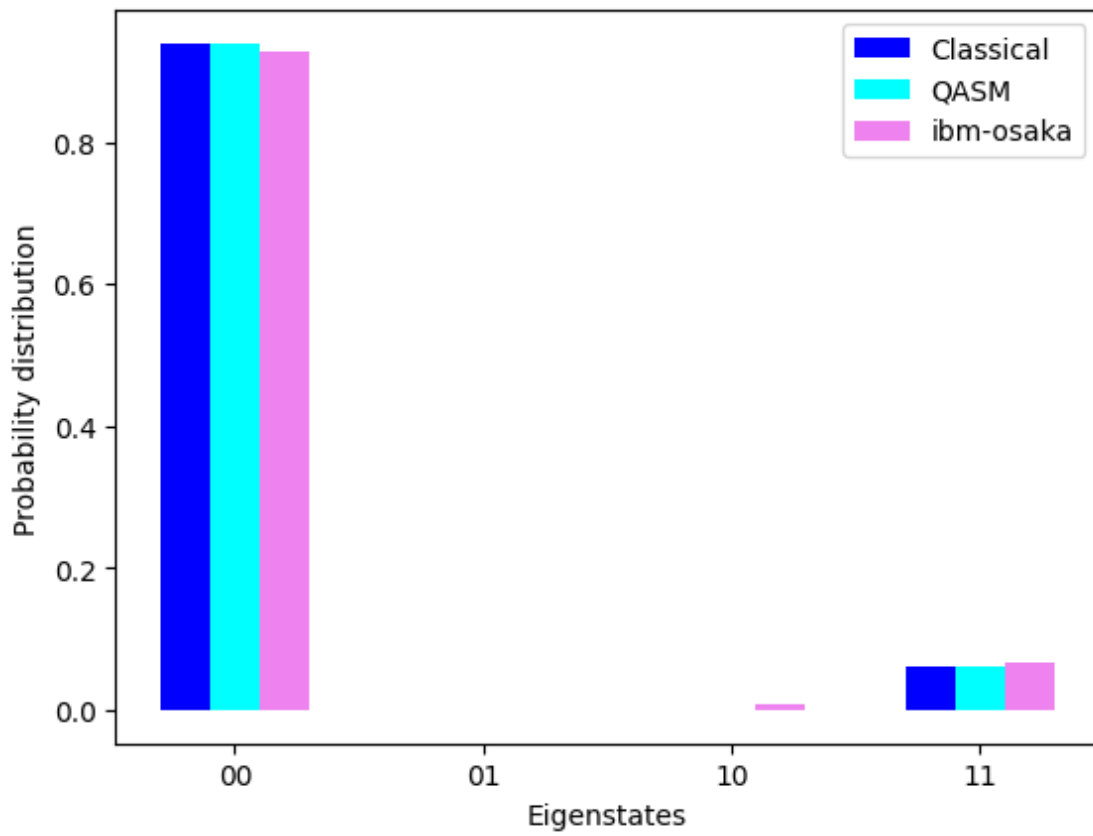


Figure 4: Probability distribution comparison between classical, QASM and full quantum computers.

4.2 Encoding an arbitrary Hamiltonian in the basis of Pauli matrices

In this section, we focus on the preparation of a quantum circuit that effectively implement the quantum dynamics guided by an arbitrary Hamiltonian. To perform Hamiltonian simulation on a qubit-based quantum computer, we can encode the Hamiltonian on the basis of Pauli matrices, which correspond to elementary quantum gates. The encoding allows us to express a given $N \times N$ matrix \tilde{M} as a sum of length n Pauli strings comprised of tensor products of Pauli matrices.

Where n is an integer that is rounded up by $\log_2(N)$. Expanding the \tilde{M} matrix to a $2^n \times 2^n$ matrix M by zero filling, then the decomposition is computed as a sum over all possible tensor products of unique combinations of n Pauli matrices. This linear combination can be written as:

$$M^{2^n \times 2^n} = \sum_{\xi_1, \xi_2, \dots, \xi_n} a_{\xi_1 \xi_2 \dots \xi_n} \bigotimes_{i=1}^n \sigma_{\xi_i} \quad (12)$$

with each σ_{ξ_i} is the Pauli matrix defined by $\xi_i \in \{\mathbf{I}, \mathbf{X}, \mathbf{Y}, \mathbf{Z}\}$. The 4^n coefficients $a_{\xi_1 \xi_2 \dots \xi_n}$ are computed as the Hilbert-Schmidt Inner Product of the input matrix M and the given Pauli string $\xi_1 \xi_2 \dots \xi_n$:

$$a_{\xi_1 \xi_2 \dots \xi_n} = \frac{1}{2^n} \mathbf{Tr} \left[\left(\bigotimes_{i=1}^n \sigma_{\xi_i} \right) \cdot M \right] \quad (13)$$

The above equation can be proved by substituting M using Equation (12)

$$\begin{aligned} \frac{1}{2^n} \mathbf{Tr} \left[\left(\bigotimes_{i=1}^n \sigma_{\xi_i} \right) \cdot M \right] &= \sum_{\xi'_1, \xi'_2, \dots, \xi'_n} \frac{1}{2^n} \mathbf{Tr} \left[\left(\bigotimes_{i=1}^n \sigma_{\xi_i} \right) \cdot a_{\xi'_1 \xi'_2 \dots \xi'_n} \bigotimes_{i=1}^n \sigma_{\xi'_i} \right] \\ &= \sum_{\xi'_1, \xi'_2, \dots, \xi'_n} \frac{1}{2^n} a_{\xi'_1 \xi'_2 \dots \xi'_n} \prod_{i=1}^n \mathbf{Tr} [\sigma_{\xi_i} \sigma_{\xi'_i}] \\ &= \sum_{\xi'_1, \xi'_2, \dots, \xi'_n} \frac{1}{2^n} a_{\xi'_1 \xi'_2 \dots \xi'_n} \prod_{i=1}^n 2 \delta_{\xi_i, \xi'_i} \\ &= a_{\xi_1 \xi_2 \dots \xi_n} \end{aligned} \quad (14)$$

While performing the decomposition, we must iterate through all possible tensor product combinations of n Pauli matrices. These are represented as strings which are translated into their corresponding arrays via a dictionary. To avoid looping over each element of each Pauli string we utilize vectorized dictionary querying, which allows us to convert the entire Pauli string with one query. The resulting tuple containing the Pauli string in matrix form is then used to evaluate the recursive Kronecker product of all elements of the tuple, resulting in a $N \times N$ matrix. This resulting matrix is used for evaluation of the Hilbert-Schmidt inner product. These are performed with utility functions, included in Script 4.11.

Script 4.11: Pauli Matrix Decomposition - Utility Functions



```

1 def vec_query(arr, my_dict):
2     '''
3     This function vectorizes dictionary querying.
4     It allows us to query `my_dict` with a np.array `arr` of keys.
5     This avoids a loop through the list of keys.
6     '''
7     import numpy as np
8     return np.vectorize(my_dict.__getitem__, otypes=[tuple])(arr)
9
10 def nested_kronecker_product(a):
11     '''
12     Handles Kronecker Products for list (i.e., a = [Z, Z, Z] will evaluate $Z \otimes Z
13     \leftrightarrow \otimes Z$).
14     Given list `a` this recursively evaluates the kronecker product of all elements.
15     This allows us to avoid having to call `np.kron` n-1 times for a list of length n.
16     '''
17     import numpy as np
18     if len(a) == 2:
19         return np.kron(a[0], a[1])
20     else:
21         return np.kron(a[0], nested_kronecker_product(a[1:]))
22
23 def Hilbert_Schmidt(mat1, mat2):
24     '''
25     Return the Hilbert-Schmidt Inner Product of two matrices.
26     This gives the coefficients for each term in the sum of tensor products of Paulis.
27     '''
28     import numpy as np
29     return np.trace(mat1.conj().T * mat2)

```

With these utility functions, we can then use the following to perform the Pauli Matrix decomposition:

Script 4.12: Pauli Matrix Decomposition



```

1 def decompose(Ham_arr, tol=10):
2     '''
3     Function that decomposes `Ham_arr` into a sum of Pauli strings.
4     '''
5     import numpy as np
6     import itertools
7     # Define a dictionary with the four Pauli matrices:
8     pms = {'I': np.array([[1, 0], [0, 1]], dtype=complex),
9           'X': np.array([[0, 1], [1, 0]], dtype=complex),
10          'Y': np.array([[0, -1j], [1j, 0]], dtype=complex),
11          'Z': np.array([[1, 0], [0, -1]], dtype=complex)}
12     pauli_keys = list(pms.keys()) # Keys of the dictionary
13     nqb = int(np.log2(Ham_arr.shape[0])) # Determine the # of qubits needed
14     output_string = '' # Initialize an empty string to which we can add our terms
15     # Make all possible combinations of nqb Pauli matrices
16     sigma_combinations = list(itertools.product(pauli_keys, repeat=nqb))
17     # Loop through each unique combination of Pauli Matrices
18     for ii in range(len(sigma_combinations)):
19         # Take the full Pauli string
20         pauli_str = ''.join(sigma_combinations[ii])
21         # Compute the coefficient for each Pauli string. This is done as follows:
22         # 1) Evaluate the tensor product of Pauli String to get a 2^n by 2^n matrix:
23         #    Turn each element of Pauli String ('ZIXI') into its corresponding
24         #    2 by 2 Pauli Matrix, then evaluate the Kronecker product.
25         # 2) Compute the Hilbert-Schmidt Inner Product.
26         # 3) Normalize by (1/(2^n)).
27         norm_factor = (1/(2**nqb))
28         # Convert the Pauli string to list of arrays
29         tmp_mat_list = vec_query(np.array(sigma_combinations[ii]), pms)
30         # Evaluate the Kronecker product of the matrix array
31         tmp_p_matrix = nested_kronecker_product(tmp_mat_list)
32         hs_innerprod = Hilbert_Schmidt(tmp_p_matrix, Ham_arr)
33         a_coeff = norm_factor * hs_innerprod
34         # If the coefficient is non-zero, we want to use it!
35         if a_coeff != 0.0:
36             # Assert that coefficients less than 10**(-tol) are 0.
37             min_val = 10**(-tol)
38             if abs(a_coeff) < min_val:
39                 pass
40             # If non-zero:
41             else:
42                 output_string += str(np.round(a_coeff.real, tol))+"*"+alt_name
43                 output_string += '+' # Add a plus sign for the next term!
44     return output_string[:-1] # To ignore that extra plus sign

```

4.3 Quantum Split-Operator Fourier Transform Method

As shown in Section 3, the SOFT method propagates a quantum state by acting split propagators that correspond to the potential and kinetic components, as well as Fourier transforms between the position and momentum representations to the initial state. Both split propagators and the forward/inverse Fourier transform operations are represented as unitary matrices and are directly implementable as quantum gates. Therefore, the SOFT method can be directly adapted for quantum computers to run dynamics on a quantum device. Figure 5 gives the circuit for the so-called quantum SOFT method. After a quantum state $|\psi\rangle$ is initialized, it passes the quantum gates that correspond to the potential/kinetic energy propagators as well as quantum Fourier transform (QFT) and its inverse, analogous to the process in Eq. (9).

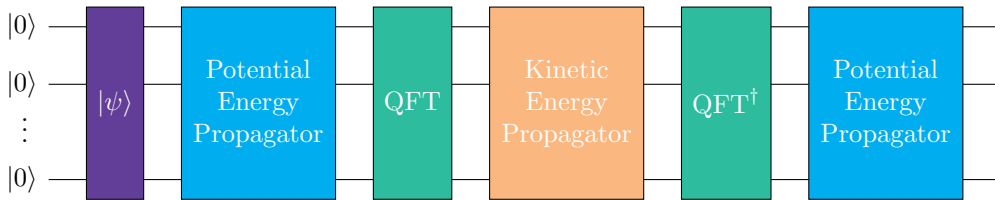


Figure 5: Representative quantum circuit for implementing the SOFT method.

Below we provide a detailed implementation of the quantum SOFT method, illustrated with a double-well potential parameterized for describing the hydrogen bonding within DNA adenine-thymine base pairs. Consider the following potential energy function:^{3,4}

$$V(x) = \alpha(0.429x - 1.126x^2 - 0.143x^3 + 0.563x^4), \quad (15)$$

where $\alpha = 0.1$ is the energy scaling parameter. Here, the position coordinate x describes the proton motion in an individual adenine-thymine (A-T) pair as it tautomerizes from the energetically favored amino–keto A-T form to the isomeric imino-enol A*-T* form.

