Quantum Computing

Chapter 07: System Dynamics, Simulation, and Polynomial Functions

Prachya Boonkwan

Version: January 14, 2026

Sirindhorn International Institute of Technology
Thammasat University, Thailand

Who? Me?

- Nickname: Arm (P'/N' Arm, etc.)
- Born: Aug 1981
- Work
  - Researcher at NECTEC 2005-2024
  - Lecturer at SIIT, Thammasat University 2025-now
- Education
  - B.Eng & M.Eng in Computer Engineering, Kasetsart University, Thailand
  - Obtained Ministry of Science and Technology Scholarship of Thailand in early 2008
  - Did a PhD in Informatics (AI & Computational Linguistics) at University of Edinburgh, UK from 2008 to 2013

Table of Contents

1. Quantum Finite-State Automaton

2. Quantum System

3. Polynomial Functions

4. HHL Algorithm

5. Conclusion

Quantum Finite-State Automaton

System Dynamics

- A complex system is a system comprised of multiple elements that interact with one another
- System dynamics is a mathematical model that represents a non-linear behavior of a complex system over time in terms of state transitions
- There are two types of system dynamics: deterministic and probabilistic
- Simulation of system dynamics helps us understand the system's behavior in a long run

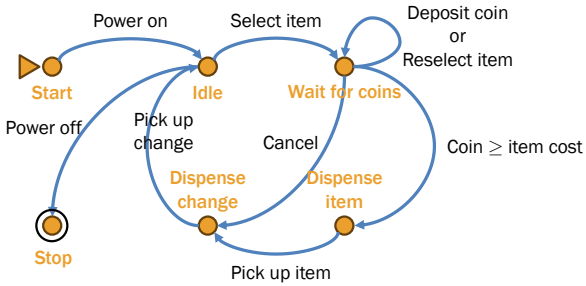


Figure 1: The system dynamics of a vending machine consists of six states. The start state is marked with a triangle, while each final state with double circles. State transition takes place when there is an input stimulus at a particular state. For example, The vending machine transitions from the idle state to the wait-for-coins state when the customer selects an item.

Deterministic Finite-State Automaton

- Finite-state automaton (FSA) describes a system dynamics over a finite set of possible internal configurations
- Deterministic FSA (DFA) is an FSA defined as a quintuple $(K, \Sigma, \delta, s, F)$, where
  - K is a finite set of states
  - Σ is an alphabet (set of input symbols)
  - $s \in K$ is the start state
  - $F \subseteq K$ is the set of final states
  - $\delta : K \times \Sigma \mapsto K$ is a function that takes a current state and an input symbol and yields the next state; i.e. $\delta(q_{\text{curr}}, a) = q_{\text{next}}$
- Derivation: $(q_{\text{curr}}, aw) \vdash (q_{\text{next}}, w)$, where $\delta(q_{\text{curr}}, a) = q_{\text{next}}$
- Time evolution of an input sequence $a_1 \dots a_N$ is a derivation $(q^{(0)}, a_1 \dots a_N) \vdash (q^{(1)}, a_2 \dots a_N) \vdash (q^{(2)}, a_3 \dots a_N) \vdash \dots \vdash (q^{(N)}, \varepsilon)$, where $q^{(0)} = s$ and $q^{(N)} \in F$
- Acceptance: $(s, w) \vdash^* (q, \varepsilon)$ and $q \in F$

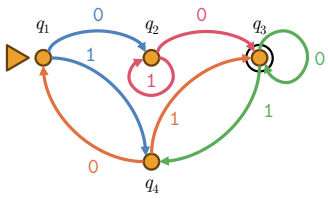


Figure 2: Deterministic finite-state automaton. The start state is marked with a triangle, while each final state is marked with double circles.

Probabilistic Finite-State Automaton

- Probabilistic FSA (PFA) is an FSA defined as a quintuple $(K, \Sigma, \Delta, s, F)$, where
  - K is a finite set of states
  - Σ is an alphabet (set of input symbols)
  - $s \in K$ is the start state
  - $F \subseteq K$ is the set of final states
  - Δ is a conditional probability over the next states given a current state and an input symbol; i.e. $\Delta(q_{\text{next}}|q_{\text{curr}}, a)$
- Probability of time evolution

$$P(q^{(1)}, \ldots, q^{(N)}|a_1, \ldots, a_N) = \prod_{k=1}^{N} \Delta(q^{(k)}|q^{(k-1)}, a_k)$$

where $q^{(0)} = s$ and $q^{(N)} \in F$

- Example: Transition from state q_1 with input symbols `0` and `1` is dictated by:

$$\Delta(q_2|q_1, 0) = 0.6 \qquad \Delta(q_4|q_1, 0) = 0.4$$
$$\Delta(q_2|q_1, 1) = 0.3 \qquad \Delta(q_4|q_1, 1) = 0.7$$

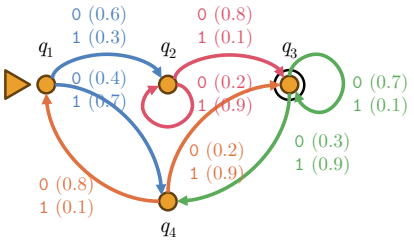


Figure 3: Probabilistic finite-state automaton

Quantum Finite-State Automaton

- Quantum FSA (QFA) is defined as a quintuple $(K, \Sigma, \mathbf{U}, s, \mathbf{F})$, where
  - K is a Hilbert space of N dimensions
  - Σ is an alphabet (set of input symbols)
  - $s \in K$ is a start quantum state
  - $\mathbf{F}$ is a projector matrix of final states
  - $\mathbf{U} : \Sigma \mapsto \mathbb{C}^L \times \mathbb{C}^L$ is a function of quantum operators for probabilistic state transition of each input symbol
- Derivation: $(q_{\text{curr}}, aw) \vdash (q_{\text{next}}, w)$ is a probability distribution over the next states:

$$P(q_{\text{next}} | q_{\text{curr}}, a) = |\mathbf{U}_a |q_{\text{curr}}\rangle|^2$$

- Time evolution: $(q^{(0)}, a_1 \ldots a_N) \vdash^* (q^{(N)}, \varepsilon)$

$$\left|\psi^{(N)}\right\rangle = \mathbf{U}_{a_N} \ldots \mathbf{U}_{a_1} \left|\psi^{(0)}\right\rangle$$

where $\left|\psi^{(0)}\right\rangle$ is the quantum equivalent of the start state s:

$$\begin{aligned} \left|\psi^{(0)}\right\rangle &= \frac{1}{\sqrt{|K|}} \sum_q |s, q\rangle \\ &= |s\rangle \otimes |+\rangle^{\otimes \log_2 |K|} \end{aligned}$$

- Acceptance:

$$P_{\text{acc}}(\psi) = \left\langle \mathbf{F}^{(\psi)} \right\rangle = \langle \psi | \mathbf{F} | \psi \rangle$$

Szegedy's Algorithm

- Conversion from PFA to QFA: For all input symbols $a \in \Sigma$, we convert each state into its quantum equivalent

$$\left|\psi_{a,q}\right\rangle = \sum_{q'} |q, q'\rangle \sqrt{\Delta(q'|q,a)}$$

- We compute the diffusion matrix for each input symbol a:

$$\Pi_a = \sum_{q} \left|\psi_{a,q}\right\rangle \left\langle\psi_{a,q}\right|$$

- Then, we compute the reflection matrix

$$\mathbf{R}_a = 2\Pi_a - \mathbf{I}$$

- Our quantum walk operator $\mathbf{U}_a$ is

$$\mathbf{U}_a = \mathbf{SWAP}(\mathbf{R}_a)$$

where $\mathbf{SWAP}(|i\rangle\langle j|) = |j\rangle\langle i|$

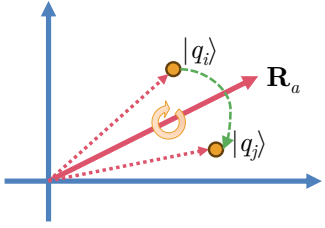


Figure 4: Quantum walk is a sequence of reflection upon the axis $\mathbf{R}_a$ and axis rotation via swapping, imitating state transition from q_i to q_j by symbol a.

Szegedy's Algorithm (cont'd)

- Example: Convert the PFA in Fig 5 to QFA.
- Markov matrices for each input symbol are

$$\mathbf{P}_a = \begin{bmatrix} 9/25 & 3/4 \\ 16/25 & 1/4 \end{bmatrix} \quad \mathbf{P}_b = \begin{bmatrix} 16/25 & 1/4 \\ 9/25 & 3/4 \end{bmatrix}$$

- Let's construct the unitary operators.

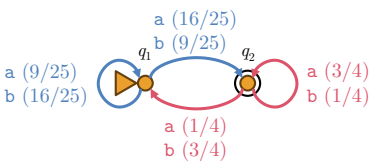


Figure 5: Simple PFA

- Step 1: We convert each state q to its quantum equivalent $|\psi_{a,q}\rangle$.

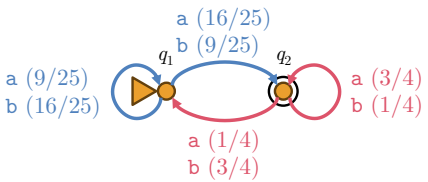
$$\begin{aligned} |\psi_{a,q_1}\rangle &= \frac{3}{5}|00\rangle + \frac{\sqrt{3}}{2}|01\rangle \\ |\psi_{a,q_2}\rangle &= \frac{4}{5}|10\rangle + \frac{1}{2}|11\rangle \\ |\psi_{b,q_1}\rangle &= \frac{4}{5}|00\rangle + \frac{1}{2}|01\rangle \\ |\psi_{b,q_2}\rangle &= \frac{3}{5}|10\rangle + \frac{\sqrt{3}}{2}|11\rangle \end{aligned}$$

Szegedy's Algorithm (cont'd)

- Example: Convert the PFA in Fig 5 to QFA.
- Markov matrices for each input symbol are

$$\mathrm{P}_a = \begin{bmatrix} 9/25 & 3/4 \\ 16/25 & 1/4 \end{bmatrix} \quad \mathrm{P}_b = \begin{bmatrix} 16/25 & 1/4 \\ 9/25 & 3/4 \end{bmatrix}$$

- Let's construct the unitary operators.



