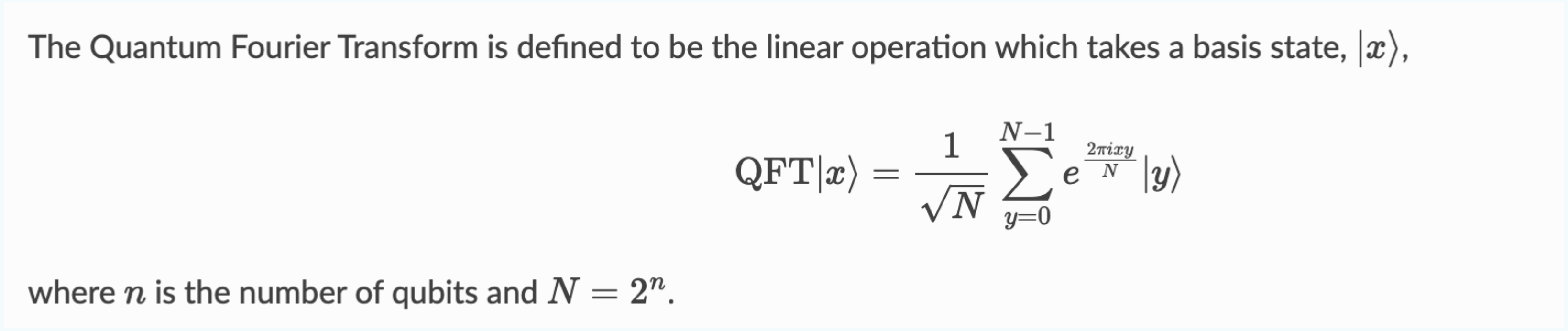
Quantum Fourier Transform (QFT) Hands-On Session

Interactive learning of quantum computing algorithms

Foundations of Quantum Fourier Transform

Complexity Comparison: Classical FFT vs QFT

- The classical FFT requires $O(N \log N)$ operations, where N is the number of data points.
- The QFT acts on n qubits, where $N = 2^n$, giving $n = \log_2(N)$.
- This leads to a quantum gate complexity of $O(n^2) = O((\log N)^2)$.
- A full comparison ultimately depends on a hardware-dependent constant relating quantum gate operations to classical floating-point operations. For sufficiently large N, the QFT provides an asymptotic advantage – despite the fact that we cannot directly access all amplitudes as in a classical FFT.



The Quantum Fourier Transform is defined to be the linear operation which takes a basis state, $|x\rangle$,

$$\mathrm{QFT}|x\rangle = \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} e^{\frac{2\pi i x y}{N}} |y\rangle$$

where n is the number of qubits and $N = 2^n$.

One-Qubit QFT Behavior

- QFT on Single Qubit
  - The one-qubit QFT transforms $|0\rangle$ and $|1\rangle$ into equal superpositions with distinct relative phases.
- Hadamard Gate Equivalence
  - One-qubit QFT is effectively implemented by a single Hadamard gate creating balanced superpositions.
- Importance of Phase
  - Phase differences in QFT dictate quantum interference essential for quantum computing speedup.
- Foundations for Multi-Qubit QFT
  - Understanding one-qubit QFT prepares learners for complex multi-qubit QFT involving controlled phase gates.

It is convenient to represent states like $|x\rangle = |101\rangle$ in their binary representation which in this case would be $2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = |5\rangle$.

For the $n = 1$ qubit case, we have the following basis states: $|0\rangle, |1\rangle$. The QFT on these is as follows:

$$\begin{aligned} \mathrm{QFT}|0\rangle &= \frac{1}{\sqrt{2}} \sum_{y=0}^{2-1} e^{\frac{2\pi i(0)y}{2}} |y\rangle \\ &= \frac{1}{\sqrt{2}} \left(e^0|0\rangle + e^0|1\rangle\right) \\ &= \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle) \\ &= |+\rangle \end{aligned} \tag{2}$$

$$\begin{aligned} \mathrm{QFT}|1\rangle &= \frac{1}{\sqrt{2}} \sum_{y=0}^{2-1} e^{\frac{2\pi i(1)y}{2}} |y\rangle \\ &= \frac{1}{\sqrt{2}} \left(e^0|0\rangle + e^{\pi i}|1\rangle\right) \\ &= \frac{1}{\sqrt{2}} (|0\rangle - |1\rangle) \\ &= |-\rangle \end{aligned} \tag{3}$$

To summarize, $\mathrm{QFT}|0\rangle = |+\rangle$ and $\mathrm{QFT}|1\rangle = |-\rangle$ which can be achieved by applying Hadamard gates.