Script 4.13: 1D PES for A-T tautomerization



```
1 d=6 # number of qubits
2 mass=1
3 xMin=-5
4 xMax=-xMin
5 x = np.linspace(xMin,xMax, num=2**d)
6 VV = (0.429*x-1.126*x**2-0.143*x**3+0.563*x**4) * 0.1
```

The initial state is described as a Gaussian wavepacket, as follows:

Script 4.14: Gaussian initial wavepacket



```
1 # Gaussian wavepacket on a grid
2 mu= 1
3 alpha = 1
4 psi = (alpha/np.pi)**(0.25) * np.exp(-alpha * (x-mu)**2 * 0.5)
5 psi/= np.sqrt(np.sum(np.abs(psi)**2))
```

The position and momentum operators, as well as their corresponding exponential propagators, are defined as in classical SOFT:

Script 4.15: Preparation of potential and kinetic split propagators



```
1 dx=(xMax-xMin)/(2**d-1)
2 # KE operator
3 dp=2*np.pi/(xMax-xMin)
4 N=2**d
5 p=np.zeros(N,dtype=float)
6 for i in range(N):
7     p[i]=dp*(i-N/2)
8 p=np.fft.fftshift(p)
9 time_step = 0.01
10 VVd_prop=np.diag(np.exp(-1j*VV*time_step)) #potential propagator
11 KEd_prop=np.diag(np.exp(-1j*p**2/2/mass*time_step)) #kinetic propagator
```

Next, real time propagation for $t = 60$ a.u. is carried out with the quantum circuit that corresponds to Fig. 5. Note in particular that the QFT operator in qiskit executes what would be defined as the inverse Fourier transform in numpy.

Script 4.16: Quantum SOFT circuit preparation



```
1 from qiskit.circuit.library import QFT
2 # Initialize an Empty Circuit
3 nqubits=d
4 q_reg=QuantumRegister(nqubits)
5 c_reg=ClassicalRegister(nqubits)
6 qc = QuantumCircuit(q_reg)
7
8 qc.initialize(psi,q_reg[:])
9 for k in trange(600):
10     bound_op = Operator(VVd_prop)
11     qc.append(bound_op, q_reg)
12     qc.append(QFT(d,do_swaps=True,inverse=True),q_reg)
13     bound_op = Operator(KEd_prop)
14     qc.append(bound_op, q_reg)
15     qc.append(QFT(d,do_swaps=True,inverse=False),q_reg)
```

Executing this circuit leads to the final, propagated state, psi_c:

Script 4.17: Quantum SOFT circuit execution



```
1 psin = execute(qc, backend=BasicAer.get_backend('statevector_simulator')).result()
2 psin = psin.get_statevector()
```

This result is then benchmarked by the classical SOFT result:



Script 4.18: Classical SOFT Benchmark



```
1 #Classical SOFT routine
2 def soft(fxy,emat,Pxy):
3     # soft propagation
4     out=emat*fxy
5     fp=np.fft.fft(out)*Pxy
6     out=np.fft.ifft(fp)
7     return out
8 psi_c_init = psi
9 psi_c = psi_c_init
10 iterations = 600
11 for i in range(iterations):
12     psi_c = soft(psi_c,np.diag(VVd_prop),np.diag(KEd_prop))
```

And the comparison between classical and quantum SOFT are plotted below. One can see that the two results match closely and describe the expected tautomerization through

the double well.

```
Script 4.19: Plotting initial and final wavefunctions  
```

```
1 # Visualization
2 plt.rcParams["figure.figsize"] = [12.50, 6.50]
3 #plt.plot(x,np.real(psin), label='real')
4 #plt.plot(x,np.imag(psin), label='imag')
5 plt.plot(x,np.abs(psin), label='abs, quantum SOFT')
6 plt.plot(x,np.abs(psi_c),'--', label='abs, classical SOFT')
7 plt.plot(x,VV,linewidth=3)
8 plt.plot(x,abs(psi),'x', label='initial')
9 leg = plt.legend(loc='upper right')
10 plt.ylim(-0.1,0.35)
11 plt.show()
```

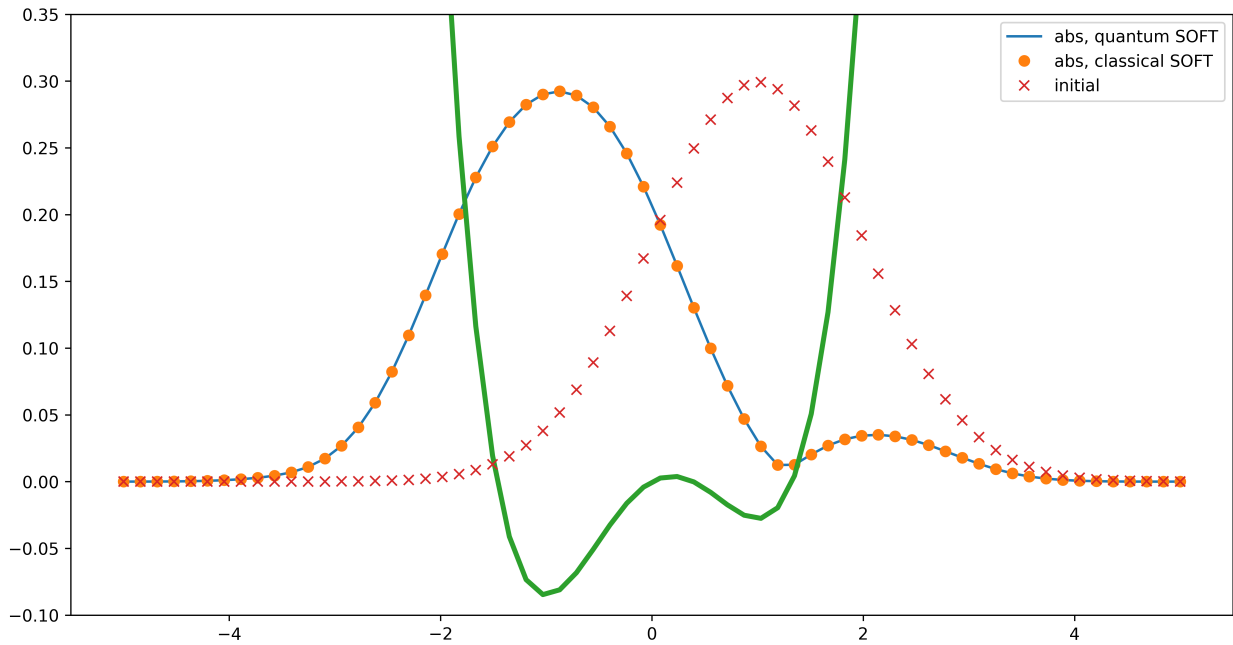


Figure 6: Probability densities of the initial and final quantum states for the DNA tautomerization model, as propagated by classical and quantum SOFT methods. Shown in green is the double-well PES.

4.4 Simulating an Hamiltonian Expressed in the Basis of Pauli Matrices

For this section we demonstrate how to encode and simulate the dynamics of a Hamiltonian expressed in the basis of Pauli matrices. We demonstrate this implementation using the Hamiltonian for the Heisenberg model, defined as follows:

$$H = \sum_{n=0}^{N-1} \Omega_n \sigma_n^z - \frac{1}{2} \sum_{n=0}^{N-2} (J_{n,n+1}^x \hat{\sigma}_n^x \hat{\sigma}_{n+1}^x + J_{n,n+1}^y \hat{\sigma}_n^y \hat{\sigma}_{n+1}^y + J_{n,n+1}^z \hat{\sigma}_n^z \hat{\sigma}_{n+1}^z) \quad (16)$$

where the coupling elements are described in terms of the $\sigma_x, \sigma_y, \sigma_z$ (Pauli X, Y and Z) matrices with a coupling associated with each type of interaction term for each site, Ω_n , or pair of sites, $J_{n,n+1}^p$, for $p \in \{x, y, z\}$.

This Hamiltonian can be used to describe the chemical process of electron transfer across a chromophore chain such as the functionalized graphene nanoribbon studied by Wang and coworkers.⁵ This system has alternating polymer sites containing radical character and the stability of the radical character at each site can be described by the on-site parameter, Ω_n , and the coupling between sites, $J_{n,n+1}$, governed by the properties of the linker regions containing the diketone groups. These parameters can be tuned by synthetic design of each component part of the nanoribbon.

As an example of the simulation of the spin-chain dynamics we consider the parameters used in the work by Fiori *et al.*,⁶ summarized in Table 1, for a reduced system size of 3 spin sites.

Table 1: Hamiltonian parameters used in the spin chain simulation⁶

Parameter	$n = 0$	$n \neq 0$
Ω_n	0.65	1.0
$J_{n,n+1}^x$	0.75	1.0
$J_{n,n+1}^y$	0.75	1.0
$J_{n,n+1}^z$	0.0	0.0

We use the same form of initial state as in the work by Fiori and coworkers,⁶ with a

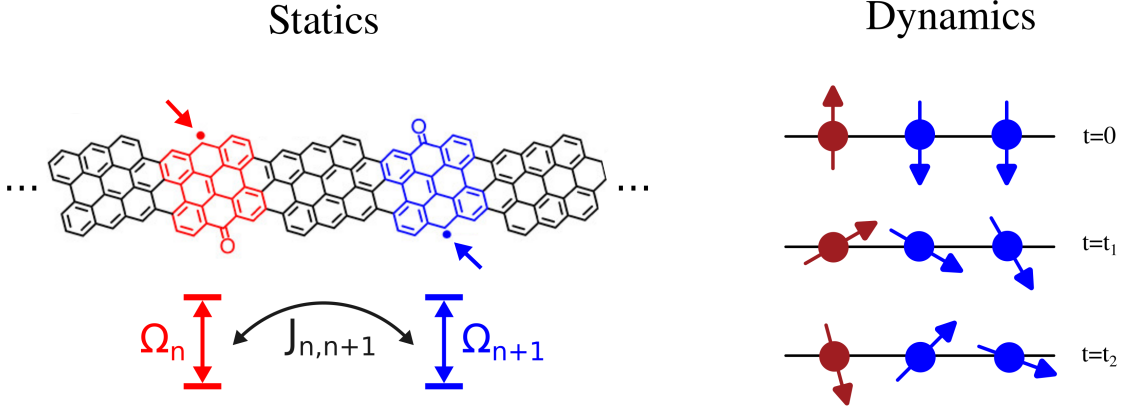


Figure 7: The static versus dynamical picture of the Heisenberg spin-chain. Left: The static onsite parameters of the Hamiltonian, Ω_n , encode the energy required to flip the spin state at a particular polymer site while the offsite couplings, $J_{n,n+1}$ encode the entanglement between the spins at each site. Right: The spin configuration of the system changes as function of time, but the total spin of the system is conserved under a closed dynamics formalism.

starting configuration of spin up, $|\uparrow\rangle = [1 \ 0]^T$, on the first site and spin down, $|\downarrow\rangle = [0 \ 1]^T$, on all others,

$$|\psi_0\rangle = |\uparrow\downarrow\downarrow\rangle = |\uparrow\rangle \otimes |\downarrow\rangle \otimes |\downarrow\rangle. \quad (17)$$

Although the dynamics of this Hamiltonian can be simulated in a classical computer, we could also use a quantum computer to simulate this same problem. One way to do so is by harnessing Qiskit, a python library containing functions that ease the simulation of the problem in quantum device framework.

If we seek to implement this Hamiltonian term-by-term within the qiskit framework, the goal is to implement each term, such as the $Y_n Y_{n+1}$.

$$\hat{\sigma}_n^y \hat{\sigma}_{n+1}^y = I \otimes I \cdots \otimes Y_n \otimes Y_{n+1} \otimes \cdots \otimes I$$

We can leverage their SparsePauliOp built-in representation, by supplying a string encoding the term and multiply it by an appropriate coefficient to implement each of the terms.

Script 4.20: Heisenberg Hamiltonian for site n



```
1 from qiskit.quantum_info import SparsePauliOp
2 def get_hamiltonian_n_site_terms(n, coeff, n_qubits):
3     XX_coeff = coeff[0]
4     YY_coeff = coeff[1]
5     ZZ_coeff = coeff[2]
6     Z_coeff = coeff[3]
7
8     XX_term = SparsePauliOp(("I" * n + "XX" + "I" * (n_qubits - 2 - n)))
9     XX_term *= XX_coeff
10    YY_term = SparsePauliOp(("I" * n + "YY" + "I" * (n_qubits - 2 - n)))
11    YY_term *= YY_coeff
12    ZZ_term = SparsePauliOp(("I" * n + "ZZ" + "I" * (n_qubits - 2 - n)))
13    ZZ_term *= ZZ_coeff
14    Z_term = SparsePauliOp(("I" * n + "Z" + "I" * (n_qubits - 1 - n)))
15    Z_term *= Z_coeff
16
17    return (XX_term + YY_term + ZZ_term + Z_term)
```

Furthermore, we can generate all such terms for a N-spin chain model by calling the prior function for each of the sites.