- Step 2: We compute the diffusion matrix $\Pi_a = \sum_q \left|\psi_{a,q}\right\rangle \left\langle\psi_{a,q}\right|$.

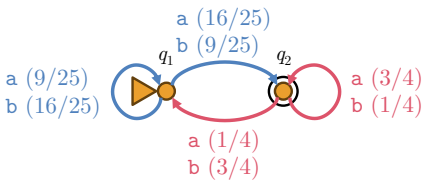
$$\begin{aligned} \Pi_a &= \left|\psi_{a,q_1}\right\rangle \left\langle\psi_{a,q_1}\right| + \left|\psi_{a,q_2}\right\rangle \left\langle\psi_{a,q_2}\right| \\ &= \frac{9}{25}|00\rangle\langle 00| + \frac{3\sqrt{3}}{10}|00\rangle\langle 01| \\ &\quad + \frac{3\sqrt{3}}{10}|01\rangle\langle 00| + \frac{3}{4}|01\rangle\langle 01| \\ &\quad + \frac{16}{25}|10\rangle\langle 10| + \frac{4}{10}|10\rangle\langle 11| \\ &\quad + \frac{4}{10}|11\rangle\langle 10| + \frac{1}{4}|11\rangle\langle 11| \end{aligned}$$

Szegedy's Algorithm (cont'd)

- Example: Convert the PFA in Fig 5 to QFA.
- Markov matrices for each input symbol are

$$\mathbf{P}_a = \begin{bmatrix} 9/25 & 3/4 \\ 16/25 & 1/4 \end{bmatrix} \quad \mathbf{P}_b = \begin{bmatrix} 16/25 & 1/4 \\ 9/25 & 3/4 \end{bmatrix}$$

- Let's construct the unitary operators.



- Step 2: We compute the diffusion matrix $\Pi_a = \sum_q \left|\psi_{a,q}\right\rangle \left\langle\psi_{a,q}\right|$.

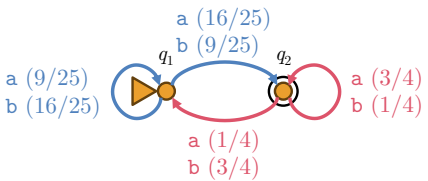
$$\begin{aligned} \Pi_b &= \left|\psi_{b,q_1}\right\rangle \left\langle\psi_{b,q_1}\right| + \left|\psi_{b,q_2}\right\rangle \left\langle\psi_{b,q_2}\right| \\ &= \frac{16}{25}|00\rangle\langle 00| + \frac{4}{10}|00\rangle\langle 01| \\ &\quad + \frac{4}{10}|01\rangle\langle 00| + \frac{1}{4}|01\rangle\langle 01| \\ &\quad + \frac{9}{25}|10\rangle\langle 10| + \frac{3\sqrt{3}}{10}|10\rangle\langle 11| \\ &\quad + \frac{3\sqrt{3}}{10}|11\rangle\langle 10| + \frac{3}{4}|11\rangle\langle 11| \end{aligned}$$

Szegedy's Algorithm (cont'd)

- Example: Convert the PFA in Fig 5 to QFA.
- Markov matrices for each input symbol are

$$\mathbf{P}_a = \begin{bmatrix} 9/25 & 3/4 \\ 16/25 & 1/4 \end{bmatrix} \quad \mathbf{P}_b = \begin{bmatrix} 16/25 & 1/4 \\ 9/25 & 3/4 \end{bmatrix}$$

- Let's construct the unitary operators.



- Step 3: We compute the reflection matrix $\mathbf{R}_a = 2\Pi_a - \mathbf{I}$.

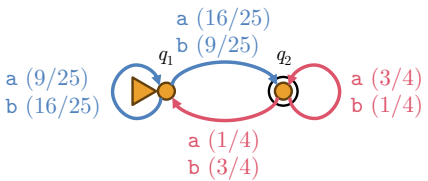
$$\begin{aligned} \mathbf{R}_a &= 2\Pi_a - \mathbf{I} \\ &= -\frac{7}{25}|00\rangle\langle 00| + \frac{6\sqrt{3}}{10}|00\rangle\langle 01| \\ &\quad + \frac{6\sqrt{3}}{10}|01\rangle\langle 00| + \frac{2}{4}|01\rangle\langle 01| \\ &\quad + \frac{7}{25}|10\rangle\langle 10| + \frac{8}{10}|10\rangle\langle 11| \\ &\quad + \frac{8}{10}|11\rangle\langle 10| - \frac{2}{4}|11\rangle\langle 11| \end{aligned}$$

Szegedy's Algorithm (cont'd)

- Example: Convert the PFA in Fig 5 to QFA.
- Markov matrices for each input symbol are

$$\mathrm{P}_a = \begin{bmatrix} 9/25 & 3/4 \\ 16/25 & 1/4 \end{bmatrix} \quad \mathrm{P}_b = \begin{bmatrix} 16/25 & 1/4 \\ 9/25 & 3/4 \end{bmatrix}$$

- Let's construct the unitary operators.



- Step 3: We compute the reflection matrix $\mathrm{R}_a = 2\Pi_a - \mathrm{I}$.

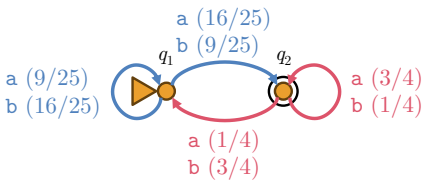
$$\begin{aligned} \mathrm{R}_b &= 2\Pi_b - \mathrm{I} \\ &= \frac{7}{25}|00\rangle\langle 00| + \frac{8}{10}|00\rangle\langle 01| \\ &\quad + \frac{8}{10}|01\rangle\langle 00| - \frac{2}{4}|01\rangle\langle 01| \\ &\quad - \frac{7}{25}|10\rangle\langle 10| + \frac{6\sqrt{3}}{10}|10\rangle\langle 11| \\ &\quad + \frac{6\sqrt{3}}{10}|11\rangle\langle 10| + \frac{2}{4}|11\rangle\langle 11| \end{aligned}$$

Szegedy's Algorithm (cont'd)

- Example: Convert the PFA in Fig 5 to QFA.
- Markov matrices for each input symbol are

$$\mathbf{P}_a = \begin{bmatrix} 9/25 & 3/4 \\ 16/25 & 1/4 \end{bmatrix} \quad \mathbf{P}_b = \begin{bmatrix} 16/25 & 1/4 \\ 9/25 & 3/4 \end{bmatrix}$$

- Let's construct the unitary operators.



- Step 4: We compute the transition operator $\mathbf{U}_a = \mathbf{SWAP}(\mathbf{R}_a)$.

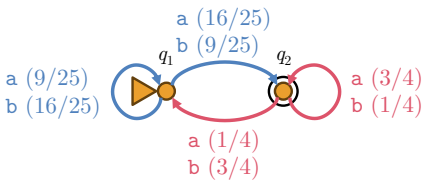
$$\begin{aligned} \mathbf{U}_a &= \mathbf{SWAP}(\mathbf{R}_a) \\ &= -\frac{7}{25}|00\rangle\langle 00| + \frac{6\sqrt{3}}{10}|01\rangle\langle 00| \\ &\quad + \frac{6\sqrt{3}}{10}|00\rangle\langle 01| + \frac{2}{4}|01\rangle\langle 01| \\ &\quad + \frac{7}{25}|10\rangle\langle 10| + \frac{8}{10}|11\rangle\langle 10| \\ &\quad + \frac{8}{10}|10\rangle\langle 11| - \frac{2}{4}|11\rangle\langle 11| \end{aligned}$$

Szegedy's Algorithm (cont'd)

- Example: Convert the PFA in Fig 5 to QFA.
- Markov matrices for each input symbol are

$$\mathbf{P}_a = \begin{bmatrix} 9/25 & 3/4 \\ 16/25 & 1/4 \end{bmatrix} \quad \mathbf{P}_b = \begin{bmatrix} 16/25 & 1/4 \\ 9/25 & 3/4 \end{bmatrix}$$

- Let's construct the unitary operators.