Multi-Qubit QFT and Circuit Structure

General Multi-Qubit QFT Mechanics

- QFT Gate Structure
    - Multi-qubit QFT uses Hadamard and controlled-R_k gates applying fractional phase shifts in a recursive pattern.
- Phase Encoding and Superposition
    - QFT encodes complex fractional phase rotations creating superpositions with intricate amplitude and phase relationships.
- Circuit Complexity and Optimization
    - Full QFT requires $O(n^2)$ gates; approximations reduce minor rotations to optimize near-term quantum hardware use.
- Educational and Practical Insights
    - Studying multi-qubit QFT helps understand hierarchical design, gate parameters, and quantum coherence challenges.

Let us now extend this and see how we can apply this to $n = 3$ qubits.

For $n = 3$, $N = 2^3 = 8$ and we would like to perform $\mathrm{QFT}|101\rangle$ which in binary is $2^2 \times 1 + 2^1 \times 0 + 2^0 \times 1 = 5$:

$$\begin{aligned} \mathrm{QFT}|101\rangle = \mathrm{QFT}|5\rangle &= \frac{1}{\sqrt{8}} \sum_{y=0}^{8-1} e^{\frac{2\pi i(5)y}{8}} |y\rangle \\ &= \frac{1}{\sqrt{8}} [e^0|0\rangle + e^{\frac{5i\pi(1)}{4}}|1\rangle + e^{\frac{5i\pi(2)}{4}}|2\rangle + \ldots + + e^{\frac{5i\pi(7)}{4}}|7\rangle] \\ &= \frac{1}{\sqrt{8}} [|000\rangle + e^{\frac{5i\pi(1)}{4}}|001\rangle + e^{\frac{5i\pi(2)}{4}}|010\rangle + \ldots + + e^{\frac{5i\pi(7)}{4}}|111\rangle] \\ &= (0.35 + 0i)|000\rangle + (-0.25 - 0.25i)|001\rangle + (0 + 0.35i)|010\rangle + \ldots + (-0.25 + 0.25i)|111\rangle \end{aligned} \tag{4}$$