Script 4.21: Heisenberg Hamiltonian for N Sites



```

1 def get_heisenberg_hamiltonian(n_qubits, coeff=None):
2
3     # Three qubits because for 2 we get H_0 = 0
4     assert n_qubits >= 3
5
6     if coeff == None:
7         'Setting default values for the coefficients'
8         coeff = [[1.0, 1.0, 1.0, 1.0] for i in range(n_qubits)]
9
10    # Even terms of the Hamiltonian
11    # (summing over individual pair-wise elements)
12    H_E = sum((get_hamiltonian_n_site_terms(i, coeff[i], n_qubits)
13              for i in range(0, n_qubits-1, 2)))
14
15    # Odd terms of the Hamiltonian
16    # (summing over individual pair-wise elements)
17    H_O = sum((get_hamiltonian_n_site_terms(i, coeff[i], n_qubits)
18              for i in range(1, n_qubits-1, 2)))
19
20    # adding final Z term at the Nth site
21    final_term = SparsePauliOp("I" * (n_qubits - 1) + "Z")
22    final_term *= coeff[n_qubits-1][3]
23    if (n_qubits % 2) == 0:
24        H_E += final_term
25    else:
26        H_O += final_term
27
28    # Returns the list of the two sets of terms
29    return [H_E, H_O]

```

We can check the representation and correctness of the Hamiltonian at this point by making a call to the function and printing the operator representation:



```
1 num_q = 3
2 # XX YY ZZ, Z
3 ham_coeffs = ([[0.75/2, 0.75/2, 0.0, 0.65]]
4               + [[0.5, 0.5, 0.0, 1.0]
5                 for i in range(num_q-1)])
6
7 spin_chain_hamiltonian = get_heisenberg_hamiltonian(num_q,
8                                                     ham_coeffs)
9 print('Hamiltonian separated into even and odd components:')
10 print(spin_chain_hamiltonian)
11 print('Hamiltonian combining even and odd components:')
12 print(sum(spin_chain_hamiltonian))
```

4.4.1 Implementing Real-Time Dynamics with Trotterization

One of the major difficulties for solving the time-dependent Schrodinger equation with exponential propagator,

$$|\Psi(t)\rangle = e^{-i\hat{H}t/\hbar}|\Psi(0)\rangle, \quad (18)$$

lies in the fact that the exponential of Hamiltonian is hard to evaluate efficiently. In many cases, splitting the Hamiltonian into several components and evaluating their exponentials separately greatly facilitate the computation, as was shown in the SOFT scheme.

However, the components of the Hamiltonian do not necessarily commute and thus the Suzuki-Trotter decomposition is employed:

$$e^{\delta(\hat{A}+\hat{B})} = e^{\delta\hat{A}} \cdot e^{\delta\hat{B}} + O(\delta^2) \quad (19)$$

For small values of δ , this approximation is suitable. The code below implements the time evolution operator for the Heisenberg Hamiltonian, but it should be generalizable to any Hamiltonian written in the basis of Pauli matrices.

Script 4.23: Trotterized Time Evolution Operator



```

1 from qiskit.circuit.library import PauliEvolutionGate
2 def get_time_evolution_operator(num_qubits, tau, trotter_steps, coeff=None):
3     heisenberg_hamiltonian = get_heisenberg_hamiltonian(num_qubits, coeff)
4
5     evo_op = PauliEvolutionGate(heisenberg_hamiltonian, tau,
6                                 synthesis=SuzukiTrotter(order=2, reps=trotter_steps))
7     return evo_op.definition
8
9 num_q = 3
10 evolution_timestep = 0.1
11 n_trotter_steps = 1
12 ham_coeffs = ([[0.75/2, 0.75/2, 0.0, 0.65]]
13               + [[0.5, 0.5, 0.0, 1.0] for i in range(num_q-1)])
14 time_evo_op = get_time_evolution_operator(
15     num_qubits=num_q, tau=evolution_timestep,
16     trotter_steps=n_trotter_steps, coeff=ham_coeffs)
17 print(time_evo_op)

```

4.4.2 More Compact Trotterization Scheme

While this propagator is encoded in a black-box fashion, we can manually encode the Trotter decomposition for Hamiltonians containing 1 and 2-qubit Pauli operators. The general idea is to encode the 1-qubit terms as rotation gates (A) and the 2-qubit terms as the optimal representation (B,C). The 2-qubit terms can be grouped by layers targetting even and odd indices separately. The end goal is to generate circuits of the form, corresponding to a single Trotter step:

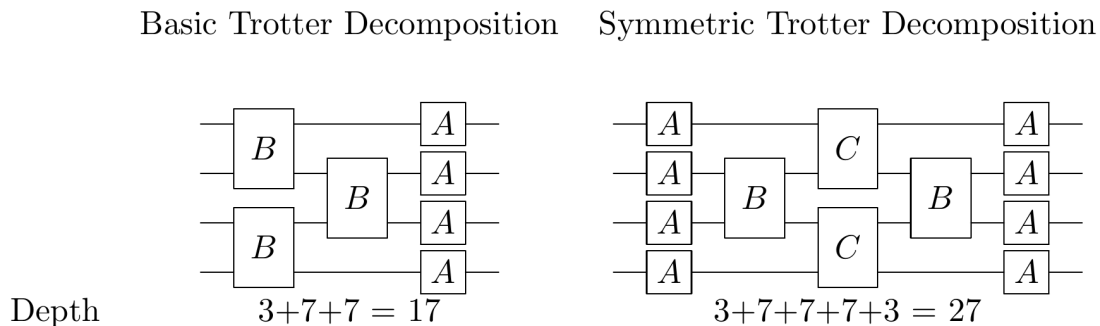


Figure 8: Representative quantum circuit for implementing the basic and symmetric Trotter decomposition for Hamiltonians expressed in terms of two-qubit Pauli operators.

First we can sort the Hamiltonian terms into 1-qubit, 2-qubit even and 2-qubit odd terms using the following function that takes a qiskit SparsePauliOp Hamiltonian:

Script 4.24: Sorting Terms by Interaction order



```
1 def find_string_pattern(pattern, string):
2     match_list = []
3     for m in re.finditer(pattern, string):
4         match_list.append(m.start())
5     return match_list
6
7 def sort_Pauli_by_symmetry(ham):
8
9     one_qubit_terms = []
10    two_qubit_terms = []
11    # separating the one-qubit from two-qubit terms
12    for term in ham:
13        matches = find_string_pattern('X|Y|Z', str(term.paulis[0]))
14        pauli_string = term.paulis[0]
15        coeff = np.real(term.coeffs[0])
16        str_tag = pauli_string.to_label().replace('I', '')
17        if len(matches) == 2:
18            two_qubit_terms.append((pauli_string, coeff, matches, str_tag))
19        elif len(matches) == 1:
20            one_qubit_terms.append((pauli_string, coeff, matches, str_tag))
21
22    # sorting the two-qubit terms according to index on which they act
23    two_qubit_terms = sorted(two_qubit_terms, key=lambda x: x[2])
24    # separating the even from the odd two-qubit terms
25    even_two_qubit_terms = list(filter(lambda x: not x[2][0]%2, two_qubit_terms))
26    odd_two_qubit_terms = list(filter(lambda x: x[2][0]%2, two_qubit_terms))
27
28    even_two_qubit_terms = [list(v) for i, v in groupby(even_two_qubit_terms,
29                                                         lambda x: x[2][0])]
30    odd_two_qubit_terms = [list(v) for i, v in groupby(odd_two_qubit_terms,
31                                                         lambda x: x[2][0])]
32
33    return one_qubit_terms, even_two_qubit_terms, odd_two_qubit_terms
```

Then we can encode each of the type of operators with a given circuit pattern. For the 1-qubit terms, we need to express the rotation angles as a function of the exponential argument. Since the $R_X(\theta)$, $R_Y(\theta)$ and $R_Z(\theta)$ rotation gates are given as follows,

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

we note that the angle $\theta/2 = h_i\tau \implies \theta = 2h_i\tau$.

Script 4.25: Circuit for Exponential of 1-Qubit Pauli Term



```

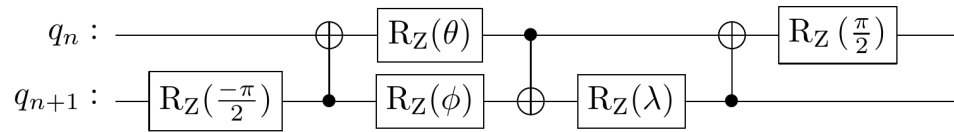
1 def generate_circ_pattern_1qubit(circ, term, delta_t):
2     coeff = 2 * term[1] * delta_t
3     if term[3] == 'X':
4         circ.rx(coeff, term[2])
5     elif term[3] == 'Y':
6         circ.ry(coeff, term[2])
7     elif term[3] == 'Z':
8         circ.rz(coeff, term[2])
9
10    return circ

```

For the 2-qubit terms we can use the following based on the optimal decomposition of $U(4)$:

$$U = (A_1 \otimes A_2)N(\alpha, \beta, \gamma)(A_3 \otimes A_4) \quad (20)$$

Where, $N(\alpha, \beta, \gamma) = \exp\{i(\alpha\sigma_x \otimes \sigma_x + \beta\sigma_y \otimes \sigma_y + \gamma\sigma_z \otimes \sigma_z)\}$ Since the $N(\alpha, \beta, \gamma)$ is exactly the term we seek to implement, we examine its circuit:



where,

$$\theta = \pi/2 - 2\gamma, \quad \phi = 2\alpha - \pi/2, \quad \lambda = \pi/2 - 2\beta \quad (21)$$

Note that $\alpha = J_{n,n+1}^x\tau, \beta = J_{n,n+1}^y\tau, \gamma = J_{n,n+1}^z\tau$ 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 implements the most general approach.

Script 4.26: Circuit for Exponential of 2-Qubit Pauli Term



```

1 def generate_circ_pattern_2qubit(circ, term, delta_t):
2
3     # wires to which to apply the operation
4     wires = term[0][2]
5
6     # angles to parameterize the circuit,
7     # based on exponential argument
8     if any('XX' in sublist for sublist in term):
9         g_phi = ( 2 * (-1) * term[0][1] * delta_t - np.pi / 2)
10    else:
11        g_phi = - np.pi / 2
12    if any('YY' in sublist for sublist in term):
13        g_lambda = (np.pi/2 - 2 * (-1) * term[1][1] * delta_t)
14    else:
15        g_lambda = np.pi/2
16    if any('ZZ' in sublist for sublist in term):
17        g_theta = (np.pi/2 - 2 * (-1) * term[2][1] * delta_t)
18    else:
19        g_theta = np.pi/2
20
21    # circuit
22    circ.rz(-np.pi/2, wires[1])
23    circ.cx(wires[1], wires[0])
24    circ.rz(g_theta, wires[0])
25    circ.ry(g_phi, wires[1])
26    circ.cx(wires[0], wires[1])
27    circ.ry(g_lambda, wires[1])
28    circ.cx(wires[1], wires[0])
29    circ.rz(np.pi/2, wires[0])
30    return circ

```

Finally we can make a function that assembles the manual Trotterization of exponential of Hamiltonians containing one and two-qubit Pauli operators, with the structures supported basic (BCA) and symmetric structures (ACBCA) indicated above. Furthermore, selecting more than 1 Trotter step will scale the exponential argument by dividing by the number of Trotter steps and include that number of repetitions of the Trotter circuit structure.

Script 4.27: Manual Trotterization of Propagator



```

1 def get_manual_Trotter(num_q, pauli_ops, timestep, n_trotter=1,
2                       trotter_type='basic', reverse_bits=True):
3     # sorts the Pauli strings according to qubit number they affect and symmetry
4     one_q, even_two_q, odd_two_q = sort_Pauli_by_symmetry(pauli_ops)
5     # scales the timestep according to the number of trotter steps
6     timestep_even_two_q = timestep / n_trotter
7     timestep_odd_two_q = timestep / n_trotter
8     timestep_one_q = timestep / n_trotter
9     # symmetric places 1/2 of one_q and odd_two_q before and after even_two_q
10    if trotter_type == 'symmetric':
11        timestep_odd_two_q /= 2
12        timestep_one_q /= 2
13    # constructs circuits for each segment of the operators
14    qc_odd_two_q, qc_even_two_q, qc_one_q = QuantumCircuit(num_q),
15    ↪ QuantumCircuit(num_q), QuantumCircuit(num_q)
16    for i in even_two_q:
17        qc_even_two_q = generate_circ_pattern_2qubit(qc_even_two_q, i,
18    ↪ timestep_even_two_q)
19    for i in odd_two_q:
20        qc_odd_two_q = generate_circ_pattern_2qubit(qc_odd_two_q, i,
21    ↪ timestep_odd_two_q)
22    for i in one_q:
23        qc_one_q = generate_circ_pattern_1qubit(qc_one_q, i, timestep_one_q)
24    # assembles the circuit for Trotter decomposition of exponential
25    qr = QuantumRegister(num_q)
26    qc = QuantumCircuit(qr)
27    if trotter_type == 'basic':
28        qc = qc.compose(qc_even_two_q)
29        qc = qc.compose(qc_odd_two_q)
30        qc = qc.compose(qc_one_q)
31    elif trotter_type == 'symmetric':
32        qc = qc.compose(qc_one_q)
33        qc = qc.compose(qc_odd_two_q)
34        qc = qc.compose(qc_even_two_q)
35        qc = qc.compose(qc_odd_two_q)
36        qc = qc.compose(qc_one_q)
37    # repeats the single_trotter circuit several times to match n_trotter
38    for i in range(n_trotter-1):
39        qc = qc.compose(qc)
40    if reverse_bits:
41        return qc.reverse_bits()
42    else:
43        return qc

```

The overall depth per layer of the manual Trotterization scheme is 15 gates for the basic decomposition and 23 gates for the symmetric decomposition. This function can be used in

place of the qiskit built-in Trotterization approach and codes included in this section.