- Step 4: We compute the transition operator $\mathbf{U}_a = \mathbf{SWAP}(\mathbf{R}_a)$.

$$\begin{aligned} \mathbf{U}_b &= \mathbf{SWAP}(\mathbf{R}_b) \\ &= \frac{7}{25}|00\rangle\langle 00| + \frac{8}{10}|01\rangle\langle 00| \\ &\quad + \frac{8}{10}|00\rangle\langle 01| - \frac{2}{4}|01\rangle\langle 01| \\ &\quad - \frac{7}{25}|10\rangle\langle 10| + \frac{6\sqrt{3}}{10}|11\rangle\langle 10| \\ &\quad + \frac{6\sqrt{3}}{10}|10\rangle\langle 11| + \frac{2}{4}|11\rangle\langle 11| \end{aligned}$$

From Paths to Quantum Lattice

- In each quantum walk, all next states are explored in parallel, gradually producing a quantum lattice that enumerates all possible paths in its structure
- Due to quantum nature, we cannot assign a definite probability to unobserved paths

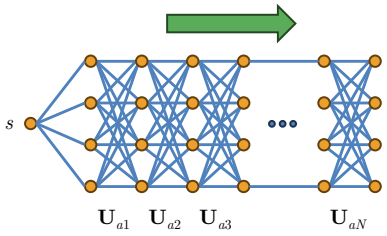


Figure 6: Quantum lattice

- Probability can be given to a path only when it is observed with measurement
- We run a simulation to observe the state in each quantum walk

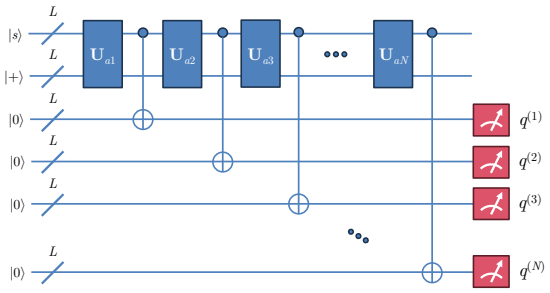


Figure 7: QFA circuit

Quantum Finite-State Automaton in PennyLane

```
no_states = 2
markov = { 'a': [ (0, 0,  9/25), (0, 1, 16/25), (1, 1, 3/4), (1, 0, 1/4) ],
           'b': [ (0, 0, 16/25), (0, 1,  9/25), (1, 1, 1/4), (1, 0, 3/4) ] }
input_seq = ['a', 'a', 'b', 'a', 'b', 'b']

no_qubits_per_state = int(np.ceil(np.log2(no_states)))
state_qubits = list(range(0, no_qubits_per_state))
trans_qubits = list(range(no_qubits_per_state,
                          no_qubits_per_state * (len(input_seq) + 1)))

dev = qml.device('default.qubit', wires = state_qubits + trans_qubits)

def markov_to_unitary(markov, no_states):
    unitary_tbl = {}
    for symbol in markov:
        mat = np.zeros((no_states, no_states), dtype=np.complex128)
        for (i, j, prob) in markov[symbol]:
            mat[j, i] = prob
        unitary_tbl[symbol] = 2 * mat - np.eye(no_states)
    return unitary_tbl
```

Quantum Finite-State Automaton in PennyLane (cont'd)

```
@qml.qnode(dev, shots=1000)
def evolution_circuit():
    unitary_tbl = markov_to_unitary(markov, 2)
    for i, symbol in enumerate(input_seq):
        qml.QubitUnitary(unitary_tbl[symbol], wires=state_qubits)
        for j in range(no_qubits_per_state):
            qml.CNOT(wires=[j, (i + 1) * no_qubits_per_state + j])
    return qml.counts(wires=trans_qubits)

evolution_circuit()
# Expected result:
#   State transitions : Counts
# { '001010'          : 150,
#   '001101'          : 425,
#   '010010'          : 150,
#   '010101'          : 12.5,
#   '101010'          : 12.5,
#   '101101'          : 50,
#   '110010'          : 150,
#   '110101'          : 50     }
```

Exercise 7.1: Quantum Finite-State Automaton

Answer the following questions.

1. Convert this PFA to Markov matrices and an equivalent QFA via Szegedy's Algorithm.

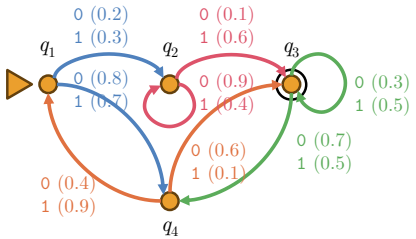


Figure 8: Example PFA

2. Discuss the differences between deterministic FSA and probabilistic FSA.
3. What are key differences between probabilistic FSA and quantum FSA?
4. Explain why we cannot directly use the Markov matrices $\mathbf{P}$ of PFA as a quantum operator $\mathbf{U}$ of QFA?
5. In Szegedy's Algorithm, what are the roles of the diffusion matrix Π_a and the reflection matrix $\mathbf{R}_a$?
6. Explain why quantum walks create a quantum lattice.

Quantum System

Simulation of System Dynamics

- Simulation is a representation of a system's trajectory (path of actions) by means of a model
- In deterministic simulation, the trajectory is dictated by a deterministic model; i.e. there is only one next state given the current state and input stimulus
- In stochastic simulation, each state transition is dictated by a probability distribution given the current state and input stimulus
- In quantum simulation, each state transition is dictated by a quantum operator, resulting in a state lattice

- Challenge: Quantum simulation is computationally expensive due to iterative matrix multiplications
- Solution: We can reduce the quantum walk operator $\mathbf{U}$ into a special form called the Hamiltonian $\hat{\mathbf{H}}$, so that matrix multiplication becomes simple addition

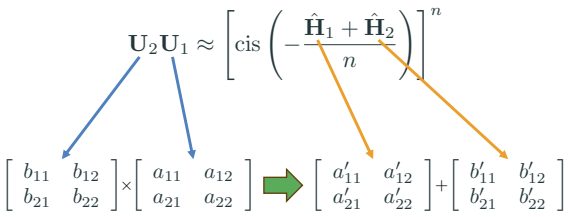


Figure 9: Trotter-Suzuki decomposition

Quantum System and Hamiltonian

- Quantum system is a QFA that describes the system dynamics via time evolution on the Hilbert space (i.e. complex vectors)
- Time evolution: $(q^{(0)}, a_1 \dots a_N) \vdash^* (q^{(N)}, \varepsilon)$

$$\left|q^{(N)}\right\rangle \quad = \quad \mathbf{U}_N \dots \mathbf{U}_1 \left|q^{(0)}\right\rangle$$

- For a QFA with a large set of states, each $\mathbf{U}_a$ becomes very large, making sequential multiplication intractable
- Imitating $\Psi(t) = \mathrm{cis}(-\omega t)$, we can rewrite

$$\mathbf{U} \quad = \quad \mathrm{cis}(-\hat{\mathbf{H}})$$

We call $\hat{\mathbf{H}}$ the Hamiltonian of $\mathbf{U}$

- Time evolution can be rewritten as

$$\left|q^{(N)}\right\rangle \quad = \quad \mathrm{cis}(-\hat{\mathbf{H}}_N) \dots \mathrm{cis}(-\hat{\mathbf{H}}_1) \left|q^{(0)}\right\rangle$$

- Let's recall that matrix multiplication is non-commutative

$$\mathbf{AB} \quad \neq \quad \mathbf{BA}$$

- Therefore:

$$\mathbf{U}_N \dots \mathbf{U}_1 \quad \neq \quad \mathrm{cis} - (\hat{\mathbf{H}}_1 + \dots + \hat{\mathbf{H}}_N)$$

because $\mathbf{U}_N \dots \mathbf{U}_1 \neq \mathbf{U}_1 \dots \mathbf{U}_N$

- Later, we can still estimate multiplication from the Hamiltonians

Conversion to Hamiltonian by Pauli Decomposition

- Step 1: If a quantum operator $\mathbf{U}$ is Hermition, we let $\mathbf{U}' = \mathbf{U}$. Otherwise, we have to appproximate its Hermitian:

$$\mathbf{U}' = \frac{1}{2}(\mathbf{U} + \mathbf{U}^*)$$

- Step 2: Enumerate all 4^n possible Pauli axes $\{\mathbf{I}, \mathbf{X}, \mathbf{Y}, \mathbf{Z}\}$, where n is the number of qubits. For example, a two-qubit operator have $4^2 = 16$ Pauli axes:

$$\mathbf{I}\otimes\mathbf{I} \quad \mathbf{I}\otimes\mathbf{X} \quad \mathbf{I}\otimes\mathbf{Y} \quad \mathbf{I}\otimes\mathbf{Z} \quad \dots \quad \mathbf{Z}\otimes\mathbf{Z}$$

- Tip: $\left(\bigotimes_{k=1}^{N} \mathbf{A}_k\right)\left(\bigotimes_{k=1}^{N} \mathbf{B}_k\right) = \bigotimes_{k=1}^{N} \mathbf{A}_k\mathbf{B}_k$

- Step 3: Compute the coefficient c_k for each Pauli axis $\mathbf{P}_k$

$$c_k = \mathrm{tr}(\mathbf{U}'\mathbf{P}_k)$$

- Step 4: The Hamiltonian of $\mathbf{U}$ is an LCU (linear combination of unitaries) of

$$\hat{\mathbf{H}} = \sum_{k=0}^{4^n-1} c_k \mathbf{P}_k$$

- These steps can be computed by `qml.pauli_decompose(mat, wire_order=...)`, where the wire order specifies the input wires

Conversion to Hamiltonian by Pauli Decomposition (cont'd)

- Example: Compute the Hamiltonian of

$$\mathbf{A} = \begin{bmatrix} 1 & 2+i \\ 3-i & 4 \end{bmatrix}$$

- Step 1: Convert $\mathbf{A}$ into its Hermitian matrix

$$\mathbf{A}' = \begin{bmatrix} 1 & 5/2+i \\ 5/2-i & 4 \end{bmatrix}$$

- Step 2: Enumerate all Pauli axes

$$\mathrm{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \mathrm{X} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\mathrm{Y} = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad \mathrm{Z} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

- Step 3: Compute each coefficient

$$\begin{aligned} c_{\mathrm{I}} &= \mathrm{tr}\left(\begin{bmatrix} 1 & 5/2+i \\ 5/2-i & 4 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right) \\ &= 1+4 = 5 \\ c_{\mathrm{X}} &= \mathrm{tr}\left(\begin{bmatrix} 1 & 5/2+i \\ 5/2-i & 4 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\right) \\ &= 5/2+i+5/2-i = 5 \\ c_{\mathrm{Y}} &= \mathrm{tr}\left(\begin{bmatrix} 1 & 5/2+i \\ 5/2-i & 4 \end{bmatrix}\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}\right) \\ &= -1+5i/2+1-5i/2 = 0 \\ c_{\mathrm{Z}} &= \mathrm{tr}\left(\begin{bmatrix} 1 & 5/2+i \\ 5/2-i & 4 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}\right) \\ &= 1-4 = -3 \end{aligned}$$

