Simulation of the Dynamics of the Heisenberg Model using IBM Quantum Computers

Delmar Cabral, Batista Group

August 4th, 2023
What does the Heisenberg Hamiltonian describe?

Hamiltonian for the Heisenberg model, defined as follows:

\[
H = \sum_{n=0}^{N-1} \hbar \Omega_n \hat{\sigma}^z_n - \frac{1}{2} \sum_{n=0}^{N-2} \left( J_{n,n+1}^x \hat{\sigma}^x_n \hat{\sigma}^x_{n+1} + J_{n,n+1}^y \hat{\sigma}^y_n \hat{\sigma}^y_{n+1} + J_{n,n+1}^z \hat{\sigma}^z_n \hat{\sigma}^z_{n+1} \right)
\]

Dynamics:

Which chemical systems can be studied using this methodology?

Figure: Magnetization along graphene nanoribbon\(^2\)

\(^2\)Nano Lett. 2022, 22, 1, 164–171
Connecting $H$ to the molecular system

Hamiltonian for the Heisenberg model, defined as follows:

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

Parameters used throughout this presentation and associated notebook:\footnote{Non-Markovian decay beyond the Fermi Golden Rule: Survival Collapse of the polarization in spin chains}

- $N = 3$
- $\hbar \Omega_0 = 0.65$
- $\hbar \Omega_n = 1.0, \ n > 0$
- $J_{0,1}^x = J_{1,0}^y = 0.75$
- $J_{n,n+1}^x = J_{n,n+1}^y = 1.0, \ n > 0$
- $J_{n,n+1}^z = 0, \ \forall n$

Functionalized graphene nanoribbon with unpaired electrons:

Adapted from Nano Lett. 2022, 22, 1, 164–171
How to perform the time-evolution?

We propagate the initial state, $|\psi_0\rangle$, using the time-evolution operator:

$$e^{-i\hat{H}t/\hbar} = e^{-i(\hat{H}_z + \hat{H}_{xx} + \hat{H}_{yy} + \hat{H}_{zz})t/\hbar}$$

We use the Trotter decomposition formula to implement the exponential of a matrix:

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

We decompose the Hamiltonian into even and odd components, uncoupling the terms as follows:

$$e^{-i\hat{H}t/\hbar} \rightarrow e^{-i\hat{H}_{\text{even}} t/2\hbar} e^{-i\hat{H}_{\text{odd}} t/\hbar} e^{-i\hat{H}_{\text{even}} t/2\hbar}$$
Which libraries are needed?

For Hamiltonian creation:

```python
from qiskit.quantum_info import Pauli, Operator, SparsePauliOp
```

For time-evolution operator

```python
from qiskit.circuit.library import PauliEvolutionGate
```

Trotter-Suzuki implementation for decomposition of exponentials of matrices

```python
from qiskit.synthesis import SuzukiTrotter
```

For quantum circuit creation and execution:

```python
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit import transpile, execute
```
How to construct the Hamiltonian term by term using Qiskit?

Each term is a tensor product of operators. For example, the $\hat{\sigma}_n^y \hat{\sigma}_{n+1}^y$ is:

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

Which must be represented as a strings of characters $s=’I...YY...I’$;

```python
SparsePauliOp(('I' * n + 'YY' + 'I' * (n_qubits - 2 - n))) * YY_coeff
```

Similarly we can construct all terms associated with site $n$:

```python
def get_hamiltonian_n_site_terms(n, coeff, n_qubits):
    XX_coeff = coeff[0]
    YY_coeff = coeff[1]
    ZZ_coeff = coeff[2]
    Z_coeff = coeff[3]
    return (SparsePauliOp(('I' * n + 'XX' + 'I' * (n_qubits - 2 - n))) * XX_coeff +
            SparsePauliOp(('I' * n + 'YY' + 'I' * (n_qubits - 2 - n))) * YY_coeff +
            SparsePauliOp(('I' * n + 'ZZ' + 'I' * (n_qubits - 2 - n))) * ZZ_coeff +
            SparsePauliOp(('I' * n + 'Z' + 'I' * (n_qubits - 1 - n))) * Z_coeff)
```
How to construct the complete Hamiltonian?