Script 4.28: Manual Trotter Circuits



```
1 spin_chain_hamiltonian = get_heisenberg_hamiltonian(num_q,
2                                                     ham_coeffs)
3
4 spin_chain_hamiltonian = sum(spin_chain_hamiltonian)
5 print(get_manual_Trotter(num_q, spin_chain_hamiltonian,
6                           0.1).draw())
7 print(get_manual_Trotter(num_q, spin_chain_hamiltonian, 0.1,
8                           n_trotter=2).draw())
9 print(get_manual_Trotter(num_q, spin_chain_hamiltonian, 0.1,
10                          trotter_type='symmetric').draw())
11 print(get_manual_Trotter(num_q, spin_chain_hamiltonian, 0.1,
12                          n_trotter=2,
13                          trotter_type='symmetric').draw())
```

4.4.3 Initializing a Quantum Circuit with Qiskit

Depending on the number of quantum bits needed for simulation and the number of classical bits for recording measurement outcomes, we require a quantum circuit object containing the required form. However, the initial state for this circuit is the vacuum state, $|0\rangle \otimes |0\rangle \otimes |0\rangle = |000\rangle$:

Script 4.29: Quantum Circuit Initialization



```
1 from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
2 # specifying a quantum register with specific number of qubits
3 qr = QuantumRegister(num_q)
4 # classical register used for measurement of qubits
5 cr = ClassicalRegister(num_q)
6 # quantum circuit combining quantum and classical registers
7 qc = QuantumCircuit(qr, cr) # instantiated here
8 qc.draw(style='iqp')
9 print(qc)
```

We can initialize it by bit flipping to obtain the initial state $I|0\rangle \otimes X|0\rangle \otimes X|0\rangle = |011\rangle$

Script 4.30: Quantum Circuit for Vacuum State Initialization



```
1 from qiskit import execute
2 from qiskit import BasicAer
3
4 # specifying initial state by flipping qubit states
5 for qubit_idx in range(num_q):
6     if qubit_idx == 0:
7         # generate only one spin-up at first qubit
8         qc.id(qubit_idx)
9     else:
10        # make all other spins have the spin-down state
11        qc.x(qubit_idx)
12 qc.barrier()
13 qc.draw(style='iqp')
14 print(qc)
15
16 # checking the initial state
17 device = BasicAer.get_backend('statevector_simulator')
18 qc_init_state = execute(qc, backend=device).result()
19 qc_init_state = qc_init_state.get_statevector()
20 print(qc_init_state)
```

or by amplitude encoding ($|000\rangle \rightarrow |011\rangle$):

Script 4.31: State Initialization: Amplitude Encoding



```
1 qr_init = QuantumRegister(num_qubits)
2 qc_init = QuantumCircuit(qr_init)
3 qc_init.initialize('011', qr_init[:])
4 qc.append(qc_init)
```

Finally, we append the time evolution operator and check the overall quantum circuit structure and depth:

Script 4.32: Applying Time Evolution Operator to Circuit



```
1 # generating the time evolution operator for a specific set of
2 # hamiltonian parameters and timestep
3 time_evo_op = get_time_evolution_operator(num_qubits=num_q,
4     tau=evolution_timestep,
5     trotter_steps=n_trotter_steps,
6     coeff=hamiltonian_coefficients)
7
8 # appending the Hamiltonian evolution to the circuit
9 qc.append(time_evo_op, list(range(num_q)))
10 qc.barrier()
11 qc.draw(style='iqp')
12 print(qc)
13
14 # Depth check
15 print('Depth of the circuit is', qc.depth())
16 # transpiled circuit to statevector simulator
17 qct = transpile(qc, device, optimization_level=2)
18 qct.decompose().decompose()
19 qct.draw(style='iqp')
20 print(qct)
21
22 print('Depth of the circuit after transpilation is '
23       f'{qct.depth()}')
```

4.4.4 Qubit-based Quantum Experiments

To perform the quantum experiment we can use either a classical simulator, such as Statevector, which uses linear algebra to solve for the final circuit state based on the circuit operations, or an IBMQ device which would run on an experimental platform. Be mindful that quantum circuits should be relatively shallow in depth (≤ 100 linear operations). Furthermore, execution on an experimental setup requires transpilation to use the supported operations of the actual platform.

We start by implementing a function to implement the iterative statevector simulation by reading out the exact final state after propagation for a small time step:

Script 4.33: Execution of Quantum Experiment



```

1 import numpy as np
2 from qiskit import BasicAer, execute
3 from qiskit import QuantumCircuit, QuantumRegister
4
5 # Quantum circuit for propagation
6 def qsolve_statevector(psin, qc):
7     # Determining number of qubits from the length of the state vector
8     d = int(np.log2(n))
9     # Circuit preparation
10    qre = QuantumRegister(d)
11    circ = QuantumCircuit(qre)
12    circ.initialize(psin,qre)
13    circ.barrier()
14    circ.append(qc, qre)
15    circ.barrier()
16    # Circuit execution
17    device = BasicAer.get_backend('statevector_simulator')
18    psin = execute(circ, backend=device).result()
19    return psin.get_statevector()

```

And execute the function by using the previously mentioned parameters and initial state to obtain the absolute value of the survival amplitude observable:

$$|\langle \psi_0 | \psi_t \rangle| = \left| \langle \psi_0 | e^{-i\hat{H}t/\hbar} | \psi_0 \rangle \right| \quad (22)$$

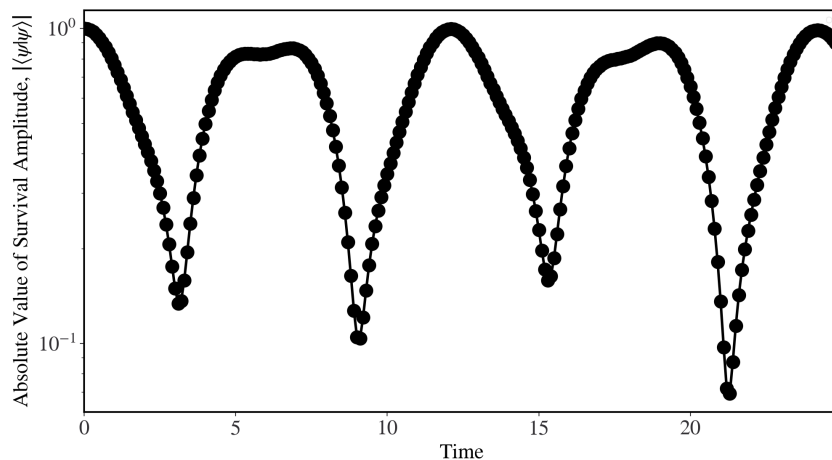


Figure 9: Absolute value of the survival amplitude calculated using the statevector approach, in agreement with the classical benchmark.



```
1 # Qubit basis states
2 zero_state = np.array([[1], [0]])
3 one_state = np.array([[0], [1]])
4
5 # For a 011 initial state prepare as follows
6 psin = zero_state # for the first spin
7 # iterates over the remaining spins, by performing Kronecker Product
8 for i in range(num_q-1):
9     psin = np.kron(psin, one_state)
10 psin0 = psin.flatten()
11 print(psin0)
12
13 nsteps = 250
14 psin_list = []
15 psin_list.append(psin0)
16 correlation_list = []
17 # performs dynamical propagation by statevector re-initialization
18 for k in range(nsteps):
19     print(f'Running dynamics step {k}')
20     if k > 0:
21         psin = qsolve_statevector(psin_list[-1], time_evo_op)
22         # removes the last initial state to save memory
23         psin_list.pop()
24         # stores the new initial state
25         psin_list.append(psin)
26     correlation_list.append(np.vdot(psin_list[-1], psin0))
27
28 t = np.arange(0, evolution_timestep*(nsteps),
29             evolution_timestep)
30 np.save(f'{num_q}_spin_chain_time', t)
31 sa_observable = np.abs(correlation_list)
32 np.save(f'{num_q}_spin_chain_SA_obs', sa_observable)
33 # plotting
34 plt.plot(t, sa_observable, '-o')
35 plt.ylabel('Survival Amplitude')
36 plt.yscale('log')
37 plt.show()
```


4.5 Using the Hadamard Test for Calculating Expectation Values

The real and imaginary parts of the expectation value $\langle \psi | U | \psi \rangle$ of a unitary operator U can be obtained by measuring an ancilla qubit in a circuit of a Hadamard test, as shown in Fig. 10. To understand these circuits, let's follow the evolution of the state as it evolves

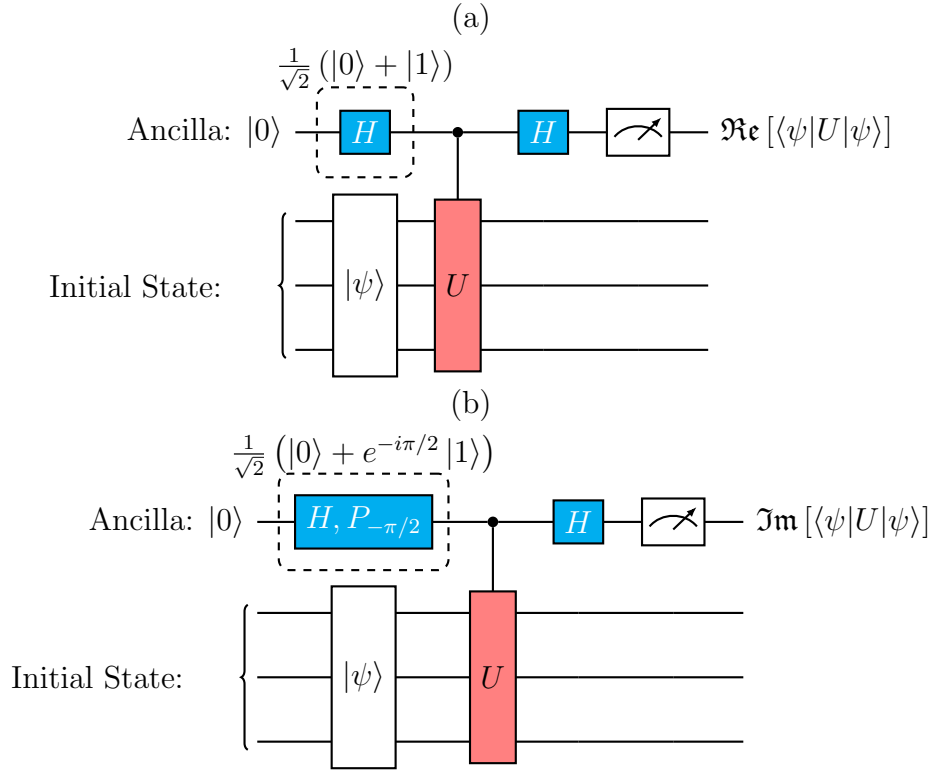


Figure 10: Circuits for computations of the real (a) and imaginary (b) parts of $\langle \psi | U | \psi \rangle$ according to the Hadamard test.

through the circuit shown in Fig. 10 panel (a). Note that after application of the Hadamard gate to the first qubit, we obtain the following state:

$$H|0\rangle \otimes |\Psi\rangle = \frac{1}{\sqrt{2}} (|0\rangle \otimes |\Psi\rangle + |1\rangle \otimes |\Psi\rangle) \quad (23)$$

Applying the controlled unitary, applies U to $|\psi\rangle$ when the ancilla is $|1\rangle$, as follows:

$$CU(H|0\rangle \otimes |\Psi\rangle) = \frac{1}{\sqrt{2}} (|0\rangle \otimes |\Psi\rangle + |1\rangle \otimes U|\Psi\rangle), \quad (24)$$

and applying the second Hadamard gate, we obtain $|\Phi\rangle = HCU(H|0\rangle \otimes |\Psi\rangle)$, where

$$|\Phi\rangle = \frac{1}{\sqrt{2}} (H|0\rangle \otimes |\Psi\rangle + H|1\rangle \otimes U|\Psi\rangle) \quad (25)$$

$$= \frac{1}{2} ((|0\rangle + |1\rangle) \otimes |\Psi\rangle + (|0\rangle - |1\rangle) \otimes U|\Psi\rangle) \quad (26)$$

$$= \frac{1}{2} (|0\rangle \otimes (\mathbb{I} + U)|\Psi\rangle + |1\rangle \otimes (\mathbb{I} - U)|\Psi\rangle) \quad (27)$$