Conversion to Hamiltonian (cont'd)

- Example: Compute the Hamiltonian of

$$\mathbf{A} = \begin{bmatrix} 1 & 2+i \\ 3-i & 4 \end{bmatrix}$$

- Step 1: Convert $\mathbf{A}$ into its Hermitian matrix

$$\mathbf{A}' = \begin{bmatrix} 1 & 5/2+i \\ 5/2-i & 4 \end{bmatrix}$$

- Step 2: Enumerate all Pauli axes

$$\mathrm{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \mathrm{X} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\mathrm{Y} = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad \mathrm{Z} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

- Step 3: Compute each coefficient

$$\begin{aligned} c_{\mathrm{I}} &= 5 \\ c_{\mathrm{X}} &= 5 \\ c_{\mathrm{Y}} &= 0 \\ c_{\mathrm{Z}} &= -3 \end{aligned}$$

- Step 4: The Hamiltonian of $\mathbf{A}$ is

$$\begin{aligned} \hat{\mathrm{H}} &= 5\mathrm{I} + 5\mathrm{X} + 0\mathrm{Y} - 3\mathrm{Z} \\ &= \begin{bmatrix} 5-3 & 5 \\ 5 & 5+3 \end{bmatrix} \\ &= \begin{bmatrix} 2 & 5 \\ 5 & 8 \end{bmatrix} \end{aligned}$$

Exercise 7.2: Hamiltonian Matrix

Convert the following matrices to Hamiltonians. Note that some of them are not Hermitian.

1. $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
2. $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
3. $\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
4. $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
5. $\begin{bmatrix} 3 & 4i \\ -5i & 6 \end{bmatrix}$
6. $\mathbf{I} \otimes \mathbf{X} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
7. $\mathbf{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
8. $\mathbf{X} \otimes \mathbf{Y}$
9. $\mathbf{X} \otimes \mathbf{Y} \otimes \mathbf{Z}$
10. $\begin{bmatrix} \operatorname{cis}(-\frac{\theta}{2}) & 0 \\ 0 & \operatorname{cis}\frac{\theta}{2} \end{bmatrix}$ for any θ

Commutator

- Commutator of two matrices $\mathbf{A}$ and $\mathbf{B}$ is positional discrepancy between $\mathbf{A}$ and $\mathbf{B}$:

$$[\mathbf{A},\mathbf{B}] \quad = \quad \mathbf{AB}-\mathbf{BA}$$

- $\mathbf{A}$ and $\mathbf{B}$ are said to be commutative if and only if $[\mathbf{A},\mathbf{B}]=\mathbf{0}$; i.e. no discrepancy
- Let's define the repetitive commutator that applies s pairs of $\mathbf{B}$ and then r pairs of $\mathbf{A}$ on matrix $\mathbf{C}$:

$$[\mathbf{A}^r\mathbf{B}^s\mathbf{C}] \quad = \quad \overbrace{[\mathbf{A},[\ldots,[\mathbf{A},\ \underbrace{[\mathbf{B},[\ldots,[\mathbf{B},\mathbf{C}]]]}_{s\text{ pairs}}\ ]]]}^{r\text{ pairs}}$$

where $r+s>0$

- Example: Both operands are non-communative.

$$\begin{bmatrix}1&2\\3&4\end{bmatrix}\times\begin{bmatrix}5&6\\7&8\end{bmatrix} \quad = \quad \begin{bmatrix}19&22\\43&50\end{bmatrix}$$

$$\begin{bmatrix}5&6\\7&8\end{bmatrix}\times\begin{bmatrix}1&2\\3&4\end{bmatrix} \quad = \quad \begin{bmatrix}23&34\\31&46\end{bmatrix}$$

- Their positional discrepancy is

$$\left[\begin{bmatrix}1&2\\3&4\end{bmatrix},\begin{bmatrix}5&6\\7&8\end{bmatrix}\right] \quad = \quad \begin{bmatrix}-4&-12\\12&4\end{bmatrix}$$

- If $[\mathbf{A},\mathbf{B}]=\mathbf{0}$, then $\mathbf{AB}=\mathbf{BA}$.

Baker–Campbell–Hausdorff Expansion

- Note that swapping the order of applying $\mathbf{A}$ and $\mathbf{B}$ also results in different results
- Let's denote a combination of applying $\mathbf{A}$ and $\mathbf{B}$ with:

$$\Delta(\mathbf{A},\mathbf{B},g) = \frac{[\mathbf{A}^{r_1}\mathbf{B}^{s_1}[\ldots[\mathbf{A}^{r_{K-1}}\mathbf{B}^{s_{K-1}}[\mathbf{A}^{r_K}\mathbf{B}^{s_K}\mathbf{B}]]]]}{r_1!s_1!\ldots r_K!s_K!}$$

where $g = (r_1, s_1, \ldots, r_K, s_K)$ represents one combination of applying $\sum_{k=1}^{K} r_k$ times of $\mathbf{A}$ and $\sum_{k=1}^{K} s_k$ times of $\mathbf{B}$, and each $r_k + s_k > 0$

- Baker–Campbell–Hausdorff expansion: Commutator is the foundation for approximating

$$\log(\exp\mathbf{A}\exp\mathbf{B}) = \mathbf{A}+\mathbf{B}+\sum_{K=1}^{\infty}\frac{(-1)^{K-1}}{K}\sum_{g\in G_K}\frac{\Delta(\mathbf{A},\mathbf{B},g)}{\sum g}$$

where G_K is a set of all possible tuples $g = (r_1, s_1, \ldots, r_K, s_K)$ such that each $r_k + s_k > 0$

- In practice, we set the limit for the granularity K; i.e. the more granularity, the closer the approximation is

Exercise 7.3: Commutator

Answer the following questions

1. Show that $[\mathbf{A}, \mathbf{B}]$ is an additive inverse of $[\mathbf{B}, \mathbf{A}]$.
2. Show that if $\mathbf{A}$ is Hermitian, then $[\mathbf{A}, \mathbf{A}^*] = \mathbf{0}$.
3. Let $\mathbf{A}$ and $\mathbf{B}$ be non-Hermitian matrices. Discuss why $[\mathbf{A}, \mathbf{B}]$ reflects positional discrepancy between $\mathbf{A}$ and $\mathbf{B}$.
4. Suppose two non-Hermitian matrices $\mathbf{A}$ and $\mathbf{B}$ are converted to their Hamiltonians $\hat{\mathbf{H}}_A$ and $\hat{\mathbf{H}}_B$, respectively. Explain why $[\mathbf{A}, \mathbf{B}] \neq \mathbf{0}$.
5. Let $\mathbf{A}$ and $\mathbf{B}$ are non-Hermitian matrices, whose approximate Hamiltonians are $\hat{\mathbf{H}}_A$ and $\hat{\mathbf{H}}_B$, respectively. Show that

$$\log(\mathbf{AB}) \quad = \quad -i(\hat{\mathbf{H}}_A + \hat{\mathbf{H}}_B) + \mathbf{E}$$

where $\mathbf{E}$ is an error-term matrix.

Trotter-Suzuki Decomposition

- If $\log \mathbf{U}_1 = -i\hat{\mathbf{H}}_1$ and $\log \mathbf{U}_2 = -i\hat{\mathbf{H}}_2$, then

$$\begin{aligned} \log(\mathbf{U}_2\mathbf{U}_1) &= -i(\hat{\mathbf{H}}_1 + \hat{\mathbf{H}}_2) + \mathbf{E} \\ \therefore \quad \mathbf{U}_2\mathbf{U}_1 &= \exp\left[-i(\hat{\mathbf{H}}_1 + \hat{\mathbf{H}}_2) + \mathbf{E}\right] \end{aligned}$$

  where $\mathbf{E}$ is the error term in Baker-Campbell-Hausdorff expansion

- Trotter-Suzuki decomposition (a.k.a. trotterization): We can approximate

$$\mathbf{U}_2\mathbf{U}_1 \approx \left[\mathrm{cis}\left(-\frac{\hat{\mathbf{H}}_1 + \hat{\mathbf{H}}_2}{n}\right)\right]^n + O(1/n)$$

  Note that the error term $O(1/n)$ depends on the chosen number n

- Proof:

$$\mathbf{U}_2\mathbf{U}_1 = \lim_{n\to\infty}\left(\exp\frac{-i\hat{\mathbf{H}}_1}{n}\exp\frac{-i\hat{\mathbf{H}}_2}{n}\right)^n$$

- Since $e^x = 1 + x + O(1/n^2)$ for small x, we obtain the analogy:

$$\begin{aligned} \mathbf{U}_2\mathbf{U}_1 = \lim_{n\to\infty} &\left(\mathbf{I} - \frac{i\hat{\mathbf{H}}_1}{n} + O(1/n^2)\right)^n \\ &\times \left(\mathbf{I} - \frac{i\hat{\mathbf{H}}_2}{n} + O(1/n^2)\right)^n \end{aligned}$$

  because the Hamiltonians are divided a very large number n.

