**QUANTUM DATASETS DOCUMENT**

* Kavya Ramatatagari

# **Classical Computing & Its Disadvantages**

Classical computing relies on **binary bits (0s and 1s)** to process information, using logical operations and deterministic algorithms. It powers everything from smartphones to supercomputers but has limitations.

## **Disadvantages of Classical Computing**

1. **Limited Parallelism:** Classical computers process tasks sequentially or with limited parallelism.
2. **Exponential Growth in Complexity:** Problems like factorizing large numbers become infeasible as input size increases.
3. **Energy & Hardware Constraints:** High computational power demands more energy and heat dissipation.
4. **Inefficiency in Certain Problems:** Tasks like simulating molecules or optimization problems are inefficient on classical machines.

So, we are delving into quantum computing.

# **Quantum Computing**

Quantum computing leverages **quantum mechanics** principles to process information in a fundamentally different way, using **qubits** instead of classical bits.

**Key Concepts in Quantum Computing**

1. **Qubits & Superposition:** Unlike classical bits, qubits exist in multiple statessimultaneously (0 and 1 at the same time).
2. **Entanglement:** Qubits can be strongly correlated, meaning the state of one qubit affects another instantly, enabling faster computations.
3. **Quantum Parallelism:** Due to superposition, quantum computers can process many possibilities at once, making them powerful for optimization and simulation tasks.
4. **Quantum Gates & Circuits:** Quantum logic gates (like Hadamard, CNOT, and Toffoli) manipulate qubits to perform complex calculations.

## **Superposition:**

Superposition means a **qubit can be in multiple states at once**.

Instead of just being |0⟩ or |1⟩, a qubit can be in a combination:

Superposition is denoted by ∣ψ⟩=α∣0⟩+β∣1⟩,

Where:

* |ψ⟩ is the quantum state,
* α and β are complex numbers called **amplitudes**,
* The probabilities must add up:

∣α∣2+∣β∣2=1

**Multi-Qubit Superposition:**

For 2 qubits, a general superposition is:

|ψ⟩ = α₀₀|00⟩ + α₀₁|01⟩ + α₁₀|10⟩ + α₁₁|11⟩

Where each α is a complex amplitude, and they must satisfy the normalization condition:

|α₀₀|² + |α₀₁|² + |α₁₀|² + |α₁₁|² = 1

This means the total probability of measuring any of the 4 possible outcomes (00, 01, 10, 11) is 100%.

## **Entanglement:**

Entanglement is a quantum phenomenon where two or more qubits become correlated in such a way that the state of one qubit cannot be described independently of the state of the others, even when separated by large distances.

**Key Properties**:

* Entangled qubits share information instantaneously.
* The overall quantum state cannot be written as a tensor product of individual qubit states.
* Measurement on one qubit affects the state of the other.

**Example**: Bell State  
One of the simplest entangled states is:

**|ψ⟩ = ()(|00⟩ + |11⟩)**

This means:

* Both qubits are in a superposition of |00⟩ and |11⟩.
* Measuring one qubit instantly determines the outcome of the other (if one is 0, the other is 0; if one is 1, the other is 1).
* Apply Hadamard gate to the first qubit to create superposition.
* Apply CNOT with control on qubit 0 and target on qubit 1 to entangle.

**Applications**:

* Quantum teleportation
* Superdense coding
* Quantum cryptography
* Quantum error correction

## **Matrix Representation of Qubits**

**Ket Notation (Dirac Notation):**

* **|0⟩** is the basis state representing 0
* **|1⟩** is the basis state representing 1

**Vector (Matrix) Form:**

|0⟩ = [1 0]T

|1⟩ = [0 1]T

## **Orthogonality:**

Two vectors are **orthogonal** if their **dot product is zero**.

Let's compute the inner product (⟨0|1⟩):

⟨0| = [1 0]

|1⟩ = [0 1]T

**⟨0|1⟩ = (1 \* 0) + (0 \* 1) = 0**

⟨0| = [1 0]

|0⟩ = [1

0]

**⟨0|0⟩ = (1×1) + (0×0) = 1**

⟨1| = [0 1]

|1⟩ = [0

1]

**⟨1|1⟩ = (0×0) + (1×1) = 1**

Hence, |0⟩ and |1⟩ are **orthogonal**.