Iterating over all sites in the chain to obtain all the Hamiltonian terms:

```python
def get_heisenberg_hamiltonian(n_qubits, coeff=None):
    assert n_qubits >= 3

    if coeff == None:
        coeff = [[1.0, 1.0, 1.0, 1.0] for i in range(n_qubits)]

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

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

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

    # Returns the list of the two sets of terms
    return [H_E, H_O]
```
def get_time_evolution_operator(num_qubits, tau, trotter_steps, coeff=None):
    # Constructing the Hamiltonian here; heisenberg_hamiltonian = [H_E, H_O]
    heisenberg_hamiltonian = get_heisenberg_hamiltonian(num_qubits, coeff)

    # e^(-i*H*evo_time), with Trotter decomposition
    # The Trotter order=2 applies one set of the operators for half a timestep,
    # then the other set for a full timestep, then the first for another half a step
    # note that reps includes the number of repetitions of the Trotterized operator
    # higher number means more repetitions, and thus allowing larger timestep
    evo_op = PauliEvolutionGate(heisenberg_hamiltonian, tau,
        synthesis=SuzukiTrotter(order=2, reps=trotter_steps))

    return evo_op.definition
How to construct the Time Evolution Operator?

```python
def get_time_evolution_operator(num_qubits, tau, trotter_steps, coeff=None):
    # Constructing the Hamiltonian here; heisenberg_hamiltonian = [H_E, H_O]
    heisenberg_hamiltonian = get_heisenberg_hamiltonian(num_qubits, coeff)

    # e^{-i*H*evo_time}, with Trotter decomposition
    # The Trotter order=2 applies one set of the operators for half a timestep,
    # then the other set for a full timestep, then the first for another half a step
    # note that reps includes the number of repetitions of the Trotterized operator
    # higher number means more repetitions, and thus allowing larger timestep
    evo_op = PauliEvolutionGate(heisenberg_hamiltonian, tau,
                                synthesis=SuzukiTrotter(order=2, reps=trotter_steps))

    return evo_op.definition
```

```python
num_q = 3
evolution_timestep = 0.1
n_trotter_steps = 1
# XX YY ZZ, Z
hamiltonian_coefficients = (
    [[0.75/2, 0.75/2, 0.0, 0.65]]
    + [[0.5, 0.5, 0.0, 1.0]
        for i in range(num_q-1)])

time_evo_op = get_time_evolution_operator(
    num_qubits=num_q, tau=evolution_timestep,
    trotter_steps=n_trotter_steps,
    coeff=hamiltonian_coefficients)
```
def get_time_evolution_operator(num_qubits, tau, trotter_steps, coeff=None):
    # Constructing the Hamiltonian here; heisenberg_hamiltonian = [H_E, H_O]
    heisenberg_hamiltonian = get_heisenberg_hamiltonian(num_qubits, coeff)

    # e^(-i*H*evo_time), with Trotter decomposition
    # The Trotter order=2 applies one set of the operators for half a timestep,
    # then the other set for a full timestep, then the first for another half a step
    # note that reps includes the number of repetitions of the Trotterized operator
    # higher number means more repetitions, and thus allowing larger timestep
    evo_op = PauliEvolutionGate(heisenberg_hamiltonian, tau,
                                synthesis=SuzukiTrotter(order=2, reps=trotter_steps))

    return evo_op.definition

num_q = 3
evolution_timestep = 0.1
n_trotter_steps = 1
# XX YY ZZ, Z
hamiltonian_coefficients = ([
     [0.75/2, 0.75/2, 0.0, 0.65] +
     [0.5, 0.5, 0.0, 1.0]
     for i in range(num_q-1)])

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

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

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

time_evo_op = get_time_evolution_operator(
    num_qubits=num_q, tau=evolution_timestep,
    trotter_steps=n_trotter_steps,
    coeff=hamiltonian_coefficients)
How to create a Quantum Circuit object?