Trotter-Suzuki Decomposition (cont'd)

- We expand the RHS into

$$\begin{aligned}
\mathrm{U}_2\mathrm{U}_1 &= \lim_{n\to\infty}\left[\left(\mathrm{I}-\frac{i\hat{\mathrm{H}}_1}{n}\right)\left(\mathrm{I}-\frac{i\hat{\mathrm{H}}_2}{n}\right)\right]^n \\
&\quad +O(1/n) \\
&= \lim_{n\to\infty}\left[\mathrm{I}-\frac{i(\hat{\mathrm{H}}_1+\hat{\mathrm{H}}_2)}{n}+\frac{\hat{\mathrm{H}}_1\hat{\mathrm{H}}_2}{n^2}\right]^n \\
&\quad +O(1/n) \\
&= \lim_{n\to\infty}\left[\mathrm{I}-\frac{i(\hat{\mathrm{H}}_1+\hat{\mathrm{H}}_2)}{n}\right]^n+O(1/n) \\
&= \lim_{n\to\infty}\left[\exp\frac{-i(\hat{\mathrm{H}}_1+\hat{\mathrm{H}}_2)}{n}\right]^n+O(1/n)
\end{aligned}$$

- Generalization: If there is only one unitary operator $\mathrm{U}=\mathrm{cis}(-\hat{\mathrm{H}})$ in the time evolution of N timesteps:

$$\left|\psi^{(N)}\right\rangle = \mathrm{U}^N\left|\psi^{(0)}\right\rangle$$

we can generalize Trotter-Suzuki decomposition into

$$\mathrm{U}^N \approx \left[\mathrm{cis}\left(-\frac{N\hat{\mathrm{H}}}{n}\right)\right]^n+O(1/n)$$

- This formula is useful when the length of time evolution is predefined, because we can precompute U^N in advance

Trotter-Suzuki Decomposition in PennyLane

```
mat = np.zeros((4, 4), dtype=np.complex128)            # Markov matrix
mat[0,0] = -7/25;                  mat[1,0] = 6 * np.sqrt(3) / 10
mat[2,1] = 6 * np.sqrt(3) / 10;    mat[1,1] = 2/4
mat[2,2] = 7/25;                   mat[3,2] = 8/10
mat[2,3] = 8/10;                   mat[1,3] = -2/4

dev = qml.device('default.qubit', wires=[0,1])

def make_hamiltonian(opr):
    hermitian = 0.5 * (opr + opr.T.conj())             # Enforce Hermitian matrix
    hamiltonian = qml.pauli_decompose(hermitian, wire_order=[0,1])
    return hamiltonian

@qml.qnode(dev, shots=1000)
def evolution_circuit():
    hmlt = make_hamiltonian(mat)
    qml.TrotterProduct(hmlt, time=10.0, n=20)          # Run for 10 steps with
                                                       # granularity degree n = 100
    return qml.counts(wires=[0,1])

evolution_circuit()
# Expect result: {'00': 600, '01': 125, '10': 250, '11': 25}
# These are probable final states after 10 steps of time evolution
```

Qubitization Algorithm

- Limitation: Trotter-Suzuki decomposition can approximate $\mathbf{U}^N$, but the accuracy will reduce in a large number of timesteps
- Qubitization: Any non-unitary Hermitian matrix $\mathbf{A}$ can be converted to a unitary one $\mathbf{U}$ via block encoding:

$$\begin{aligned}\mathbf{U} &= \mathrm{BLOCK}(\mathbf{A}) \\ &= \begin{bmatrix} \mathbf{A} & \cdot \\ \cdot & \cdot \end{bmatrix}\end{aligned}$$

- We can estimate $\mathbf{U}^N |\psi\rangle$ by rotating $\mathbf{U}$ many times with QPE and measuring the distribution of destination states

- Step 1: Since $\mathbf{A}$ is Hermitian, we can perform Pauli decomposition on it to obtain the LCU

$$\mathbf{A} = c_k \mathbf{P}_k$$

where each $\mathbf{P}_k$ is the k-th Pauli axis

- Step 2: Construct the preparation operator $\mathrm{PREP}(\mathbf{A})$ that maps

$$|0\rangle^{\otimes M} \otimes |\psi\rangle \mapsto \sum_{k=0}^{K-1} |\mathbb{B}_M(k), \psi\rangle \sqrt{\frac{|c_k|}{Z}}$$

where normalizing constant $Z = \sum_{k=1}^{K} |c_k|$

Qubitization Algorithm (cont'd)

- Step 3: Construct the selection operator $\mathbf{SEL}(\mathbf{A})$ that maps

$$|\mathbb{B}_M(k), \psi\rangle \quad \mapsto \quad |\mathbb{B}_M(k)\rangle \otimes \mathbf{P}_k |\psi\rangle$$

- Step 4: Construct the block encoder:

$$\begin{aligned} & \mathbf{BLOCK}(\mathbf{A}) \\ = \; & \mathbf{PREP}^*(\mathbf{A})\, \mathbf{SEL}(\mathbf{A})\, \mathbf{PREP}(\mathbf{A}) \end{aligned}$$

and apply it to $|0, \psi\rangle$ with phase kickback:

$$\begin{aligned} & \mathbf{BLOCK}(\mathbf{A}) |0, \psi\rangle \\ = \; & \frac{\lambda}{Z} |0, \psi\rangle + |0, \psi\rangle^{\perp} \sqrt{1 - (\lambda/Z)^2} \end{aligned}$$

where $|v\rangle^{\perp}$ is perpendicular to $|v\rangle$

- Step 5: Apply QPE to measure the distribution for the destination states

$$P(q^{(2^M)}) \quad = \quad \mathbf{QPE} |0, \psi\rangle$$

- Step 6: We can deduce the phase of the Hamiltonian $\hat{\mathbf{H}}$ by

$$\theta \quad = \quad 2\pi r^* / 2^M$$

where the index $r^* = \arg\max_r P(r)$

- Step 7: We can approximate

$$\mathbf{U}^N |\psi\rangle \quad \approx \quad \mathrm{cis}(-iN\theta) |\psi\rangle$$

and the eigenvalue of $\mathbf{A}$ is $\lambda = Z \cos\theta$

Qubitization Algorithm (cont'd)

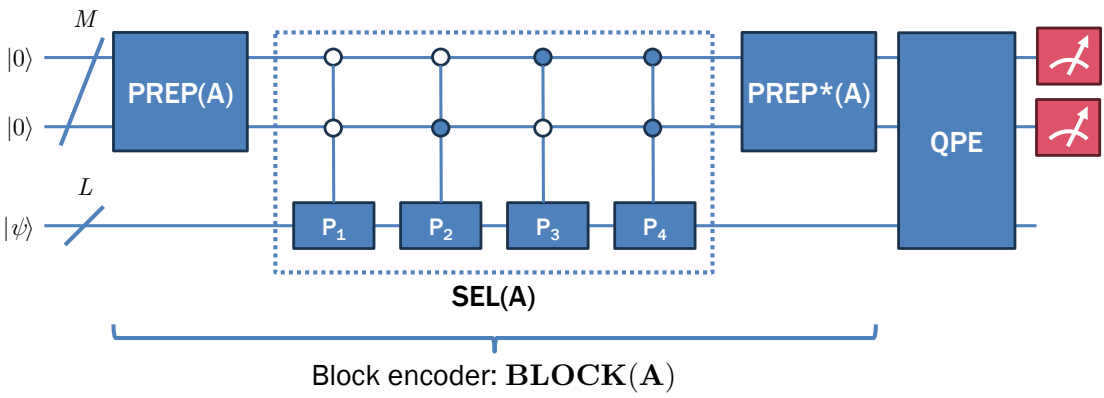


Figure 10: Qubitization algorithm. Each $\mathbf{P}_k$ is the k-th Pauli axis of the input matrix $\mathbf{A}$.

Proof via Jacobi-Anger Expansion

- Jacobi-Anger expansion explains why we can deduce the eigenvalue λ from the estimated phase θ; i.e.

$$\operatorname{cis}\theta \quad \mapsto \quad \operatorname{cis}(\cos\theta) \quad = \quad \operatorname{cis}\frac{\lambda}{Z}$$

- Jacobi-Anger expansion

$$\operatorname{cis}(z\cos\theta) \quad = \quad \sum_{k=-\infty}^{\infty} i^k J_k(z)(\operatorname{cis}\theta)^k$$

where $J_n(z)$ is the Bessel function of order n applied on z

$$J_k(z) \quad = \quad \frac{1}{\pi}\int_0^{\pi} \cos(kt - z\sin t)dt$$

- Because $\mathbf{U} = \operatorname{cis}(-\hat{\mathbf{H}})$, we can expand it with Jacobi-Anger expansion for matrix

$$\mathbf{U}^N \quad = \quad \sum_{k=-\infty}^{\infty} i^k J_k(N)(-\hat{\mathbf{H}})^k$$

- Suppose $\mathbf{U}$ has an eigen-pair $(\lambda, \mathbf{v})$ and $\hat{\mathbf{H}}$ has phase θ

$$\begin{aligned} \mathbf{U}^N\mathbf{v} &= \left(\sum_{k=-\infty}^{\infty} i^k J_k(N)\operatorname{cis}(-k\theta)\right)\mathbf{v} \\ \lambda\mathbf{v} &= \operatorname{cis}(Z\cos\theta)\mathbf{v} \\ \therefore \quad \lambda &= Z\cos\theta \end{aligned}$$

where Z is the normalizing constant

Qubitization Algorithm in PennyLane

```
import pennylane as qml
from pennylane import numpy as np

# A unitary matrix
mat = np.zeros((4, 4), dtype=np.complex128)
mat[0,0] = -7/25;                  mat[1,0] = 6 * np.sqrt(3) / 10
mat[2,1] = 6 * np.sqrt(3) / 10;    mat[1,1] = 2/4
mat[2,2] = 7/25;                   mat[3,2] = 8/10
mat[2,3] = 8/10;                   mat[1,3] = -2/4

target_wires = ['t0','t1']                          # N = input qubits
control_wires = ['c1','c2','c3','c4']               # 2^N wires
estimation_wires = ['e1','e2','e3','e4','e5','e6']  # Granularity

dev = qml.device('default.qubit', wires=target_wires + control_wires + estimation_wires)

def make_hamiltonian(opr):
    hermitian = 0.5 * (opr + opr.T.conj())
    hamiltonian = qml.pauli_decompose(hermitian, wire_order=target_wires)
    return hamiltonian
```