## **Normalization:**

Each state must be **normalized** — the sum of squares of amplitudes = 1:

‖|0⟩‖² = |1|² + |0|² = 1

‖|1⟩‖² = |0|² + |1|² = 1

## **Inner Product: ⟨row|φ⟩**

* **Definition:**  
  The inner product (also known as the **bra-ket**) measures the **similarity or overlap** between two quantum states: ⟨row| and |φ⟩.
* **Mathematical Form:**  
  ⟨row|φ⟩ = ∑(conjugate(rowᵢ) × φᵢ) for all i  
  This is a **complex scalar** (a single number).
* **Geometric Interpretation:**  
  It’s like the dot product in linear algebra, giving the projection of one state onto another.
* **Physical Meaning:**
  + If ⟨row|φ⟩ = 0 → states are orthogonal (completely different).
  + If ⟨row|φ⟩ = 1 → states are identical (perfect overlap).
  + In quantum mechanics, the **probability amplitude** of transitioning from state |φ⟩ to state |row⟩ is ⟨row|φ⟩, and the **probability** is |⟨row|φ⟩|².
* **Example:** Let |row⟩ = [, ], |φ⟩ = [1, 0]

Then ⟨row|φ⟩ = ()\*1 + ()\*0 =

## **Outer Product: |row⟩⟨φ|**

* **Definition:**  
  The outer product creates a **matrix (or linear operator)** from two vectors: one in column form (|row⟩) and one in row form (⟨φ|).
* **Mathematical Form:**  
  |row⟩⟨φ| = matrix where each entry is (rowᵢ × conjugate(φⱼ))
* **Physical Meaning:**
  + Represents a **projection operator** onto the state |row⟩.
  + If applied to a quantum state |ψ⟩:  
    (|row⟩⟨φ|)|ψ⟩ = |row⟩ × ⟨φ|ψ⟩ → projects |ψ⟩ onto |row⟩ weighted by overlap with |φ⟩.
* **Use Cases:**
  + Building density matrices: ρ = |ψ⟩⟨ψ|
  + Projecting quantum states
  + Describing measurements and observables
* **Example:** Let |row⟩ = [1, 0]ᵗ and ⟨φ| = [0, 1]  
  Then |row⟩⟨φ| =

**[1 0]T [0 1] = [0 1]**

**[0 0]**

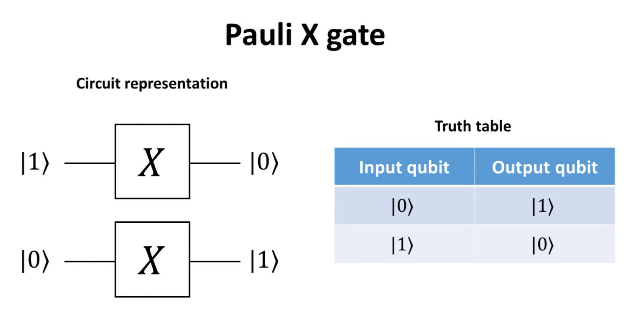
## **Quantum Gates & Circuits:**

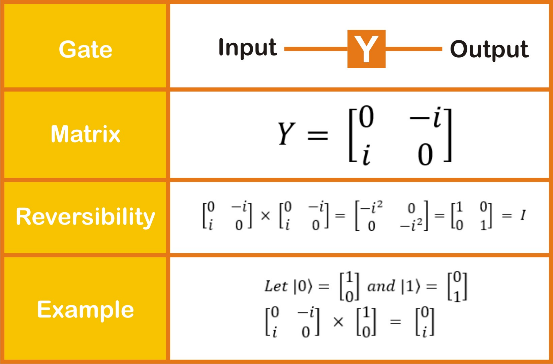
**Quantum Gates** are unitary operations that manipulate qubit states. They are the building blocks of quantum circuits, similar to logic gates in classical circuits.