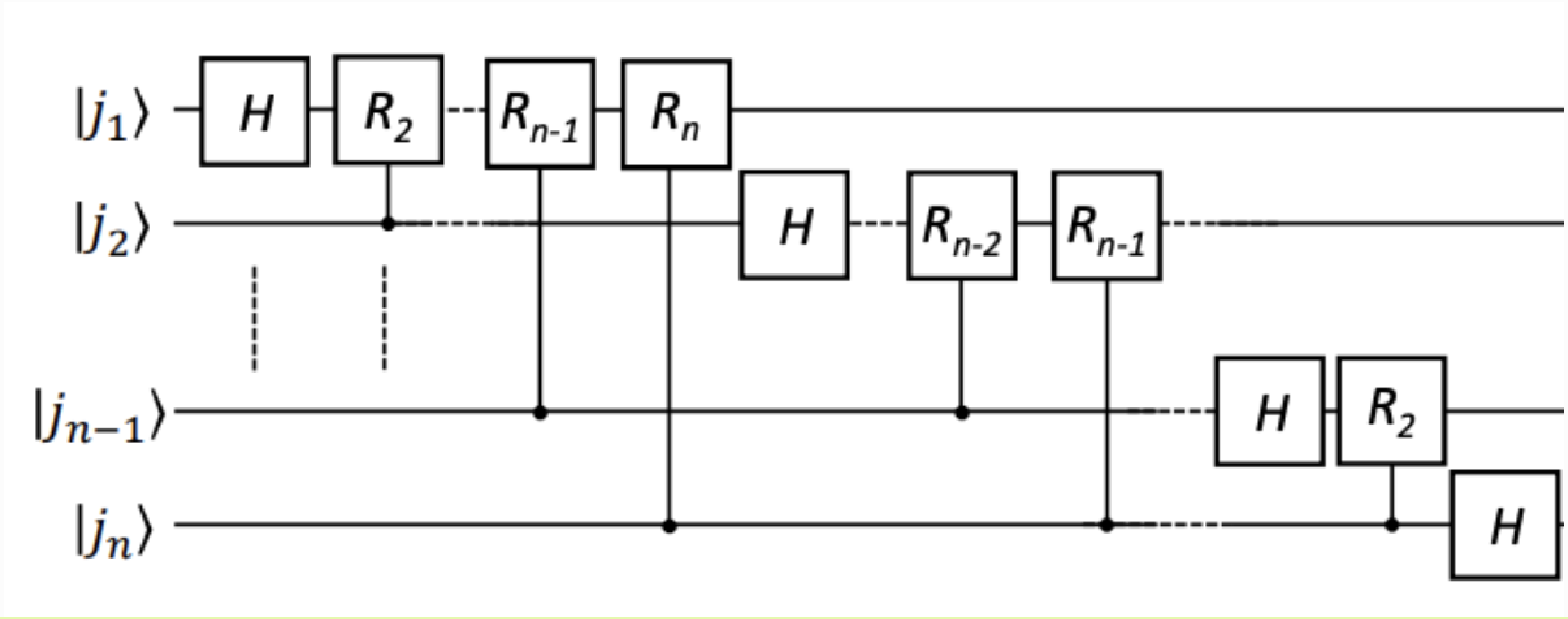
where we have used the fact that $e^{i\theta} = \cos(\theta) + i\sin(\theta)$.

Here, we see how in addition to the Hadamard gates, we need some phase gates to apply the appropriate phases to the basis states. It turns out that the gate we need is the controlled R_k rotation where the R_k gate is denoted by:

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix} \quad (5)$$

Here is a generalized ciruit for a n-qubit quantum Fourier transform:

Here is a generalized ciruit for a n-qubit quantum Fourier transform:



QFT Circuit

CUDA-Q Implementation of QFT

Programming QFT in CUDA-Q

- Quantum Kernel Definition
  - CUDA-Q uses @cudaq.kernel in Python to define quantum kernels for implementing QFT on input state vectors.
- Circuit Initialization
  - Qubits are initialized to computational basis states by applying X gates conditioned on input bits equal to one.
- QFT Logic Implementation
  - Hadamard gates and controlled-R_k phase rotations are applied in nested loops to execute the QFT algorithm.
- Learning and Experimentation
  - Students experiment with gate modifications to understand quantum amplitudes, phase shifts, and algorithm precision.

Example : QCRG website-> Courses -> Nvidia Course -> Examples -> Applications -> QFT

QFT Application Example in the Cuda-Q Webpage

- For students with Linux laptops the install is simply:
  - python3 –m venv cudaq-env
  - source cudaq-env/bin/activate
  - Pip install cudaq

- Fore MacOS and Windows do Qiskit as in the following slides.

Qiskit Implementation of QFT

Qiskit on Mac

- python3 -m venv qiskit-env
- source qiskit-env/bin/activate
- pip install qiskit
- python3 -c "import qiskit; print(qiskit.version)”
- pip install qiskit-aer
- pip install jupyterlab
- jupyter lab
- https://github.com/samaraseeri/QFT/blob/main/qiskit

CUDA-Q vs Qiskit: QFT Implementation

Concept	CUDA-Q	Qiskit
Circuit Init	cudaq.qvector(n)	QuantumCircuit(n)
Applying Gates	x(qubits[i])	qc.x(i)
Controlled Phase	cr1(angle, [ctrl], tgt)	qc.cp(angle, ctrl, tgt)
Draw Circuit	cudaq.draw(...)	qc.draw('text')
Get Statevector	cudaq.get_state(...)	Statevector.from_instruction(qc)
Simulation Target	cudaq.set_target("qpp-cpu")	Built-in CPU simulation (no setup needed)