Qubitization Algorithm in PennyLane (cont'd)

```
hmlt = make_hamiltonian(mat)    # Convert a unitary matrix to a Hamiltonian
normconst = sum([abs(coeff) for coeff in hmlt.terms()[0]])    # Normalizing constant
print(qml.matrix(hmlt).real)    # This is how we print a Hamiltonian out in the matrix form

@qml.qnode(dev)
def evolution_circuit():
    qml.QuantumPhaseEstimation(
        qml.Qubitization(hmlt, control_wires),
        estimation_wires=estimation_wires
    )
    return qml.probs(wires=estimation_wires)

probs = evolution_circuit()
r = np.argmax(probs)
phase = 2 * np.pi * r / 2 ** len(estimation_wires)
eigval = np.cos(phase) * normconst
print(f'phase = {phase}')
print(f'eigenvalue = {eigval}')
# Expected result:
# phase = 1.7671
# eigenvalue = -0.4096
```

Exercise 7.4: Trotter-Suzuki Decomposition and Qubitization Algorithm

1. Consider the general case of trotterization

$$\mathbf{U}^N \approx \left[\text{cis}\left(-\frac{N\hat{\mathbf{H}}}{n}\right)\right]^n + O(1/n)$$

   Explain why the accuracy deteriorates when the number of timesteps N becomes large. [*Hint*: Consider $e^x = 1 + x + O(1/n^2)$ for small x.]

2. Consider the trotterization of $\mathbf{U}_2\mathbf{U}_1$:

$$\mathbf{U}_2\mathbf{U}_1 \approx \left[\text{cis}\left(-\frac{\hat{\mathbf{H}}_1 + \hat{\mathbf{H}}_2}{n}\right)\right]^n + O(1/n)$$

   Discuss why the accuracy deteriorates when both matrices are non-Hermitian.

3. What is the role of block encoding in the qubitization algorithm?
4. Explain how the qubitization algorithm addresses the issue of simulation over large timesteps N by opting for the global phase rotation of the block encoder.
5. Discuss why we have to choose the right number of estimation wires for the QPE module. What happens when M is large? What happens when M is small.
6. Compare the pros and cons of trotterization and qubitization in terms of speed, space, and programming complexities.

Polynomial Functions

Polynomial Functions

- Polynomial function is a mathematical expression consisting of coefficients and exponentiated variables, e.g.

$$P(x) \quad = \quad 4x^3 + (1+2i)x^2 - x + 7$$

The coefficients are $\Theta = (7, -1, 1+2i, 4)$

- The degree of a polynomial function is the greatest exponentiation

$$\deg P(x) \quad = \quad 3$$

- Polynomial function is fundamental for modeling a non-linear behavior of a complex system, where each variable represents a stimulus

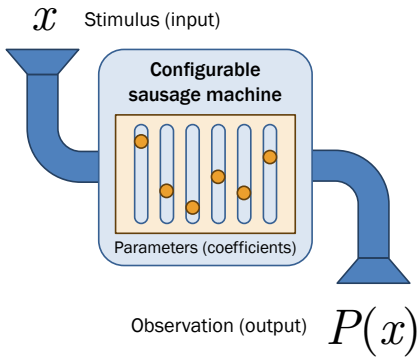


Figure 11: Polynomial function is sometimes seen as a configurable sausage machine. The parameters (i.e. adjustable setting) are tuned to alter the sausage type it is producing. Here, the parameters are the coefficients of the polynomial function.

Quantum Signal Processing (QSP)

- Quantum signal processing (QSP) is a technique for computing a polynomial function from an input qubit
- Let's define the Z-rotation operator

$$\mathrm{RZ}(\theta) = \begin{bmatrix} \mathrm{cis}(\theta) & 0 \\ 0 & \mathrm{cis}(-\theta) \end{bmatrix}$$

and a signal rotation operator

$$\mathrm{W}(a) = \begin{bmatrix} a & i\sqrt{1-a^2} \\ i\sqrt{1-a^2} & a \end{bmatrix}$$

where θ is a phase parameter and a is a diagonal parameter

- We can decompose a time series into a QSP operator that consists of several components:

$$\mathrm{QSP}(\Theta, a) = \mathrm{RZ}(\theta_0) \prod_{k=1}^{D} \mathrm{W}(a)\mathrm{RZ}(\theta_k)$$

where D is the degree, $\Theta = (\theta_0, \ldots, \theta_D)$ is a set of phase parameters, and a is a diagonal parameter

- If we adjust paramters Θ and a, we can estimate the original time series
- In function fitting, we optimize Θ and a to minimize the prediction loss, e.g. mean squared error (MSE)

QSP as Polynomial Function

- By letting $b = i\sqrt{1-a^2}$, we have

$$\mathbf{W}(a)\mathbf{RZ}(\theta) = \begin{bmatrix} a\,\mathrm{cis}(\theta) & b\,\mathrm{cis}(\theta) \\ b\,\mathrm{cis}(-\theta) & a\,\mathrm{cis}(-\theta) \end{bmatrix}$$

- It follows that the polynomial degree 2 is

$$\mathbf{W}(a)\mathbf{RZ}(\theta_1)\mathbf{W}(a)\mathbf{RZ}(\theta_2)$$

$$= \begin{bmatrix} a^2\,\mathrm{cis}(\theta_1+\theta_2) & ab\,\mathrm{cis}(\theta_1+\theta_2) \\ \quad +b^2\,\mathrm{cis}(\theta_1-\theta_2) & \quad +ab\,\mathrm{cis}(\theta_1-\theta_2) \\ ab\,\mathrm{cis}(-\theta_1+\theta_2) & b^2\,\mathrm{cis}(-\theta_1+\theta_2) \\ \quad +ab\,\mathrm{cis}(-\theta_1-\theta_2) & \quad +a^2\,\mathrm{cis}(-\theta_1-\theta_2) \end{bmatrix}$$

Observe that crossing elements are the conjugate of each other

- Generalization: QSP of degree D has a polynomial form once we expand the multiplications:

$$\mathbf{QSP}(\Theta, a) = \begin{bmatrix} P(a) & bQ(a) \\ bQ^*(a) & P^*(a) \end{bmatrix}$$

where $a \in [-1,1]$ and $b = i\sqrt{1-a^2}$

- $P(a)$ and $Q(a)$ are polynomial functions that satisfy the following constraints
  - $\deg P(a) \le D$ and $\deg Q(a) \le D-1$
  - $P(a)$ has parity of $D\%2$, while $Q(a)$ has parity of $(D-1)\%2$
  - $|P(a)|^2 + (1-a)^2|Q(a)|^2 = 1$

Polynomial Function of Unitary Operators

- Generalized QSP: We treat the input signal as a control qubit, resulting in a block encoding of $\mathrm{poly}(U)$:

$$U \mapsto \begin{bmatrix} \mathrm{poly}(U) & \cdot \\ \cdot & \cdot \end{bmatrix}$$

- PennyLane: `qml.GQSP(opr=U, angles=..., control=...)`
- We have to convert the polynomial coefficients into angles by the method `qml.poly_to_angles(coeffs, 'GQSP')`

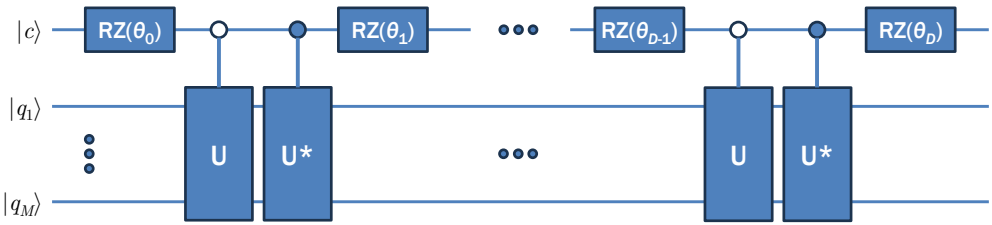


Figure 12: Generalized quantum signal processing

Quantum Signal Processing in PennyLane

```
control_wires = ['c']
target_wires = ['t1', 't2', 't3']
dev = qml.device('default.qubit', wires=target_wires + control_wires)

@qml.prod        # Make this function an operator argument
def my_opr(wires):
    qml.Hadamard(wires=wires[0])
    qml.X(wires=wires[2])
    qml.CNOT(wires=[wires[0], wires[1]])
    qml.CNOT(wires=[wires[0], wires[2]])

@qml.qnode(dev, shots=1000)
def circuit():
    coeffs = np.array([0.1, 0.2j, 0.3])                    # 0.1 + 0.2j x + 0.3 x^2
    phases = qml.poly_to_angles(coeffs, routine='GQSP')    # Convert the coefficients to phases
    qml.GQSP(my_opr(target_wires), phases, control=control_wires)
    return qml.counts(wires=target_wires)

circuit()
# Expected result:
# {'000': 268, '001': 23, '011': 243, '100': 202, '110': 21, '111': 243}
```

Quantum Singular Value Transformation (QSVT)

- Quantum singular value transformation (QSVT) computes a polynomial function of a Hamiltonian
- Suppose we have $\mathbf{A} = \mathrm{cis}(-\hat{\mathbf{H}})$, where

$$\hat{\mathbf{H}} = \sum_{k=1}^{M} \lambda_k |v_k\rangle \langle v_k|$$

where each $(\lambda_k, |v_k\rangle)$ is an eigen-pair of $\mathbf{A}$

- QSVT transforms $\mathbf{A}$ into

$$\mathbf{A} \mapsto \mathrm{cis}(-\mathrm{poly}(\hat{\mathbf{H}}))$$

where $\mathrm{poly}(\hat{\mathbf{H}})$ is a polynomial function, e.g. $P(\hat{\mathbf{H}}) = 4\hat{\mathbf{H}}^2 + 3i\hat{\mathbf{H}} - 7$