It should now be clear that a measurement of the ancilla with σ_z gives the $\text{Re}[\langle\Psi|U|\Psi\rangle]$, as follows:

$$\begin{aligned} \langle\Phi|\sigma_z \otimes \mathbb{I}|\Phi\rangle &= \frac{1}{4} (\langle\Psi|(\mathbb{I} + U^\dagger) \otimes \langle 0| + \langle\Psi|(\mathbb{I} - U^\dagger) \otimes \langle 1|) \sigma_z \otimes (|0\rangle \otimes (\mathbb{I} + U)|\Psi\rangle + |1\rangle \otimes (\mathbb{I} - U)|\Psi\rangle) \\ &= \frac{1}{4} (\langle\Psi|(\mathbb{I} + U^\dagger) \otimes \langle 0|\sigma_z|0\rangle \otimes (\mathbb{I} + U)|\Psi\rangle + \langle\Psi|(\mathbb{I} + U^\dagger) \otimes \langle 0|\sigma_z|1\rangle \otimes (\mathbb{I} - U)|\Psi\rangle \\ &\quad + \langle\Psi|(\mathbb{I} - U^\dagger) \otimes \langle 1|\sigma_z|0\rangle \otimes (\mathbb{I} + U)|\Psi\rangle + \langle\Psi|(\mathbb{I} - U^\dagger) \otimes \langle 1|\sigma_z|1\rangle \otimes (\mathbb{I} - U)|\Psi\rangle) \\ &= \frac{1}{4} (\langle\Psi|(\mathbb{I} + U^\dagger)(\mathbb{I} + U)|\Psi\rangle + \langle\Psi|(\mathbb{I} - U^\dagger)(\mathbb{I} - U)|\Psi\rangle) \\ &= \frac{1}{4} (\langle\Psi|\Psi\rangle + \langle\Psi|U|\Psi\rangle + \langle\Psi|U^\dagger|\Psi\rangle + \langle\Psi|U^\dagger U|\Psi\rangle \\ &\quad - \langle\Psi|\Psi\rangle + \langle\Psi|U|\Psi\rangle + \langle\Psi|U^\dagger|\Psi\rangle - \langle\Psi|U^\dagger U|\Psi\rangle) \\ &= \frac{1}{2} (\langle\Psi|U|\Psi\rangle + \langle\Psi|U^\dagger|\Psi\rangle) \\ &= \frac{1}{2} (\langle\Psi|U|\Psi\rangle + \langle\Psi|U|\Psi\rangle^\dagger) \\ &= \text{Re}[\langle\Psi|U|\Psi\rangle] \end{aligned}$$

Analogously, we can show that a measurement of the ancilla with σ_z for the circuit of panel (b) gives the $\text{Im}[\langle\Psi|U|\Psi\rangle]$. Note that the only difference between the two circuits is that in panel (b) the first Hadamard is followed by the phase shift $P_{-\pi/2} = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\pi/2} \end{bmatrix}$.

4.5.1 Hadamard Test Function

The Hadamard test circuit can be constructed by setting up the ancilla preparation, wavefunction initialization and including the controlled unitaries corresponding to the expectation

values to be calculated. Measurement of the ancilla is included to obtain the counts and calculate the real and imaginary components of the expectation value.

Script 4.35: Hadamard Test Function



```

1 def get_hadamard_test(num_q, initial_state, control_operation,
2                       control_repeats=0, imag_expectation=False):
3     # Circuit object framework
4     qr_hadamard = QuantumRegister(num_q+1)
5     cr_hadamard = ClassicalRegister(1)
6     qc_hadamard = QuantumCircuit(qr_hadamard, cr_hadamard) # instantiated here
7     # Initialization of calculation qubits
8     qc_hadamard.append(initial_state, qr_hadamard[1:]) # initial psi
9     qc_hadamard.barrier()
10    # Hadamard test structure
11    qc_hadamard.h(0)
12    if imag_expectation:
13        qc_hadamard.p(-np.pi/2, 0) # qc_hadamard.s(0).inverse() may be equivalent
14    # iterates over the number of times the control operation should be added
15    for i in range(control_repeats):
16        qc_hadamard.append(control_operation, qr_hadamard[:])
17        qc_hadamard.h(0)
18        qc_hadamard.barrier()
19    # Measuring the ancilla
20    qc_hadamard.measure(0,0)
21    return qc_hadamard

```

4.5.2 Processing the Hadamard Test Results

The result of a quantum experiment is typically performed in the σ_z basis, yielding either the state $|0\rangle$ or $|1\rangle$ for each qubit. Each of these eigenstates have the following eigenvalues.

$$\langle 0 | \sigma_z | 0 \rangle = 1$$

$$\langle 1 | \sigma_z | 1 \rangle = -1$$

Thus we account for the number of measurements of the ancilla qubit in the $|0\rangle$ and $|1\rangle$ states and obtain the average values associated with the measurement:

$$\begin{aligned} \langle \psi | U | \psi \rangle &\rightarrow \frac{\langle 0 | \sigma_z | 0 \rangle N_{|0\rangle} + \langle 1 | \sigma_z | 1 \rangle N_{|1\rangle}}{N_{|0\rangle} + N_{|1\rangle}} \\ &= \frac{N_{|0\rangle} - N_{|1\rangle}}{N_{|0\rangle} + N_{|1\rangle}} \end{aligned}$$

This yields the corresponding expectation value of the unitary operator U .

Script 4.36: Hadamard Test Post-Processing



```

1 def get_spin_correlation(counts):
2     qubit_to_spin_map = {
3         '0': 1,
4         '1': -1,
5     }
6     total_counts = 0
7     values_list = []
8     for k,v in counts.items():
9         values_list.append(qubit_to_spin_map[k] * v)
10        total_counts += v
11    average_spin = (sum(values_list)) / total_counts
12    return average_spin

```

4.5.3 How to execute the Hadamard test for our operator?

Using the `time_evo_op` for a small time-step, we generate the controlled unitary,

Script 4.37: Creation of Controlled Time-Evolution Operator



```

1 controlled_time_evo_op = time_evo_op.control()

```

We then execute Hadamard test for all times by propagating from the initial state at time zero using the controlled unitary, and computing the real and imaginary components of the expectation value using the code in the subsequent code cell.

Script 4.38: Hadamard Test Execution



```

1 # IMPORTANT: Use qasm_simulator to obtain meaningful statistics
2 # statevector is not appropriate for this method
3 simulator = BasicAer.get_backend('qasm_simulator')
4 real_amp_list = []
5 imag_amp_list = []
6 for idx,time in enumerate(time_range):
7     print(f'Running dynamics step {idx}')
8     # Real component -----
9     qc_had_real = get_hadamard_test(num_q, init_circ,
10                                     controlled_time_evo_op,
11                                     control_repeats=idx,
12                                     imag_expectation=False)
13     had_real_counts = get_circuit_execution_counts(
14         qc_had_real, simulator, n_shots=num_shots)
15     real_amplitude = get_spin_correlation(had_real_counts)
16     real_amp_list.append(real_amplitude)
17
18     # Imag component -----
19     qc_had_imag = get_hadamard_test(num_q, init_circ,
20                                     controlled_time_evo_op,
21                                     control_repeats=idx,
22                                     imag_expectation=True)
23     had_imag_counts = get_circuit_execution_counts(
24         qc_had_imag, simulator, n_shots=num_shots)
25     imag_amplitude = get_spin_correlation(had_imag_counts)
26     imag_amp_list.append(imag_amplitude)
27     print(f'Finished step {idx}, where '
28           f'Re = {real_amplitude:.3f} '
29           f'Im = {imag_amplitude:.3f}')
30
31 real_amp_array = np.array(real_amp_list)
32 imag_amp_array = np.array(imag_amp_list)
33
34 # plotting the data
35 plt.plot(time_range, np_abs_correlation_with_hadamard_test,
36           '.', label='Hadamard Test')
37 plt.xlabel('Time')
38 plt.ylabel('Absolute Value of Survival Amplitude')
39 plt.legend()
40 plt.show()

```

We verify that the Hadamard test data with the given number of shots agrees with both the classical benchmark and the statevector results (figure 11). We note that increasing the number of shots would further refine the agreement at the expense of additional execution time.

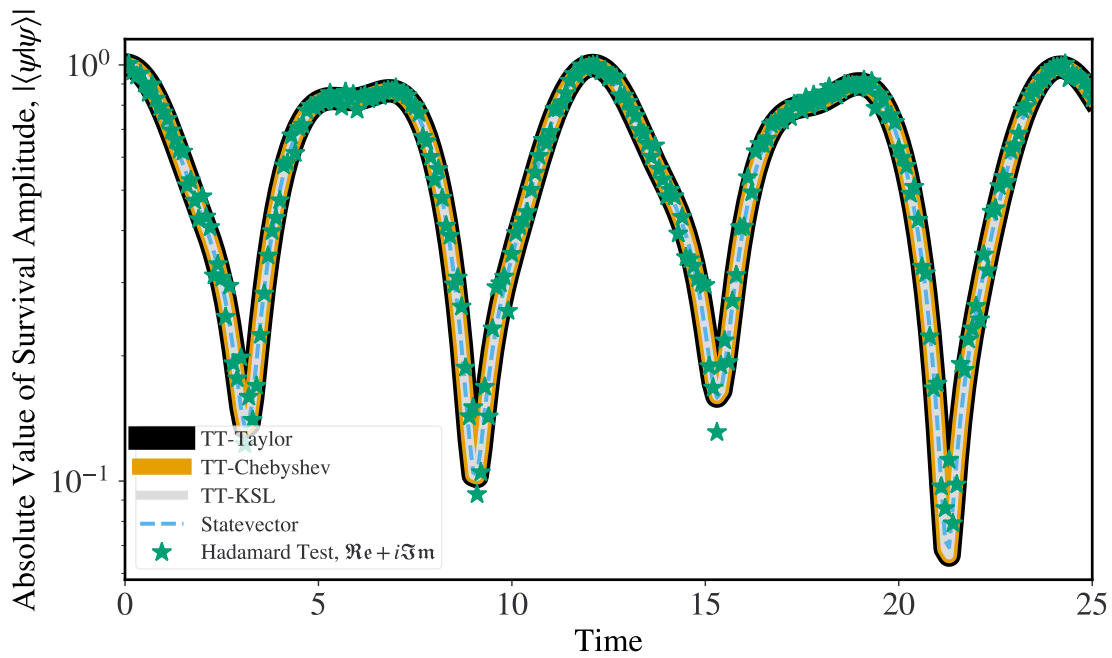


Figure 11: Results of simulating dynamics with the Hadamard test (stars) on a quantum simulator as compared to the statevector simulation (dashed line) and classical benchmarks (continuous lines).

4.6 Variational Quantum Real Time Evolution

Simulating Trotterization for long time scales requires deep quantum circuits which are likely to be noisy when run on near term quantum computers. Therefore, noise resilient hybrid alternatives are proposed. One such alternative algorithm is the Variation Quantum Real Time Evolution (VarQRTE) algorithm.

VarQRTE approximates the time evolution of a system by restricting it to a subspace of the total Hilbert space. The accessible subspace is determined by the ansatz $U(\theta)$, indicating that a proper ansatz for a system must be chosen carefully. The evolution of the input state is approximated through calculating $\dot{\theta}(t)$, and this value is used to determine the parameters at the next timestep $\theta(t + dt) = \theta(t) + \dot{\theta}(t)dt$. The change in parameters for each timestep $\dot{\theta}(t)$ are calculated through measuring the Pauli strings of the Hamiltonian and the generators of the anstaz as derived below. Once the parameters are determined for a given timestep, the state can be obtained through application of the ansatz to the starting state, $U(\theta(t)) |\psi(0)\rangle \approx |\psi(t)\rangle$.

The change of the parameters $\dot{\theta}(t)$, is determined by applying McLachlan's variational principle to the Schrödinger equation.

$$\partial \left\| \left(\frac{\partial}{\partial t} + i\mathcal{H} \right) |\psi(\theta(t))\rangle \right\| = 0 \quad (28)$$

When varying over an ansatz $U(\theta(t)) |\psi(0)\rangle = \psi(\theta(t))$, this reduces to a system of linear equations,

$$\sum_j A_{ij} \dot{\theta}_j = C_i \quad (29)$$

With the matrix A_{ij} and vector C_i defined as such,⁷

$$A_{ij} = \text{Re} \left(\frac{\partial \langle \psi(\theta(t)) |}{\partial \theta_i} \frac{\partial |\psi(\theta(t))\rangle}{\partial \theta_j} \right) \quad (30)$$

$$C_i = -\text{Im}\left(\frac{\partial \langle \psi(\theta(t)) |}{\partial \theta_i} \mathcal{H} | \psi(\theta(t)) \rangle\right) \quad (31)$$

Taking derivatives with respect to the parameters of the ansatz returns the generators G of the ansatz, $\frac{\partial \langle \psi(\theta(t)) |}{\partial \theta_i} = -iG | \psi(\theta(t)) \rangle$. For an example, for a single qubit Z-rotation, $\frac{\partial}{\partial \theta} e^{-i\theta\sigma_z} | \psi \rangle = -i\sigma_z e^{-i\theta\sigma_z} | \psi \rangle$. Therefore, evaluating these terms in A_{ij} and C_i is reduced to performing a Hadamard test on the generators of the ansatz.