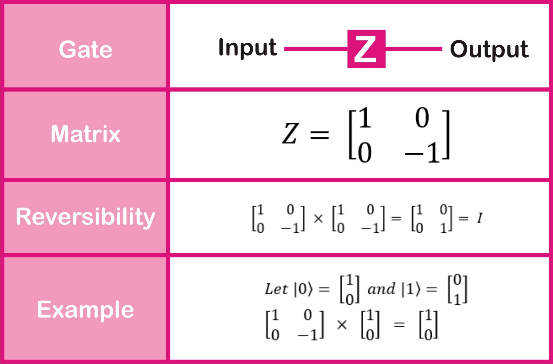
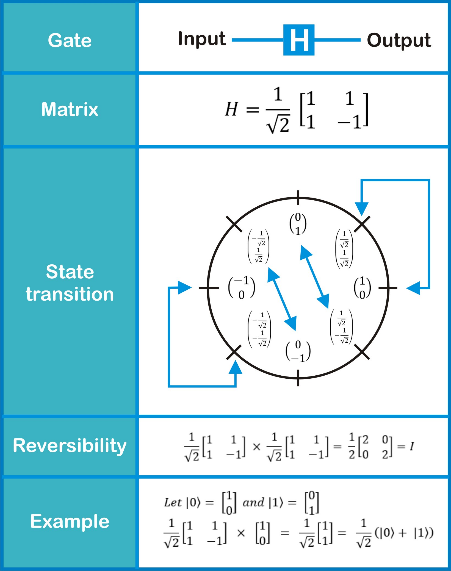
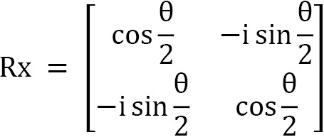
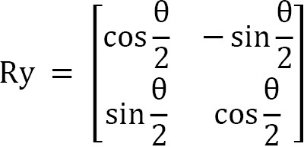
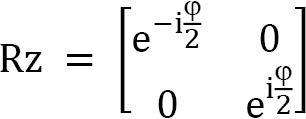
**Single-Qubit Gates**

| **Gate** | **Symbol** | **Matrix Representation** | **Action** |
| --- | --- | --- | --- |
| **Pauli-X** | X | [0 1]  [​1 0​] | Bit-flip (like NOT gate) |
| **Pauli-Y** | Y | [0 i]  [​−i 0​] | Bit + phase flip |
| **Pauli-Z** | Z | [1 0]  [0 −1] | Phase-flip |
| **Hadamard** | H | [1 1]  [1−1] | Creates superposition |
| **RX(θ)** | RX | Rotation around X-axis | cos() |
| **RY(θ)** | RY | Rotation around Y-axis | cos() |
| **RZ(θ)** | RZ | Rotation around Z-axis | Phase rotation |

**Examples:**

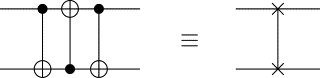
* 1. **Pauli-X Gate (X Gate)**
* **Action**: Flips the state |0⟩ to |1⟩ and vice versa.
* **Example**:
  + Input: |0⟩
  + After X gate: |1⟩
  + Think of it like a quantum NOT gate.



* 1. **Pauli-Y Gate (Y Gate)**
* **Action**: Bit and phase flip simultaneously.
* **Example**:
  + Input: |0⟩
  + After Y gate: i∣1⟩
  1. **Pauli-Z Gate (Z Gate)**
* **Action**: Flips the phase of |1⟩.
* **Example**:
  + Input: ∣ψ⟩= ()(|0⟩ + |1⟩)
  + After Z gate: |ψ⟩ = ()(|0⟩ − |1⟩)
  + The amplitude of |1⟩ gets a minus sign.
  1. **Hadamard Gate (H Gate)**
* **Action**: Creates a superposition of |0⟩ and |1⟩.
* **Example**:
  + Input: |0⟩
  + After H gate: |ψ⟩ = ()(|0⟩ + |1⟩)
  + The qubit now has a 50% chance of being 0 or 1 upon measurement.
  1. **Rotation Gates (RX, RY, RZ)**
* **RX(θ)**: Rotates qubit state about X-axis on the Bloch Sphere.
  + Example: Start with |0⟩ and apply RX(π) → ends up in |1⟩.
* **RY(θ)**: Rotates qubit around Y-axis.
  + Example: RY() on |0⟩ →  
    |ψ⟩ = cos)|0⟩ + sin ()|1⟩ = ()(|0⟩ + |1⟩)
* **RZ(θ)**: Adds a relative phase between |0⟩ and |1⟩.
  + Example: RZ(π) on |ψ⟩ = ()(|0⟩ + |1⟩) → ()(|0⟩ − |1⟩)
  1. **Swap gate:**