- Imitating QSP, we convert $\hat{\mathbf{H}}$ to a unitary operator

$$\mathbf{U} = \begin{bmatrix} \hat{\mathbf{H}} & \sqrt{1-\hat{\mathbf{H}}^2} \\ \sqrt{1-\hat{\mathbf{H}}^2} & -\hat{\mathbf{H}} \end{bmatrix}$$

where

$$\sqrt{1-\hat{\mathbf{H}}^2} = \sum_{k=1}^{M} |v_k\rangle \langle v_k| \sqrt{1-\lambda_k^2}$$

- Now we intend to apply the block encoding on the operator $\mathbf{U}$:

$$\mathbf{U} \mapsto \begin{bmatrix} \mathrm{poly}(\mathbf{U}) & \cdot \\ \cdot & \cdot \end{bmatrix}$$

Backbone of QSVT

- We observe that

$$\mathbf{RZ}(\theta_k)\ \mathbf{U} = \begin{bmatrix} \mathrm{cis}(\theta_k)\hat{\mathbf{H}} & \mathrm{cis}(\theta_k)\sqrt{1-\hat{\mathbf{H}}^2} \\ \mathrm{cis}(-\theta_k)\sqrt{1-\hat{\mathbf{H}}^2} & -\mathrm{cis}(-\theta_k)\hat{\mathbf{H}} \end{bmatrix}$$

$$\mathbf{RZ}(\theta_k)\ \mathbf{U}^* = \begin{bmatrix} \mathrm{cis}(\theta_k)\hat{\mathbf{H}} & \mathrm{cis}(\theta_k)\sqrt{1-\hat{\mathbf{H}}^2} \\ \mathrm{cis}(-\theta_k)\sqrt{1-\hat{\mathbf{H}}^2} & -\mathrm{cis}(-\theta_k)\hat{\mathbf{H}} \end{bmatrix}$$

- Therefore,

$$\mathbf{RZ}(\theta_{k+1})\ \mathbf{U}^*\ \mathbf{RZ}(\theta_k)\ \mathbf{U} = \begin{bmatrix} \mathbf{poly}(\hat{\mathbf{H}}) & \cdot \\ \cdot & \cdot \end{bmatrix}$$

- We compose a polynomial function by alternatingly multiplying $\mathbf{RZ}(\theta_{2k})\ \mathbf{U}$ and $\mathbf{RZ}(\theta_{2k+1})\ \mathbf{U}^*$ until arriving at θ_D:

$$\mathbf{QSVT}(\hat{\mathbf{H}}|\Theta) = \prod_{k=0}^{\lfloor D/2 \rfloor - 1} \mathbf{RZ}(\theta_{2k+1})\ \mathbf{U}^*\ \mathbf{RZ}(\theta_{2k})\ \mathbf{U} = \begin{bmatrix} \mathbf{poly}(\hat{\mathbf{H}}) & \cdot \\ \cdot & \cdot \end{bmatrix}$$

where the polynomial function of $\hat{\mathbf{H}}$ has the set of coefficients $\Theta = (\theta_0, \ldots, \theta_D)$

Implementation of QSVT

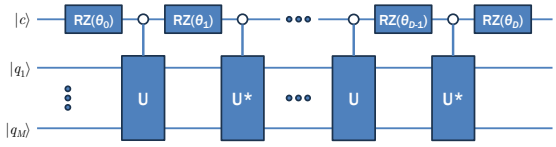


Figure 13: Quantum singular value transformation

- PennyLane: `qml.QSVT(opr, projectors)`
- The operator `opr` is constructed by `qml.BlockEncode(U, wires=...)`
- The projectors are provided by converting each phase $\theta_k \in \Theta$ into a projector-controlled phase gate `qml.PCPhase(`θ_k`, dim, wires=...)` that maps

$$\begin{aligned} |1\rangle^{\otimes M} &\mapsto \operatorname{cis}(-\theta_k)\,|1\rangle^{\otimes M} \\ |x\rangle &\mapsto \operatorname{cis}(\theta_k)\,|x\rangle \end{aligned}$$

- Note that the projectors can now be generalized beyond $\mathbf{RZ}$

Quantum Singular Value Transformation in PennyLane

```
target_wires = ['t1','t2']
dev = qml.device('default.qubit', wires=target_wires)

mat = np.array([ [0.7, 0.5],
                 [0.5, 0.4] ], dtype=np.complex128)

@qml.qnode(dev, shots=1000)
def circuit():
    coeffs = np.array([0.1, 0.2, 0.3])                # 0.1 + 0.2j x + 0.3 x^2
    block_encoding = qml.BlockEncode(mat, target_wires)
    projectors = [ qml.PCPhase(phase, dim=2, wires=target_wires)
                   for phase in coeffs ]
    qml.QSVT(block_encoding, projectors)
    return qml.counts(wires=target_wires)

circuit()
# Expected result:
# '00': 957, '01': 19, '10': 20, '11': 4}
```

Exercise 7.5: Polynomial Functions

Answer the following questions.

1. What are differences between quantum signal processing (QSP) and quantum singular value transformation (QSVT)?
2. In quantum signal processing, explain why we need both the Z-rotation operator $\mathbf{RZ}(\theta)$ and the signal rotation operator $\mathbf{W}(a)$. What do they contribute to the polynomial function?
3. Derive $\mathbf{QSP}((\theta_0, \theta_1, \theta_2, \theta_3), a)$. The result should be a 3-degree polynomial function.
4. Discuss how we can apply quantum singular value transformation (QSVT) in trotterization.
5. Show that we can eigen-decompose $\mathbf{A}$ into eigen-pairs $(\lambda_1, |v_1\rangle), \ldots, (\lambda_M, |v_M\rangle)$. Show that if $\mathbf{A} = \text{cis}(-\hat{\mathbf{H}})$, then

$$\hat{\mathbf{H}} = \sum_{k=1}^{M} \lambda_k |v_k\rangle \langle v_k|$$

HHL Algorithm

Linear Equation Systems

- Linear equation system (LES) is a collection of two or more equations involving the same set of variables

$$\begin{aligned} a_{1,1}x_1 + \ldots + a_{1,N}x_N &= b_1 \\ \vdots &\quad \vdots \\ a_{M,1}x_1 + \ldots + a_{M,N}x_N &= b_M \end{aligned}$$

  where each $a_{i,j}$ is a coefficient
- An LES can be rewritten in a matrix form

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

  where $\mathbf{A}$ is a coefficient matrix and we want to solve this LES to obtain $\mathbf{x}$

- There are three ways to solve the LES:
  - Matrix inverse: We compute the inverse of $\mathbf{A}$ and solve for $\mathbf{x}$ by

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

    But not all matrices are invertible
  - Gaussian elimination: We manually solve the LES with Gaussian elimination in $O(N^3)$ time complexity
  - Eigen-decomposition: We find the dominant eigenvalue λ of $\mathbf{A}$, so that

$$\begin{aligned} \mathbf{A}^{-1}\mathbf{v} &= \mathbf{v}/\lambda \\ \mathbf{A}^{-1}(\mathbf{A}\mathbf{x}) &= (\mathbf{A}\mathbf{x})/\lambda \\ \therefore \quad \mathbf{x} &= \mathbf{b}/\lambda \end{aligned}$$

Solving LES with Quantum Computing

- For the sake of speed improvement via quantum computing, we hope to convert $\mathbf{A}$ to a Hermitian matrix and $\mathbf{b}$ to a quantum state
- Hermitian embedding: We can construct a Hermitian matrix (i.e. $\mathbf{A} = \mathbf{A}^*$) by incorporating it into the Hermitian block

$$\hat{\mathbf{A}} = \begin{bmatrix} 0 & \mathbf{A} \\ \mathbf{A}^* & 0 \end{bmatrix}$$

- LES $\mathbf{Ax} = \mathbf{b}$ can be rewritten as

$$\hat{\mathbf{A}} \begin{bmatrix} 0 \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}$$

- Block encoding: We can also construct a polynomial block out of $\mathbf{A}$:

$$\hat{\mathbf{A}} = \begin{bmatrix} \mathrm{poly}(\mathbf{A}) & \cdot \\ \cdot & \cdot \end{bmatrix}$$

and encoding $\mathbf{b}$ as an amplitude vector

$$\hat{\mathbf{b}} = \sum_{k=0}^{2^N-1} b_k \, |\mathbb{B}_L(k)\rangle$$

- LES $\mathbf{Ax} = \mathbf{b}$ can be rewritten as

$$\hat{\mathbf{A}}\mathbf{x} = \hat{\mathbf{b}}$$

- Block encoding is numerically more stable than Hermitian embedding

Harrow-Hassidim-Lloyd Algorithm (HHL)

- HHL Algorithm solves a LES of N equations and N variables using QPE
- A Hermitian matrix $\hat{\mathbf{A}}$ can be rewritten as eigen-pairs $(\lambda_k, |v_k\rangle)$:

$$\mathbf{A} = \sum_{k=0}^{N-1} \lambda_k |v_k\rangle \langle v_k|$$

- And we can rewrite $\mathbf{b}$ and $\mathbf{x}$ in terms of the eigenvectors $|v_k\rangle$ as follows

$$|b\rangle = \frac{1}{|\mathbf{b}|} \sum_{k=0}^{N-1} b_k |\mathbb{B}_L(k)\rangle = \sum_{k=0}^{N-1} \beta_k |v_k\rangle$$

$$|x\rangle = \frac{1}{|\mathbf{x}|} \sum_{k=0}^{N-1} x_k |\mathbb{B}_L(k)\rangle = \sum_{k=0}^{N-1} \gamma_k |v_k\rangle$$