Script 4.39: Variation Quantum Real Time Evolution



```

1 import numpy as np
2 from qiskit_algorithms import VarQRTE, TimeEvolutionProblem
3 from qiskit.circuit.library import ExcitationPreserving
4 from qiskit_algorithms.time_evolvers.variational import RealMcLachlanPrinciple
5 from qiskit import QuantumCircuit
6 from qiskit.primitives import Estimator
7 from qiskit.quantum_info import SparsePauliOp
8
9 var_principle = RealMcLachlanPrinciple()
10 estimator = Estimator(options={"shots": 1024})
11 total_time = 5.0
12 evolution_timestep = 0.2
13
14 hamiltonian = SparsePauliOp.from_list([("ZI", 0.5), ("IZ", 0.5), ("XX", 0.2)])
15
16 def init_circ():
17     qc = QuantumCircuit(2,0)
18     qc.x(0) # initial state is |10>
19     return qc
20
21 # Ansatz at t=0 must be equal to the Identity
22 params = [np.array([0.0 for i in range(5)])]
23 anstaz = ExcitationPreserving(num_qubits=2, entanglement='linear', reps=1)
24 anstaz = init_circ().compose(anstaz)
25
26 # Define and Run the Time Evolution Problem
27 evolution_problem = TimeEvolutionProblem(hamiltonian, total_time)
28 qrte = VarQRTE(anstaz, params[0][:], variational_principle=var_principle,
29     ↪ estimator=estimator, num_timesteps=int(total_time/evolution_timestep))
29 params = qrte.evolve(evolution_problem).parameter_values
30
31 # Assemble the circuit which creates the evolved state
32 evolved_circ = anstaz.assign_parameters(params[-1])
33 print(evolved_circ)

```


4.7 Variational Quantum Eigensolver

The Variational Quantum Eigensolver (VQE) algorithm is a near term quantum algorithm, used to determine the ground state energy of a given system.^{8,9} VQE is a hybrid algorithm, meaning that the quantum algorithm is able to utilize a classical computer for part of the calculation. This helps to make VQE noise resilient, as it can offload some computation to noiseless classical computers, while still utilizing the exponentially scaling Hilbert space of a quantum computer.

The VQE algorithm is based on the variational principle, which states that the expectation value of the Hamiltonian \mathcal{H} for any state $|\psi\rangle$ must be greater than the ground state energy.

$$E_0 \leq \frac{\langle\psi|\mathcal{H}|\psi\rangle}{\langle\psi|\psi\rangle} \quad (32)$$

VQE leverages this fact by parameterizing an input wavefunction $|\psi(\theta)\rangle$. The algorithm then measures the expectation value of the Pauli strings of the Hamiltonian for this state, and sends this result to a classical computer. The classical computer then uses these measurements to determine the energy of the state, and finally the classical computer uses an optimizer such as Constrained Optimization by Linear Approximation (COBYLA) or Simultaneous Perturbation Stochastic Approximation (SPSA) algorithms to tune the parameters θ , in order to minimize the energy.

Once the optimizers converge on the final parameters θ_f , an estimate of the ground state energy $\frac{\langle\psi(\theta_f)|\mathcal{H}|\psi(\theta_f)\rangle}{\langle\psi(\theta_f)|\psi(\theta_f)\rangle} \approx E_0$ and the the ground state wavefunction $|\psi(\theta_f)\rangle \approx |\psi_0\rangle$ are obtained.

The parameterization of the wavefunction $|\psi(\theta)\rangle$ is done using an circuit which is called the ansatz. The decision of which ansatz to use is important, and can greatly affects the results obtained by this algorithm. For an example, by identifying symmetries of the Hamiltonian and using an ansatz which also obeys these symmetries, the search space of VQE can

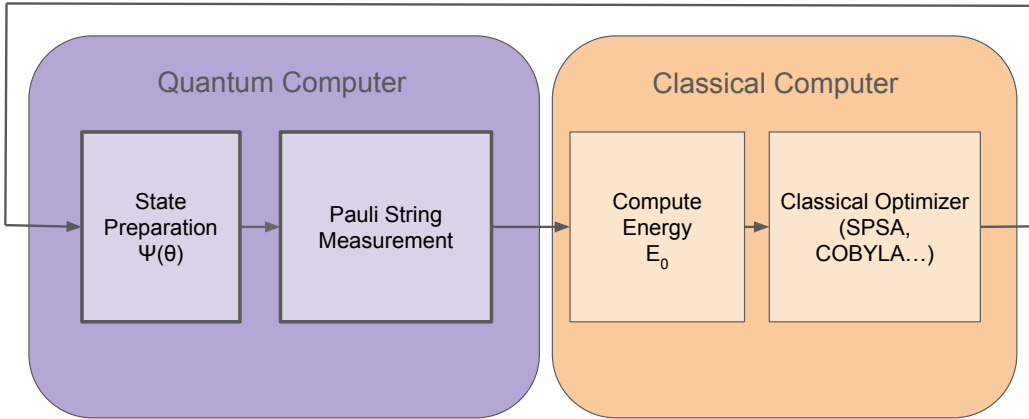


Figure 12: A box diagram of the Variational Quantum Eigensolver Algorithm.

be greatly reduced. This reduction in the search space allows for the classical optimizers to more efficiently find the ground state energy.

Below is an example of the Variational Quantum Eigensolver algorithm applied to the Hamiltonian, $\mathcal{H} = .5\sigma_{z_1} + .5\sigma_{z_2} + .2\sigma_{x_1}\sigma_{x_2}$. This Hamiltonian is defined using SparsePauliOp from QISKIT, the ansatz used is the EfficientSU2 ansatz, and the energy is calculated through using the Estimator primitive. Lastly, the ground state energy is found by using the COBYLA algorithm to minimize the energy.

Script 4.40: Variational Quantum Eigensolver



```
1 import numpy as np
2 from qiskit.circuit.library import EfficientSU2
3 from qiskit.quantum_info import SparsePauliOp
4 from qiskit.primitives import Estimator
5 from scipy.optimize import minimize
6
7 hamiltonian = SparsePauliOp.from_list([("ZI", 0.5), ("IZ", 0.5), ("XX", 0.2)])
8
9 estimator = Estimator(options={"shots": 1024})
10
11 #Estimate the energy using a quantum computer
12 def energy(params, ansatz, hamiltonian, estimator):
13     current_energy = estimator.run(ansatz, hamiltonian, params).result().values[0]
14     print("Energy: "+str(current_energy))
15     return current_energy
16
17 # Define the ansatz which will be used to prepare the ground state
18 ansatz = EfficientSU2(hamiltonian.num_qubits)
19
20 initial_params = np.random.random(ansatz.num_parameters)
21
22 # Now run the minimization algorithm to minimize the energy
23 res = minimize(energy, initial_params,
24               args=(ansatz, hamiltonian, estimator), method="cobyla")
25
26 # Lastly, output the energy estimate obtained
27 final_params = res.x
28 print("Final Energy Estimate: "+str(energy(final_params, ansatz, hamiltonian,
↪ estimator)))
```

5 Qumode-Based Simulations

Qumode-based quantum computers, which store information in continuous quantum modes like light, offer a unique approach to quantum computation, complementing the traditional qubit-based method. We remind ourselves that any quantum state with qubits as the basic unit of quantum information is uniquely characterized by a superposition of the $|0\rangle$ and $|1\rangle$ state. Extending this concept to a d -dimensional complex Hilbert space yields a *qudit*, where

any quantum state $|\phi\rangle$ admits the representation

$$|\phi\rangle = \sum_{i=0}^{d-1} \phi_i |i\rangle \quad (33)$$

where the set $\{|0\rangle, |1\rangle, \dots, |d-1\rangle\}$ constitutes an orthonormal basis for that d -dimensional Hilbert space and $\{\phi_i\}$ are the corresponding expansion coefficients. However, many physical systems such as light are intrinsically continuous, and thus a *continuum* orthonormal basis residing in an infinite-dimensional Hilbert space offers the continuous-variable model that fits the simulation of, for instance, bosonic systems. Naturally, we extend Equation 33 to the representation of a *qumode* as

$$|\psi\rangle = \int dx \psi(x) |x\rangle \quad (34)$$

where the states $|x\rangle$ span over the real line.

Interestingly, qumodes and qudits are related since qudits can be thought of as energy levels of a harmonic oscillator

$$|i\rangle = \int dx |x\rangle \langle x|i\rangle = \int dx \psi_i(x) |x\rangle, \quad (35)$$

where $\psi_i(x)$ are the Hermite polynomials.

Continuous-variable systems such as the bosonic harmonic oscillator are defined by the canonical mode creation and annihilation operators \hat{a}^\dagger, \hat{a} satisfying $[\hat{a}, \hat{a}^\dagger] = I$. Alternatively, one may work with quadrature operators

$$\begin{cases} \hat{x} = \sqrt{\frac{\hbar}{2}} (\hat{a} + \hat{a}^\dagger) \\ \hat{p} = -i\sqrt{\frac{\hbar}{2}} (\hat{a} - \hat{a}^\dagger) \end{cases} \quad (36)$$

satisfying $[\hat{x}, \hat{p}] = i\hbar$. We note that the basis states $|x\rangle$ in Equation 34 are also eigenstates

of the \hat{x} quadrature.

Starting from the vacuum state $|0\rangle$, we can perform propagation by evolving the vacuum state as

$$|\psi\rangle = \exp(-itH) |0\rangle \quad (37)$$

where H is the Hamiltonian operator written in bosonic operators and t is the evolution time. States whose Hamiltonians are at most quadratic in the quadratures \hat{x}, \hat{p} are called *Gaussian*, parametrized by a complex displacement operator α and a complex squeezing parameter z that corresponds to a Gaussian distribution. Complementary to these continuous Gaussian states are the Fock states, $|n\rangle$ for $n \in \mathbb{N}$, which are eigenstates of the number operator $\hat{n} = \hat{a}^\dagger \hat{a}$. In general, any n -mode Hamiltonian H generates a unitary operation

$$U = \exp(-it\hat{H}) \quad (38)$$

that is implementable via a sequence of gates from a universal gate set (each of which acting on one or two qumodes), which we now discuss.

A continuous variable quantum computer is *universal* if it can propagate any Hamiltonian with arbitrarily small error.¹⁰ Any universal gate set contains Gaussian gates (gates that are at most quadratic in the mode operators) and non-Gaussian gates (gates with third degree or higher), similar to the Clifford group and non-Clifford group of gates from the qubit model. Fundamental continuous-variable gates are described in Table 2, according to the convention adopted by Strawberry Fields, an open-source framework for photonic quantum computing.^{11,12} All the described gates are Gaussian except for the cubic phase and kerr gates. LLyod et al.¹³ provides the necessary and sufficient conditions for a universal gate set over continuous variables, and it has been shown that the sets $\{D_i(\alpha), R_i(\phi), S_i(z), BS_{ij}(\theta, \phi)\}$ ¹⁰ or $\{\mathcal{F}, Z_i(p), P_i(s)\}$ ¹⁴ covers all Gaussian operations. Adding the cubic phase gate $V_i(\gamma)$ to either set sufficiently allows non-Gaussian operations.