**Action**: The SWAP gate **exchanges the quantum states** of two qubits.

Suppose you have two qubits:

* Qubit 1:            |a⟩ = α|0⟩ + β|1⟩
* Qubit 2:            |b⟩ = γ|0⟩ + δ|1⟩

After applying the **SWAP gate**:

* Qubit 1 becomes: |b⟩ = γ|0⟩ + δ|1⟩
* Qubit 2 becomes: |a⟩ = α|0⟩ + β|1⟩

This gate **does not entangle** the qubits—it just swaps their individual quantum states.

# **Encoding Techniques**

## **Basis Encoding:**

Basis encoding is a simple method to encode **binary classical data** (i.e., 0s and 1s) into quantum states.

Each classical bit is **mapped directly** to the computational basis state of a qubit:

* Classical 0 → quantum state |0⟩
* Classical 1 → quantum state |1⟩

So, a classical binary string like **1010** becomes a quantum state:

**|ψ⟩ = |1⟩ ⊗ |0⟩ ⊗ |1⟩ ⊗ |0⟩**

**Quantum Gate Used**

* **Pauli-X Gate**
  + Flips |0⟩ ↔ |1⟩
  + Only applied when the classical bit is 1

**Code:**

def basis\_encoding\_circuit(data, max\_bits=4):

    scaled\_data = np.round((data - np.min(data)) / (np.max(data) - np.min(data)) \* (2\*\*max\_bits - 1))

    binary\_data = np.concatenate([

        np.array(list(map(int, format(int(x), f'0{max\_bits}b')))) for x in scaled\_data

    ])

    num\_qubits = len(binary\_data)

    dev = qml.device("default.qubit", wires=num\_qubits)

    @qml.qnode(dev)

    def circuit():

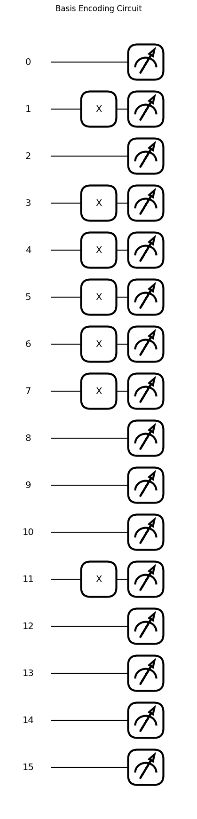
        for i, bit in enumerate(binary\_data):

            if bit == 1:

                qml.PauliX(wires=i)

        return qml.probs(wires=range(num\_qubits))  # Add measurement for visualization

    return circuit



* + 1. **Quantum Environment Setup**

A quantum environment is initialized using the default.qubit simulator in PennyLane, where the number of qubits is determined by the length of the binary-encoded data.

* + 1. **Data Normalization and Scaling**

The classical input data is normalized to the [0, 1] range and scaled to fit within the range [0,2max\_bits−1], ensuring compatibility with binary encoding.

* + 1. **Binary Representation**

Each scaled value is converted into a fixed-length binary string (using max\_bits) to capture the necessary bit-level information for encoding.

* + 1. **Binary Concatenation**

All binary strings from the dataset are concatenated into a single 1D binary array representing the entire classical input.

* + 1. **Quantum Encoding with Pauli-X**

For each bit in the binary array, a Pauli-X gate is applied to the corresponding qubit **only if the bit is 1**, flipping it from the default |0⟩ state to |1⟩.

## **Angle Encoding:**

Angle encoding (also called *parametric encoding* or *rotation encoding*) is a method of representing classical data using the rotation angles of quantum gates applied to qubits. Each classical data value is mapped to the angle parameter of a rotation gate such as **RX(θ)**, **RY(θ)**, or **RZ(θ)**.

**How It Works**

* Each data point is used as the **rotation angle** for a single-qubit rotation gate.
* Common gates used:
  + RX(θ) – rotates the qubit around the X-axis
  + RY(θ) – rotates around the Y-axis
  + RZ(θ) – rotates around the Z-axis
* For example, applying RY(π/2) to a qubit initialized in |0⟩ results in a superposition state.