Creating a circuit object:

```python
cr = QuantumRegister(num_q)
cr = ClassicalRegister(num_q)
qc = QuantumCircuit(qr, cr)  # instantiated here
```

\[ |0\rangle \otimes |0\rangle \otimes |0\rangle = |000\rangle \]
How to initialize the Quantum Circuit with $\psi_0$?

Creating a circuit object:

```python
qr = QuantumRegister(num_q)
cr = ClassicalRegister(num_q)
qc = QuantumCircuit(qr, cr) # instantiated here
```

$|0\rangle \otimes |0\rangle \otimes |0\rangle = |000\rangle$

Initializing by bit-flipping:

```python
for qubit_idx in range(num_q):
    if qubit_idx == 0:
        # generate only one spin-up at first qubit
        qc.i(qubit_idx)
    else:
        # make all other spins have the spin-down state
        qc.x(qubit_idx)
```

$I|0\rangle \otimes X|0\rangle \otimes X|0\rangle = |011\rangle$
How to initialize the Quantum Circuit with $\psi_0$?

Creating a circuit object:

```python
qr = QuantumRegister(num_q)
cr = ClassicalRegister(num_q)
qc = QuantumCircuit(qr, cr) # instantiated here
```

$|0\rangle \otimes |0\rangle \otimes |0\rangle = |000\rangle$

Initializing by bit-flipping:

```python
for qubit_idx in range(num_q):
    if qubit_idx == 0:
        # generate only one spin-up at first qubit
        qc.i(qubit_idx)
    else:
        # make all other spins have the spin-down state
        qc.x(qubit_idx)
```

$|0\rangle \otimes X|0\rangle \otimes X|0\rangle = |011\rangle$

or by amplitude encoding:

```python
qr_init = QuantumRegister(num_qubits)
qc_init = QuantumCircuit(qr_init)
qc_init.initialize('011', qr_init[:])
qc.append(qc_init)
```

$|000\rangle \rightarrow |011\rangle$
How to include the time-evolution circuit?

Recalling the time-evolution operator we constructed earlier:

```python
import qiskit

time_evo_op = get_time_evolution_operator(
    num_qubits=num_q, tau=evolution_timestep,
    trotter_steps=n_trotter_steps,
    coeff=hamiltonian_coefficients)
```

Appending the Hamiltonian circuit object:

```python
# appending the Hamiltonian evolution to the circuit
qc.append(time_evo_op, list(range(num_q)))
```

Including the measurement operation:

```python
qc.measure(range(num_q), range(num_q))
```
How to execute the circuit?

Choosing a simulator backend:

```
backend = AerSimulator(method='statevector')
```

Or a real device via IBMQ:

```
from qiskit import IBMQ

IBMQ.save_account('TOKEN')
IBMQ.load_account()  # Load account from disk

print(IBMQ.providers())  # List all available providers
provider = IBMQ.get_provider(hub='ibm-q')
print(provider.backends())

backend = provider.backend.ibmq_belem
```

And executing on that backend:

```
qc.measure(range(num_q), range(num_q))

qct = transpile(qc, backend, optimization_level=2)
qct_run = execute(qct, backend, shots=1000).result()
qct_run_counts = qct_run.get_counts()
```
Statevector Simulator Results for the Survival Amplitude

Simulating with statevector and calculating the survival amplitude, $\langle \psi_0 | \psi_f \rangle$
What is the Hadamard test?

Controlled operation contains the unitary of interest (such as time-evolution operator)

\[ \text{Ancilla} \quad \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle) \]

\[ |\psi\rangle \]

\[ \text{Initial State} \]

\[ \text{Ancilla} \quad \frac{1}{\sqrt{2}} (|0\rangle - i|1\rangle) \]

\[ |\psi\rangle \]

\[ \text{Initial State} \]

\[ \text{Re}\{\langle \psi | U | \psi \rangle\} \]

\[ \text{Im}\{\langle \psi | U | \psi \rangle\} \]
How to analyze the results of the Hadamard test?