Continuous variable measurements are also categorized into Gaussian (homodyne and

Table 2: Selected continuous-variable gates

Gate	Unitary operation
Displacement	$D_i(\alpha) = \exp\left(\alpha\hat{a}_i^\dagger - \alpha^*\hat{a}_i\right)$
Rotation	$R_i(\phi) = \exp(i\phi\hat{n}_i)$
Squeezing	$S_i(z) = \exp\left(\frac{1}{2}\left(z^*\hat{a}_i^2 - z\hat{a}_i^{\dagger 2}\right)\right)$
Beamsplitter	$BS_{ij}(\theta, \phi) = \exp\left(\theta\left(\exp\{i\phi\}\hat{a}_i^\dagger\hat{a}_j - \exp\{-i\phi\}\hat{a}_i\hat{a}_j^\dagger\right)\right)$
Two-mode Squeezing	$S2_{ij}(z) = \exp\left(z\hat{a}_1^\dagger\hat{a}_2^\dagger - z^*\hat{a}_1\hat{a}_2\right)$
Position Displacement	$X_i(x) = D_i\left(\frac{x}{\sqrt{2}}\right) = \exp\left(-i\frac{x}{\sqrt{2}}\hat{p}\right)$ with $x \in \mathbb{R}$
Momentum Displacement	$Z_i(p) = D_i\left(i\frac{p}{\sqrt{2}}\right) = \exp\left(i\frac{p}{\sqrt{2}}\hat{x}\right)$
Fourier	$\mathcal{F}_i = R_i\left(\frac{\pi}{2}\right) = \exp\left(i\frac{\pi}{2}\hat{n}\right)$
Quadratic Phase	$P_i(s) = \exp(is\hat{x}^2)$
Cubic Phase	$V_i(\gamma) = \exp\left(i\frac{\gamma}{6}\hat{x}_i^3\right)$
Kerr	$K_i(\kappa) = \exp(i\kappa\hat{n}^2)$

heterodyne) and non-Gaussian (photon counting). A homodyne detection projects the quantum state onto the eigenstates of the Hermitian operator

$$\hat{x}_\phi = (\cos \phi)\hat{x} + (\sin \phi)\hat{p} \quad (39)$$

A heterodyne measurement projects the state onto the operator

$$\frac{1}{\pi}|\alpha\rangle\langle\alpha| \quad (40)$$

where each coherent state is defined by the displaced vacuum state, $|\alpha\rangle = D(\alpha)|0\rangle$, which is the eigenstate of the annihilation operator ($\hat{a}|\alpha\rangle = \alpha|\alpha\rangle$) and admits the Fock basis decomposition

$$|\alpha\rangle = e^{-|\alpha|^2/2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle \quad (41)$$

Both homodyne and heterodyne measurements of one qumode preserves the Gaussian characteristic of the remaining ones. Complementary to these continuous (or wave-like) measurements is the particle-like photon counting, in which the quantum state is projected onto the number eigenstates $|n\rangle$, returning non-negative integer values.

5.1 Hamiltonian propagation

Quantum computers are suited to simulate dynamics of not only quantum systems but also atoms, molecules and biochemical systems. Examples include determining the ground state energy of large systems, dynamics of an ensemble of molecules, and protein folding pathways, all of which are computationally taxing with classical computations.^{15,16} Generally speaking, given an arbitrary Hamiltonian operator for a physical system, we first map it into an n -mode bosonic second quantization form (possibly via bosonization or Jordan-Schwinger transformation)

$$H = f\left(\hat{a}_1, \hat{a}_1^\dagger, \hat{a}_2, \hat{a}_2^\dagger, \dots, \hat{a}_n, \hat{a}_n^\dagger\right) \quad (42)$$

which is then decomposed into

$$H = \sum_{j=1}^N H_j \quad (43)$$

where each of the corresponding term in the time-evolution operator $\exp(-iH_j t)$ can be written as a continuous-variable gate. Such a decomposition is guaranteed, so long as a universal gate set is used. Interested readers are invited to read more on a systematic decomposition method.^{14,17} While an exact decomposition is possible and may at times significantly lower the gate count needed (given a precision cutoff), we focused on approximate decompositions which are based on the Lie-Trotter product formula

$$e^{-iHt} = \left(\prod_{j=1}^N e^{-i\frac{H_j}{k}t}\right)^k + \mathcal{O}\left(\frac{N^2 t^2 \left(\max_{1 \leq j \leq N} |H_j|\right)^2}{k}\right) \quad (44)$$

with k Trotter steps, and the error term scales quadratically with the evolution time t .

For the rest of this subsection, we demonstrate how to simulate the two-mode Bose-Hubbard Hamiltonian using Strawberry Fields. Written down in the early 1960's, the Bose-Hubbard model was initially applied to understanding the behavior of transition metal monoxides (FeO, NiO, CoO), which are antiferromagnetic insulators instead of metallic as

predicted previously.¹⁸ It offers one simple way to gain insights into how electronic interactions may result in insulating, magnetic, and even novel superconducting effects in a solid. For this example, we consider a lattice with two adjacent nodes with adjacency matrix $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, then the Bose-Hubbard Hamiltonian with on-site interactions is given by

$$H = J \left(\hat{a}_1^\dagger \hat{a}_2 + \hat{a}_2^\dagger \hat{a}_1 \right) + \frac{1}{2} U \left(\hat{n}_1^2 - \hat{n}_1 + \hat{n}_2^2 - \hat{n}_2 \right) \quad (45)$$

with J being the transfer integral (hopping term) of the boson between nodes and U the on-site interaction potential. We can now write

$$\begin{aligned} e^{-iHt} &= \left[e^{-i\frac{Jt}{k}(\hat{a}_1^\dagger \hat{a}_2 + \hat{a}_2^\dagger \hat{a}_1)} e^{-i\frac{Ut}{2k}\hat{n}_1^2} e^{-i\frac{Ut}{2k}\hat{n}_2^2} e^{i\frac{Ut}{2k}\hat{n}_1} e^{i\frac{Ut}{2k}\hat{n}_2} \right]^k + O\left(\frac{t^2}{k}\right) \\ &= [BS_{1,2}(\theta, \phi) (K_1(r)R_1(-r) \otimes K_2(r)R_2(-r))]^k + O\left(\frac{t^2}{k}\right) \end{aligned} \quad (46)$$

with $\theta = -\frac{Jt}{k}$, $\phi = \frac{\pi}{2}$, $r = -\frac{Ut}{2k}$. Schematically, we obtain Figure 13, where two layers of Trotter steps are shown.

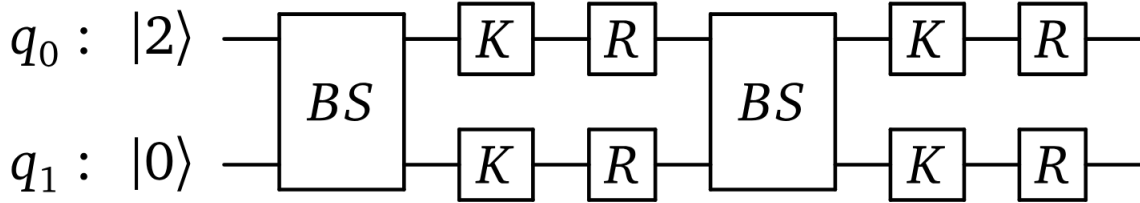


Figure 13: Two layers of Bose-Hubbard Hamiltonian evolution

To start off on Python, we import the relevant package and set up the global parameters that apply to the rest of this subsection in Script 5.1

Script 5.1: Bose Hubbard Hamiltonian Global Parameters



```
1 import numpy as np
2 cutoff = 10 # Number of levels per bosonic mode (Fock dimension truncation)
3 J = 1 # Hopping term
4 U = 1.5 # On-site interaction potential
5 k = 20 # Number of Trotter layers
6 t = 1.086 # Evolution time
7 theta = -J*t/k
8 r = -U*t/(2*k)
9 # Prepare the initial statevector
10 ket = np.zeros([10]*2, dtype=np.complex64)
11 ket[2,0] = 1.0 + 0.0j
```

To prepare a known photonic quantum architecture in Strawberry Fields, the following steps are carried out in Script 5.1:

- Declare a Strawberry Fields program with the required number of qumodes.
- Define the quantum circuit with operations in the class `sf.ops`.
- Specify an engine backend. For this example, we use the Fock basis backend (since the Kerr gate is non-Gaussian, we can not use the `gaussian` or `bosonic` backend here).
- Run engine, and extract the resulting quantum state from `results.state`.

Script 5.2: Evolving Bose-Hubbard Hamiltonian



```

1 import strawberryfields as sf
2 # Declare Program
3 prog = sf.Program(2)
4 # Define quantum circuit
5 with prog.context as q:
6     # Prepare initial quantum state
7     sf.ops.Ket(ket) | q
8     # For-loop to define each of the $k$ Trotter layers
9     for i in range(k):
10        sf.ops.BSgate(theta, np.pi/2) | (q[0],q[1]) # Parametrized beamsplitter gate
11        ↪ applied on the first and second qumode
12        sf.ops.Kgate(r) | q[0] # Parametrized kerr gate applied on the first qumode
13        sf.ops.Rgate(-r) | q[0] # Parametrized rotation gate applied on the first qumode
14        sf.ops.Kgate(r) | q[1]
15        sf.ops.Rgate(-r) | q[1]
16        # end circuit
17
18 # Define the engine backend and execute the circuit
19 eng = sf.Engine('fock', backend_options={"cutoff_dim":cutoff})
20 state = eng.run(prog).state
21
22 # Print the output state probabilities
23 print(state.fock_prob([0,2]))
24 print(state.fock_prob([1,1]))
25 print(state.fock_prob([2,0]))

```

The resulting quantum state yield the following output Fock probabilities

$$\mathbb{P}(|0, 2\rangle) = 0.5224012457200212$$

$$\mathbb{P}(|1, 1\rangle) = 0.23565287685672495$$

$$\mathbb{P}(|2, 0\rangle) = 0.24194587742326018$$

which add up to 1, indicating the particle-preserving nature of this Hamiltonian simulation.

Knowing the output statevector $|\psi\rangle$, we can compute the energy

$$E = \frac{\langle \psi | H | \psi \rangle}{\langle \psi | \psi \rangle} \quad (47)$$

by looking at the expectation value of the Hamiltonian $H = \hat{J} + \frac{1}{2}\hat{U}$ where $\hat{J} =$

$J(\hat{a}_1^\dagger + \hat{a}_2 + \hat{a}_2^\dagger \hat{a}_1)$ and $\hat{U} = U(\hat{n}_1^2 - \hat{n}_1 + \hat{n}_2^2 - \hat{n}_2)$. Now, using the notation $|\psi\rangle = \sum_{i=1}^{10} \sum_{j=1}^{10} c_{i,j} |i, j\rangle$, we analytically obtain

$$\begin{aligned} \langle \psi | \hat{a}_1^\dagger \hat{a}_2 | \psi \rangle &= \left(\sum_{i=1}^{10} \sum_{j=1}^{10} c_{i,j}^* \langle i, j | \right) \left(\sum_{i'=1}^{10} \sum_{j'=1}^{10} c_{i',j'} \sqrt{j'(i'+1)} |i'+1, j'-1\rangle \right) \\ &= \sum_{i=1}^{10} \sum_{j=1}^{10} c_{i,j}^* \sqrt{i(j+1)} c_{i-1, j+1} \end{aligned} \quad (48)$$

Applying the same process for the operator $\hat{a}_2^\dagger \hat{a}_1$ and combining with Equation 48 yields

$$\langle \psi | \hat{J} | \psi \rangle = J \sum_{i=1}^{10} \sum_{j=1}^{10} c_{i,j}^* \left(\sqrt{i(j+1)} c_{i-1, j+1} + \sqrt{j(i+1)} c_{i+1, j-1} \right) \quad (49)$$

On the other hand,

$$\langle \psi | \hat{n}_1 | \psi \rangle = \left(\sum_{i=1}^{10} \sum_{j=1}^{10} c_{i,j}^* \langle i, j | \right) \left(\sum_{i'=1}^{10} \sum_{j'=1}^{10} c_{i',j'} i |i', j'\rangle \right) = \sum_{i=1}^{10} \sum_{j=1}^{10} i |c_{i,j}|^2 \quad (50)$$

and similarly,

$$\langle \psi | \hat{n}_1^2 | \psi \rangle = \sum_{i=1}^{10} \sum_{j=1}^{10} i^2 |c_{i,j}|^2 \quad (51)$$

Again, applying the same process for the operators \hat{n}_2^2 and \hat{n}_2 implies

$$\langle \psi | \frac{1}{2} \hat{U} | \psi \rangle = \frac{U}{2} \sum_{i=1}^{10} \sum_{j=1}^{10} |c_{i,j}|^2 (i^2 + j^2 - i - j) \quad (52)$$

Without further simplification, we naively define the respective functions `hopping_term` and `on_site` in script 5.1 to calculate the energy of the resulting quantum state, which turns out to be 2.135642775045301.