**Advantages**

* **Efficient for real-valued data**: Can represent continuous data directly.
* **Simple circuit design**: Minimal gate overhead and easy implementation.
* **Scalable**: Suitable for high-dimensional input by using one rotation per qubit.

**Gates Used in Angle Encoding**

1. **RX(θ) — Rotation around the X-axis**
   * Gate: qml.RX(θ, wires=i)
   * Action: Rotates the qubit state by angle θ around the X-axis.
   * Effect: Useful for creating superposition and phase differences.
   * Example: RX(π) applied on |0⟩ → |1⟩
2. **RY(θ) — Rotation around the Y-axis**
   * Gate: qml.RY(θ, wires=i)
   * Action: Rotates the qubit state by angle θ around the Y-axis.
   * Effect: Commonly used for encoding continuous features.
   * Example: RY() on ∣0⟩ → cos()∣0⟩ + sin()∣1⟩ = ()(∣0⟩ + ∣1⟩)
3. **RZ(θ) — Rotation around the Z-axis**
   * Gate: qml.RZ(θ, wires=i)
   * Action: Adds a phase shift by rotating the qubit around the Z-axis.
   * Effect: Changes the relative phase without altering the probability amplitudes.
   * Example: RZ(π) on ∣ψ⟩ = (1/√2)(∣0⟩ + ∣1⟩) → (1/√2)(∣0⟩ − ∣1⟩)

**Code:**

def angle\_encoding\_circuit(data):

    num\_qubits = len(data)

    dev = qml.device("default.qubit", wires=num\_qubits)

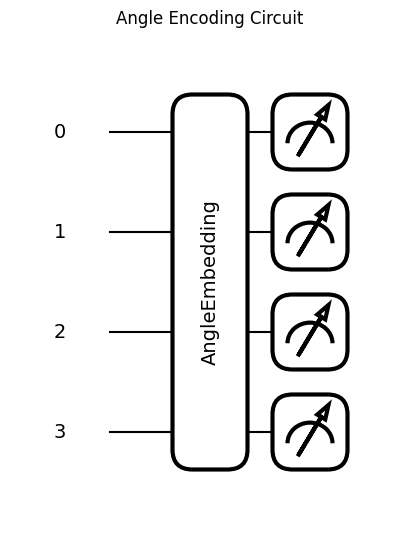
    @qml.qnode(dev)

    def circuit():

        qml.AngleEmbedding(data \* np.pi, wires=list(range(num\_qubits)), rotation="Y")

        return qml.probs(wires=range(num\_qubits))  # Add measurement for visualization

    return circuit

1. **Quantum Device Setup:** A quantum environment is initialized using the default.qubit simulator with the number of qubits equal to the length of the input data.
2. **Data Preparation:** The input classical data is multiplied by π to scale values appropriately for rotation angles.
3. **Encoding with Rotation Gates:**

**Technique:** qml.AngleEmbedding is used to encode classical numerical data into quantum states.

**Gate Used:** The Rotation around the Y-axis (**RY**) gate is applied to each qubit.

**How It Works:**

* Each data point is scaled (typically by π) to determine the rotation angle.
* The RY gate rotates the qubit's state around the Y-axis, moving it from the ∣0⟩ state into a specific superposition.

**Effect:** This method efficiently encodes continuous features into the angles of qubits on the Bloch sphere, maintaining amplitude-based relationships.

**Example:**  
Applying **RY(π/2)** on ∣0⟩ results in:  
→ ()(∣0⟩ + ∣1⟩)

## **Amplitude Encoding:**

Amplitude encoding encodes classical data directly into the amplitudes of a quantum state. A normalized classical vector is used such that its elements become the amplitudes of the quantum basis states.

**How It Works**

* A classical data vector of size 2n (where n is the number of qubits) is first normalized so that the sum of the squares of its elements equals 1.
* The vector is **normalized**, i.e.,

**i​2=1**

* This vector is then mapped onto a quantum state:

Classical data: [x0​,x1​,...,x2n−1​] →Quantum state: i=i​∣i⟩​

**Problem:**

Encode the classical data vector  
x = [0.1, −0.6, 1.0]  
into a quantum state using amplitude encoding on a 2-qubit system (4 basis states: |00⟩, |01⟩, |10⟩, |11⟩).