Measurement is performed in the $\sigma_z$ basis:

\[
\langle 0 | \sigma_z | 0 \rangle = 1 \\
\langle 1 | \sigma_z | 1 \rangle = -1
\]

Thus we account for the number of measurements of the ancilla in the $|0\rangle$ and $|1\rangle$ states:

\[
\langle \psi | U | \psi \rangle \rightarrow \frac{\langle 0 | \sigma_z | 0 \rangle N_{|0\rangle} + \langle 1 | \sigma_z | 1 \rangle N_{|1\rangle}}{N_{|0\rangle} + N_{|1\rangle}}
\]

\[
= \frac{N_{|0\rangle} - N_{|1\rangle}}{N_{|0\rangle} + N_{|1\rangle}}
\]

```python
qubit_to_spin_map = {'0': 1, '1': -1}  # maps state 0 to eigenvalue 1 and 1 to eigen -1
total_counts = 0
values_list = []
for k,v in counts.items():  # counts has number of times 0 and 1 were measured
    values_list.append(qubit_to_spin_map[k] * v)
    total_counts += v
average_spin = (sum(values_list)) / total_counts
```
How to execute the Hadamard test for our operator?

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

```python
controlled_time_evo_op = time_evo_op.control()
```

Executing Hadamard test for all times,

```python
real_amp_list = []
imag_amp_list = []
for idx, time in enumerate(time_range):
    qc_had_real = get_hadamard_test(num_q, init_circ, controlled_time_evo_op,
                                   control_repeats=idx, imag_expectation=False)
    had_real_counts = get_circuit_execution_counts(qc_had_real, simulator,
                                                   n_shots=num_shots)
    real_amplitude = get_spin_correlation(had_real_counts)
    real_amp_list.append(real_amplitude)

    qc_had_imag = get_hadamard_test(num_q, init_circ, controlled_time_evo_op,
                                    control_repeats=idx, imag_expectation=True)
    had_imag_counts = get_circuit_execution_counts(qc_had_imag, simulator,
                                                   n_shots=num_shots)
    imag_amplitude = get_spin_correlation(had_imag_counts)
    imag_amp_list.append(imag_amplitude)

real_amp_array = np.array(real_amp_list)
imag_amp_array = np.array(imag_amp_list)
```
How to construct the Hadamard test circuit?

```python
def get_hadamard_test(num_qubits, initial_state, control_operation, control_repeats=0, imag_expectation=False):
    qr_hadamard = QuantumRegister(num_qubits + 1)
    cr_hadamard = ClassicalRegister(1)
    qc_hadamard = QuantumCircuit(qr_hadamard, cr_hadamard) # instantiated here

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

    qc_hadamard.h(0)
    if imag_expectation:
        qc_hadamard.p(-np.pi/2, 0)

    # Repeatedly adds the control operation to reach time tau
    for i in range(control_repeats):
        qc_hadamard.append(control_operation, qr_hadamard[:])

    qc_hadamard.h(0)
    qc_hadamard.barrier()

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

    return qc_hadamard
```

Delmar Cabral, Batista Group

Simulation of the Dynamics of the Heisenberg Model using IBM Quantum Computers

August 4th, 2023 21 / 23
Including Hadamard Test Results for the Survival Applitude

Simulating with statevector and calculating $\langle \psi_0 | \psi_f \rangle$
What did this presentation cover?

Steps for simulating quantum dynamics using Qiskit:

- How to construct Hamiltonian operator (sum of tensor product of Pauli matrices)
- How to construct Trotter approximation of time-evolution operator \( e^{-i\hat{H}t} \)
- How to compose a circuit combining initialization and the time-evolution operator
- How to execute the combined circuit and perform final state measurement
- How to compute observables with the Hadamard test (correlation function and observables)