Script 5.3: Calculating Energy of evolved state



```
1 import numpy as np
2 cutoff = 10 # Fock dimension truncation
3
4 def hopping_term(statevec):
5     svc = np.conj(statevec)
6     hop = 0
7     for i in range(cutoff):
8         if i == 0:
9             for j in range(1,cutoff):
10                hop += (svc[i,j]*np.sqrt((i+1)*j)*statevec[i+1,j-1])
11        else:
12            for j in range(cutoff-1):
13                hop += (svc[i,j]*np.sqrt(i*(j+1))*statevec[i-1,j+1])
14            if i < cutoff-1:
15                for x in range(1,cutoff):
16                    hop += (svc[i,x]*np.sqrt((i+1)*x)*statevec[i+1,x-1])
17    return np.abs(hop*J)
18
19 def on_site(statevec):
20     on_site = 0
21     for i in range(cutoff):
22         for j in range(cutoff):
23             on_site += np.abs(statevec[i,j])*(i**2+j**2-i-j)
24    return np.abs(on_site*U/2)
25
26 sket = state.ket() # Extract resulting statevector
27 inner_prod = np.abs(np.vdot(sket,sket)) # Calculate statevector's norm
28 energy = (hopping_term(sket)+on_site(sket))/inner_prod # Calculate total energy
29 print(energy)
```

5.2 Variational Quantum Eigensolver

First introduced in 2014,¹⁹ the variational quantum eigensolver (VQE) is a hybrid algorithm that utilizes both quantum and classical computers to find the lowest energy eigenstate (or ground state) and some excited states of a physical system, such as a molecule. Provided a guessed quantum circuit, or *ansatz*, the quantum processor computes the expectation value of the system with respect to an observable, such as the Hamiltonian, which is fed to a classical optimizer to improve the guess. This algorithm is justified by the variational principle of

quantum mechanics which states that, for any normalizable quantum state $|\psi\rangle$ then

$$\frac{\langle\psi|H|\psi\rangle}{\langle\psi|\psi\rangle} \geq E_0 \quad (53)$$

where E_0 is the ground state of the Hamiltonian H , and equality is attained if and only if $|\psi\rangle$ is indeed the system's lowest energy eigenstate. The Python library Strawberry Fields, used in this tutorial, supports the Tensorflow backend which allows better optimization algorithms, other machine learning tools, and possible GPU utilization. With the essential steps for running a quantum circuit in Strawberry Fields is explained in Section 5.1, we only focus on setting up VQE in this subsection. Script 5.2 shows the necessary packages, dictates the global variables, and defines the functions needed for later visualization.

Script 5.4: Setting up for performing VQE



```
1 import numpy as np
2 import strawberryfields as sf
3 import tensorflow as tf
4 from matplotlib import pyplot as plt # for visualization
5
6 lr = 0.1 # Learning rate for TensorFlow optimizers
7 active_std = 0.001 # Standard deviations for normal distributions
8 passive_std = 0.1
9 tf.random.set_seed(42) # Global seed to ensure reproducibility over runs
```

We now first demonstrate how to optimize the parameters of a displacement gate so that a desired coherent state is attained. To remind ourselves, the coherent state $|\alpha\rangle$ is defined as the eigenstate associated with eigenvalue α of the annihilation operator. Now, for a given target statevector $|\psi_t\rangle$ (in this case, $|\psi_t\rangle = |\alpha\rangle$) in the Fock basis, its Uhlmann's fidelity, over overlap, with the pure output state $|\psi\rangle$ is given by

$$F = |\langle\psi|\psi_t\rangle|^2 \quad (54)$$

, which is always non-negative and not exceeding 1 by the Cauchy-Schwarz inequality. Here,

we run an optimization for *hyperparameter* Θ containing all parameters involved in each operation/gate in each layer of the circuit so as to maximize the fidelity F near 1 as possible. We now let $\alpha = \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}i = e^{i\frac{\pi}{4}}$, and knowing the effect of the displacement gate acting on the vacuum state

$$D(z)|0\rangle = |z\rangle \tag{55}$$

with $z = re^{i\phi}$, it is expected that r will be optimized to around 1 and ϕ to around $\frac{\pi}{4}$. Script [5.2](#) shows how to perform this optimization of the displacement gate parameters.

Script 5.5: Optimizing to coherent state via displacement gate



```

1 # Initialize engine, program objects, and parameters
2 eng = sf.Engine(backend="tf", backend_options={"cutoff_dim": 6})
3 circuit = sf.Program(1)
4 tf_r = tf.Variable(tf.random.normal(shape=[], stddev=0.001))
5 tf_phi = tf.Variable(tf.random.normal(shape=[], stddev=0.001))
6 r, phi = circuit.params("r", "phi")
7 # Define circuit
8 with circuit.context as q:
9     sf.ops.Dgate(r, phi) | q[0]
10
11 opt = tf.keras.optimizers.Adam(learning_rate=0.1) # Define optimizer
12 steps = 100
13 best_fid = 0
14 # Target coherent state in Fock basis
15 alpha = 0.70710678118+0.70710678118j
16 coh = lambda a, dim: np.array([np.exp(-0.5 * np.abs(a) ** 2) * (a) ** n /
    ↪ np.sqrt(np.math.factorial(n)) for n in range(dim)])
17 target_statevec = coh(alpha, cutoff)
18 # Optimization starts here
19 for step in range(steps):
20
21     # Reset the engine if it has already been executed
22     if eng.run_progs:
23         eng.reset()
24
25     with tf.GradientTape() as tape:
26         # Execute the engine
27         results = eng.run(circuit, args={"r": tf_r, "phi": tf_phi})
28         # Get the probability of fock state |1>
29         fid = results.state.fidelity_coherent([alpha])
30         # Save ket/statevector
31         ket = results.state.ket()
32         # Negative sign to maximize prob
33         loss = 1-tf.sqrt(fid)
34
35     gradients = tape.gradient(loss, [tf_r, tf_phi])
36     opt.apply_gradients(zip(gradients, [tf_r, tf_phi]))
37     print("Fidelity at step {}: {}".format(step, fid))
38     if fid > best_fid:
39         best_fid = fid
40         best_r = tf_r
41         best_phi = tf_phi
42         learnt_state = ket.numpy()
43
44     if step == 0:
45         first_step_state = ket.numpy()
46
47 print(best_fid.numpy(), best_r.numpy(), best_phi.numpy())

```

In fact, one can plot the Wigner phase-space representation of the target, initial and learnt states, shown respectively in Figure 14. As the optimized parameters $r \approx 1.0003$ and $\phi \approx 0.78734136$ are close to expected, the high fidelity of 0.9988 can be visualized by the high similarity of the Wigner representations of the target and learnt states.

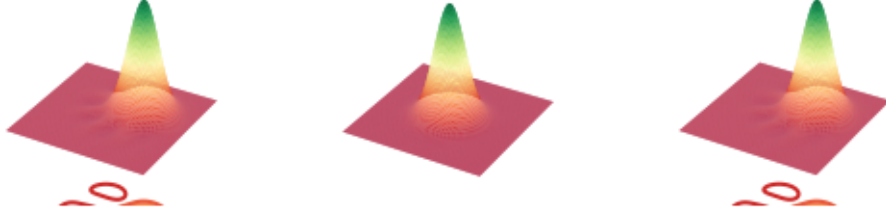


Figure 14: Wigner representations of the target, initial and learnt states

We now move on to numerically demonstrate the Hong-Ou-Mandel effect,²⁰ which states that when two identical photons enter a 50-50 beamsplitter (i.e., at $\theta = \frac{\pi}{4}$), they will always exit together in the same output mode. In the Fock basis, an input state of $|11\rangle$ will be transformed to either $|02\rangle$ or $|20\rangle$, regardless of the phase angle ϕ . Script 5.2 shows how the transmittivity angle θ and phase angle ϕ will be optimized so that the total probability of obtaining $|02\rangle$ and $|20\rangle$ is maximized near 1 as possible. As expected, θ is optimized to $\pm\frac{\pi}{4}$ (the sign of θ is not important, since the expression describing the beamsplitter effect, given in Table 2, only involves $\cos\theta$). On the other hand, ϕ is optimized to around 0, since we started with a normal distribution for ϕ centered around 0, and the Hong-Ou-Mandel effect holds regardless of the value of ϕ .



```
1 # Initialize engine and program objects
2 eng = sf.Engine(backend="tf", backend_options={"cutoff_dim":
  ↪ 7, "prepare_fock_state": [1,0], "prepare_fock_state": [1,1]})
3 circuit = sf.Program(2)
4 tf_theta = tf.Variable(tf.random.normal(shape=[], stddev=0.001))
5 tf_phi = tf.Variable(tf.random.normal(shape=[], stddev=0.001))
6 theta, phi = circuit.params("theta", "phi")
7 # Define circuit
8 with circuit.context as q:
9     sf.ops.Fock(1) | q[0]
10    sf.ops.Fock(1) | q[1]
11    sf.ops.BSgate(theta, phi) | (q[0],q[1])
12 # Define parameters for optimization
13 opt = tf.keras.optimizers.Adam(learning_rate=0.1)
14 steps = 50
15 best_prob = 0
16 # Start optimization
17 for step in range(steps):
18     if eng.run_progs:
19         eng.reset()
20     with tf.GradientTape() as tape:
21         # execute the engine
22         results = eng.run(circuit, args={"theta": tf_theta, "phi": tf_phi})
23         # get the probability of fock state |02> + |20>
24         prob = results.state.fock_prob([0,2])+results.state.fock_prob([2,0])
25         # negative sign to maximize prob
26         loss = 1-tf.sqrt(prob)
27         gradients = tape.gradient(loss, [tf_theta, tf_phi])
28         opt.apply_gradients(zip(gradients, [tf_theta, tf_phi]))
29         print("Prob at step {}: {}".format(step, prob))
30     if prob > best_prob:
31         best_prob = prob
32         best_theta = tf_theta
33         best_phi = tf_phi
34         best_state = results.state
35 print(best_prob.numpy(),best_theta.numpy(),best_phi.numpy())
```

6 Conclusion

In Part I of our tutorial, we explained the process of simulating quantum dynamics according to the Time-Dependent Schrodinger Equation with classical computers, qubit-based quantum computers and qumode-based quantum computers. We have also covered a series

of advanced quantum algorithms that facilitate the implementation of complicated chemical dynamics on quantum computers, including Sum of Unitaries decomposition for Hamiltonian simulation, propagation with Trotterization, and Variational Quantum Eigensolver for both time evolution and optimization. We expect that Part I of our tutorial serves as a starting point for carrying out molecular quantum dynamics simulations on quantum computers, and for quantum simulations of Markovian and Non-Markovian open quantum systems that we are going to cover in Part II and III.

Acknowledgement

We acknowledge the financial support of the National Science Foundation under award number 2124511, CCI Phase I: NSF Center for Quantum Dynamics on Modular Quantum Devices (CQD-MQD).

References

- (1) Johansson, J.; Nation, P.; Nori, F. QuTiP: An open-source Python framework for the dynamics of open quantum systems. *Computer Physics Communications* **2012**, *183*, 1760–1772.
- (2) Johansson, J.; Nation, P.; Nori, F. QuTiP 2: A Python framework for the dynamics of open quantum systems. *Comput. Phys. Commun.* **2013**, *184*, 1234–1240.
- (3) Godbeer, A. D.; Al-Khalili, J. S.; Stevenson, P. D. Modelling proton tunnelling in the adenine–thymine base pair. *Physical Chemistry Chemical Physics* **2015**, *17*, 13034–13044.
- (4) Soley, M. B.; Bergold, P.; Gorodetsky, A. A.; Batista, V. S. Functional Tensor-Train Chebyshev Method for Multidimensional Quantum Dynamics Simulations. *Journal of Chemical Theory and Computation* **2021**, *18*, 25–36.

- (5) 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.
- (6) Fiori, E. R.; Pastawski, H. Non-Markovian decay beyond the Fermi Golden Rule: Survival collapse of the polarization in spin chains. *Chem. Phys. Lett.* **2006**, *420*, 35–41.
- (7) Yuan, X.; Endo, S.; Zhao, Q.; Li, Y.; Benjamin, S. C. Theory of variational quantum simulation. *Quantum* **2019**, *3*, 191.
- (8) 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*.
- (9) 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.
- (10) Weedbrook, C.; Pirandola, S.; García-Patrón, R.; Cerf, N. J.; Ralph, T. C.; Shapiro, J. H.; Lloyd, S. Gaussian quantum information. *Rev. Mod. Phys.* **2012**, *84*, 621–669.
- (11) 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.
- (12) Bromley, T. R.; Arrazola, J. M.; Jahangiri, S.; Izaac, J.; Quesada, N.; Gran, A. D.; Schuld, M.; Swinarton, J.; Zabaneh, Z.; Killoran, N. Applications of near-term photonic quantum computers: software and algorithms. *Quantum Science and Technology* **2020**, *5*, 034010.

- (13) Lloyd, S.; Braunstein, S. L. Quantum Computation over Continuous Variables. *Phys. Rev. Lett.* **1999**, *82*, 1784–1787.
- (14) Sefi, S.; van Loock, P. How to Decompose Arbitrary Continuous-Variable Quantum Operations. *Phys. Rev. Lett.* **2011**, *107*, 170501.
- (15) Aspuru-Guzik, A.; Dutoi, A. D.; Love, P. J.; Head-Gordon, M. Simulated Quantum Computation of Molecular Energies. *Science* **2005**, *309*, 1704–1707.
- (16) James D. Whitfield, J. B.; Aspuru-Guzik, A. Simulation of electronic structure Hamiltonians using quantum computers. *Molecular Physics* **2011**, *109*, 735–750.
- (17) Kalajdzievski, T.; Quesada, N. Exact and approximate continuous-variable gate decompositions. *Quantum* **2021**, *5*, 394.
- (18) Scalettar, R.
- (19) 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.
- (20) Hong, C. K.; Ou, Z. Y.; Mandel, L. Measurement of subpicosecond time intervals between two photons by interference. *Physical Review Letters* **1987**, *59*, 2044–2046.

7 Appendix

7.1 Source Code

7.2 Software Requirements