**Step 1: Pad the vector**

We need 2² = 4 entries for 2 qubits, but the given vector has only 3.  
So, we pad it with 0 to get:  
x\_padded = [0.1, −0.6, 1.0, 0.0]

**Step 2: Normalize the vector**  
Calculate the L2 norm:  
‖x‖ = √(0.1² + (−0.6)² + 1.0² + 0.0²)  
‖x‖ = √(0.01 + 0.36 + 1.00 + 0.0) = √1.37 ≈ 1.170

Normalize each element:  
x\_normalized = [0.1/1.170, −0.6/1.170, 1.0/1.170, 0.0/1.170]  
x\_normalized ≈ [0.0855, −0.5128, 0.8547, 0.0]

**Step 3: Write the quantum state**  
Using amplitude encoding, the normalized vector becomes the coefficients of the basis states:

|ψ⟩ = 0.0855|00⟩ − 0.5128|01⟩ + 0.8547|10⟩ + 0.0000|11⟩

**Final Answer:**  
|ψ⟩ ≈ 0.0855|00⟩ − 0.5128|01⟩ + 0.8547|10⟩  
(The |11⟩ component is 0 and hence not present.)

**Requirements**

* The data must be of length 2n; if not, zero-padding may be required.
* The input vector must be normalized before encoding to ensure it forms a valid quantum state.

**Effect**

Amplitude encoding enables the representation of high-dimensional classical data using fewer qubits by embedding all data components into the quantum state's amplitudes simultaneously.

* **Example**:  
  A 2-qubit system can represent 4 amplitudes. Given a normalized vector [a,b,c,d][a, b, c, d][a,b,c,d], the resulting quantum state becomes:

**∣ψ⟩=a∣00⟩+b∣01⟩+c∣10⟩+d∣11⟩**

**Code:**

def amplitude\_encoding(image\_path):

    """Real amplitude encoding using qml.AmplitudeEmbedding()."""

    # Load image and preprocess

    img = Image.open(image\_path).convert("L")  # Grayscale

    img = img.resize((2, 2))  # Resize to small 2x2 image → 4 pixels

    img = np.array(img).flatten().astype(float)  # Flatten to 1D

    # Pad to reach 2^num\_qubits if needed

    target\_size = 2\*\*num\_qubits

    if len(img) < target\_size:

        img = np.pad(img, (0, target\_size - len(img)), mode="constant")

    elif len(img) > target\_size:

        img = img[:target\_size]  # Truncate if too large

    # Normalize vector to have unit L2 norm

    img = img / np.linalg.norm(img)

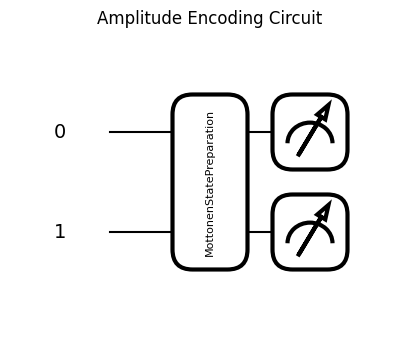
    @qml.qnode(dev)

    def circuit():

        qml.AmplitudeEmbedding(features=img, wires=range(num\_qubits), normalize=False)

        return qml.state()

    return circuit()

**Image Processing**:  
The input image is converted to grayscale, resized to 2×2, and flattened into a 1D pixel vector.

**Padding, Truncation, Normalization**:  
The vector is padded or truncated to match 2n2^n2n length (for nnn qubits) and normalized to unit L2 norm.

**Quantum Circuit:**

* **Embedding**:
  + Uses qml.AmplitudeEmbedding to encode the image vector as the **amplitudes** of a quantum state.
  + normalize=False is used because the vector is already manually normalized.
* **Measurement**:
  + qml.state() returns the full quantum state vector to inspect the amplitudes directly.