- Since $\hat{\mathbf{A}}$ may not be unitary, we convert it to one using block encoding:

$$\mathbf{U} = \mathbf{BLOCK}(\mathbf{A})$$

- Step 1: We initialize the states.

$$|\psi_1\rangle = |0\rangle \otimes \left(\sum_{k=0}^{N-1} \beta_k |v_k\rangle \right) \otimes |0\rangle^{\otimes L} \otimes |0\rangle$$

- Step 2: We extract the eigenvalues.

$$|\psi_2\rangle = \mathbf{QPE}(\mathbf{U})^{(3)}_{1,2} |\psi_1\rangle = \left(\sum_{k=0}^{N-1} \beta_k |0, v_k\rangle \otimes |\lambda_k\rangle \right) \otimes |0\rangle$$

HHL Algorithm (cont'd)

- Step 3: Compute the filter flag of λ_k by applying conditional R-rotation on the last qubit conditioned by each qubit of $|\lambda_k\rangle$.

$$\begin{aligned} |\psi_3\rangle &= \prod_{j=1}^{L} \mathrm{CY}^{(3)}_{2@3,3} |\psi_2\rangle \\ &= \sum_{k=0}^{N-1} \beta_k |0, v_k\rangle \otimes |\lambda_k\rangle \otimes |c_k\rangle \end{aligned}$$

where $|c_k\rangle$ is a filter flag of λ_k:

$$|c_k\rangle = \sqrt{1 - \frac{C}{\lambda_k^2}} |0\rangle + \frac{C}{\lambda_k} |1\rangle$$

and C is a normalizing constant.

- Step 4: We disentangle the eigenvalues from the input by running the inverse QPE

$$\begin{aligned} |\psi_4\rangle &= \mathbf{QPE}^*(\mathbf{U})^{(3)}_{1,2} |\psi_3\rangle \\ &= \sum_{k=0}^{N-1} \beta_k |0, v_k\rangle \otimes |0\rangle^{\otimes L} \otimes |c_k\rangle \end{aligned}$$

- Step 5: Recall that we want to estimate $\mathbf{x} \approx \sum_{k=1}^{N} \mathbf{b}/\lambda_k$. We measure the last qubit

$$c \sim \left\langle \mathbf{X}^{(\psi_{4@4})} \right\rangle$$

We post-select the result only when $c = 1$. That is, if $c = 0$, we go back to Step 1 and resample $|\psi_4\rangle$.

HHL Algorithm (cont'd)

- Step 6: After some iterations, we obtain

$$|\psi_5\rangle \quad = \quad \left(\sum_{k=0}^{N-1} \frac{\beta_k}{\lambda_k} |0, v_k\rangle\right) \otimes |0\rangle^{\otimes L} \otimes |1\rangle$$

- Step 7: For the solution of the LES, we can measure its first L qubits

$$\mathbf{x} \quad \approx \quad \sum_{k=0}^{N-1} \frac{\beta_k}{\lambda_k} |v_k\rangle$$

- Time complexity for HHL Algorithm is $O(\kappa^2 \log N)$, where N is the number of variables and $\kappa = \lambda_{\max}/\lambda_{\min}$ is the condition number of matrix $\mathbf{A}$

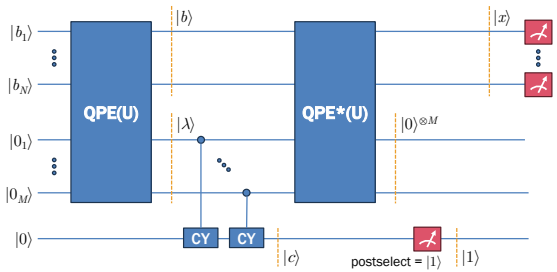


Figure 14: Harrow-Hassidim-Lloyd algorithm solves a linear equation system $\mathbf{Ax} = \mathbf{b}$. Time complexity is $O(\kappa^2 \log N)$, where N is the number of variables and $\kappa = \lambda_{\max}/\lambda_{\min}$ is the condition number of $\mathbf{A}$.

Implementation of HHL Algorithm in PennyLane

```
# Linear equation system:
#  2a + 3b - 4c + 5d = 70
# -3a +  b + 2c + 0d = 50
#   a + 2b + 3c +  d = 60
#  2a + 2b + 2c + 3d = 40
mat = np.array([ [ 2, 3, -4, 5],
                 [-3, 1,  2, 0],
                 [ 1, 2,  3, 1],
                 [ 2, 2,  2, 3] ], dtype=np.complex128)
bvec = np.array([70, 50, 60, 40], dtype=np.complex128)

extra_wires = ['x']
target_wires = ['t0','t1']
estimation_wires = ['e1','e2','e3','e4','e5']
check_wires = ['f']

dev = qml.device('default.qubit',
                 wires=extra_wires + target_wires + estimation_wires + check_wires)

def normalize_les(mat, bvec):
    norm_a = np.max(np.linalg.svd(mat)[1])
    norm_b = np.linalg.norm(bvec)
    return mat / norm_a, bvec / norm_b
```

Implementation of HHL Algorithm in PennyLane (cont'd)

```
@qml.qnode(dev, shots=1000)
def hhl_circuit(mat, bvec):
    mat, bvec = normalize_les(mat, bvec)                # Normalize A and b
    qml.AmplitudeEmbedding(bvec, wires=target_wires)    # Initialize the circuit with b
    qml.QuantumPhaseEstimation(                         # Extract eigenvalues with block encoding
        qml.BlockEncode(mat, wires=extra_wires + target_wires),
        estimation_wires=estimation_wires               # We need an extra qubit for block encoding
    )
    qml.ControlledSequence(                             # Compute the filter flag and the
        qml.Y(wires=check_wires),                       # inverse of eigenvalues
        control=estimation_wires
    )
    qml.adjoint(qml.QuantumPhaseEstimation)(            # Clean up the qubitization
        qml.BlockEncode(mat, wires=extra_wires + target_wires),
        estimation_wires=estimation_wires
    )
    m = qml.measure(wires=check_wires, postselect=1)    # Redo if the check flag is 0
    return qml.probs(wires=target_wires)

hhl_circuit(mat, bvec)
# Solution is a unit vector for each variable
# Result: [0.0, 0.40, 0.25, 0.35 ]  ->  a = 0, b = 0.4, c = 0.25, d = 0.35
```

Exercise 7.6: HHL Algorithm

Answer the following questions.

1. We learned that one extra qubit is needed for accommodating the block encoding. Compare the pros and cons of such practice against Hermitian embedding.
2. Suppose we have an LES $\mathbf{Ax} = \mathbf{b}$, and $\mathbf{A}$ is eigen-decomposed to

$$\mathbf{A} = \sum_{k=0}^{N-1} \lambda_k |v_k\rangle \langle v_k|$$

   Explain why we use QPE in the HHL Algorithm and why we can extract the eigenvalues λ_k with it.
3. Discuss why we have to disentangle the eigenvalues from the input using the inverse QPE. What will happen if we do not do that?
4. Consider the filter flag in Step 3

$$|c_k\rangle = \sqrt{1 - \frac{C}{\lambda_k^2}} |0\rangle + \frac{C}{\lambda_k} |1\rangle$$

   where C is a normalizing constant. Explain why we have to post-select $c_k = 1$, ignoring the condition of $c_k = 0$.

Conclusion

Conclusion

- System dynamics is a math model for representing a non-linear behavior of a complex system
- Quantum finite-state automaton represents such non-linear behavior, and its state transition is denoted by the quantum walk operator
- Quantum simulation of system dynamics yields a path lattice, and we can observe its paths by measurement
- However, quantum simulation is expensive; for short simulation, we trotterize the walk operator to estimate the final results
- For a long simulation, trotterization is inaccurate; we qubitize the walk operator instead
- One application of simulation is solving a linear equation system (LES), which reduces the time complexity from classical $O(N^3)$ to quantum $O(\kappa^2 \log N)$

Questions?

Target Learning Outcomes/1

- Quantum Finite-State Automaton (QFA)
  1. Contrast deterministic, probabilistic, and quantum finite-state automata in terms of their state transitions and mathematical definitions.
  2. Apply Szegedy's Algorithm to convert a Probabilistic Finite-State Automaton (PFA) into its quantum equivalent (QFA) by constructing the necessary diffusion and reflection matrices .
  3. Implement a QFA circuit using PennyLane to simulate state transitions for a given input sequence.
- Quantum System Dynamics and Simulation
  1. Define the Hamiltonian of a quantum walk operator and explain how it simplifies complex matrix multiplications into additions.
  2. Execute the conversion of a quantum operator into a Hamiltonian using the Pauli decomposition method (Linear Combination of Unitaries).
  3. Evaluate the trade-offs between Trotter-Suzuki Decomposition for short-term simulations and the Qubitization Algorithm for high-accuracy, long-term simulations.

Target Learning Outcomes/2

- Polynomial Functions and Signal Processing
  1. Explain the role of Quantum Signal Processing (QSP) in computing polynomial functions through signal and rotation operators.
  2. Analyze the Quantum Singular Value Transformation (QSVT) as a method to transform a Hamiltonian via a polynomial function.
  3. Construct quantum circuits that implement Generalized QSP (GQSP) and QSVT using block encoding and projector-controlled phase gates.
- HHL Algorithm for Linear Systems
  1. Formulate a Linear Equation System (LES) in a format suitable for quantum computation using Hermitian embedding or block encoding.
  2. Illustrate the step-by-step mechanics of the Harrow-Hassidim-Lloyd (HHL) algorithm, including eigenvalue extraction via QPE, conditional rotation, and post-selection.
  3. Compare the time complexity of the classical Gaussian elimination against the quantum HHL algorithm.