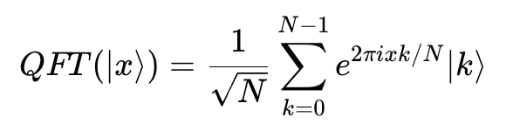
**Gate Used:**

* **AmplitudeEmbedding (Decomposed Internally)**:
  + Uses a sequence of **multi-controlled rotations and entangling gates** to embed amplitudes.
  + No direct rotation gates like RX/RY/RZ are visible, but they are **generated internally** by the Pennylane compiler when decomposing the embedding.

**Effect:**

Encodes the image information into quantum amplitudes, allowing further quantum processing or classification using this compact quantum representation.

## **Quantum Fourier Transform**

The **Quantum Fourier Transform** is the quantum analogue of the classical **Discrete Fourier Transform (DFT)**. It transforms a quantum state from the computational basis into the frequency domain.

It re-encodes the amplitude information in terms of periodicity and phase.

**What Does It Do?**

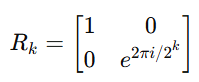
* It reveals **hidden periodicities** in quantum states — this is key in quantum algorithms like Shor’s algorithm for factoring.
* Changes the representation of quantum information from **time/spatial domain to frequency domain**.
* QFT works exponentially faster than classical FFT:
  + **Classical FFT**: O(N log N)
  + **Quantum QFT**: O((log N)²)

**Why is QFT Useful?**

1. **Shor’s Algorithm** – finds the period of a function, which helps in factoring integers exponentially faster than classical algorithms.
2. **Phase Estimation** – used to estimate eigenvalues of unitary operators.
3. **Signal Processing & Simulations** – useful in simulating periodic phenomena in quantum systems.
4. **Quantum Phase Kickback** – fundamental in quantum phase estimation and interference.

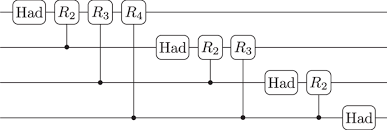
**Gates Used in QFT**

For an **n-qubit** system, QFT uses:

1. **Hadamard Gates (H):**
   * Introduce equal superposition and start the interference pattern.
   * Applied to each qubit.
2. **Controlled Phase Rotation Gates (R\_k):**
   * Apply a phase shift based on the state of another qubit.
   * 
   * These add the quantum version of the frequency phase shift.
3. **SWAP Gates (optional):**
   * At the end, to reverse the qubit order because QFT outputs the result in bit-reversed order.

**Structure of QFT Circuit (3-qubit example)**

For 3 qubits |q₀ q₁ q₂⟩:

* Apply H on q₀
* Apply R₂ (controlled by q₁), R₃ (controlled by q₂) on q₀
* Apply H on q₁
* Apply R₂ (controlled by q₂) on q₁
* Apply H on q₂
* Apply SWAP between q₀ and q₂

**Code:**

def qft(wires):

    """Applies Quantum Fourier Transform (QFT) on the given qubits."""

    n = len(wires)

    for i in range(n):

        qml.Hadamard(wires=i)

        for j in range(i+1, n):

            qml.ControlledPhaseShift(np.pi / 2\*\*(j - i), wires=[j, i])

    # Reverse qubit order (QFT convention)

    for i in range(n // 2):

        qml.SWAP(wires=[i, n - i - 1])

**Gates Used and Their Roles:**

1. **Hadamard Gate (H):**
   * Applied to each qubit starting from the most significant bit (leftmost).
   * It creates superposition and introduces the basic interference pattern needed for the Fourier basis.
   * Think of it as laying the foundation for encoding phase information.
2. **Controlled Phase Shift Gates (R\_k):**
   * These are applied between the current qubit and each following qubit.
   * The angle of the phase shift depends on their relative positions: θ=
   * These gates **encode relative phase shifts** — the core of Fourier transformation — by entangling qubits and accumulating frequency-domain information.
3. **SWAP Gates:**
   * At the end, qubit positions are reversed.
   * This reorders the output because QFT naturally gives results in **bit-reversed order**.
   * These swaps are purely classical (no quantum data is changed), just needed for proper interpretation.

**How These Gates Work Together in QFT:**

* The **Hadamard gate** starts the Fourier transform by mapping basis states into superpositions.
* The **controlled rotations** successively add **phase information** to qubits depending on the states of other qubits, encoding the Fourier coefficients.
* The **swap operations** rearrange the qubits to correct the output ordering of the QFT.

