Introduction to Quantum Computing – Quantum Applications

Marcelo Ponce

July 28, 2022

CMS-UTSC/SciNet

Today's lecture

The goal for today's lecture is to discuss how some applications/algorithms for quantum computers work and can be implemented.
We will discuss the following topics:

- Quatum Fourier Transform (QFT)
  Fourier Transform, DFT, FFT, ...
- Shor's Algorithm

Today's lecture

The goal for today's lecture is to discuss how some applications/algorithms for quantum computers work and can be implemented.
We will discuss the following topics:

- Quatum Fourier Transform (QFT)
  Fourier Transform, DFT, FFT, ...
- Shor's Algorithm

Material based on Xanadu's codebook and PennyLane documentation.

Please stop me if you have a question.

Recap Quantum Gates

Recap Quantum Gates

Gate	Matrix	Circuit element(s)	Basis state action
X	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	$\boxed{X}$ $\oplus$	$X\|0\rangle = \|1\rangle$ $X\|1\rangle = \|0\rangle$
H	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$	$\boxed{H}$	$H\|0\rangle = \frac{1}{\sqrt{2}}(\|0\rangle + \|1\rangle)$ $H\|1\rangle = \frac{1}{\sqrt{2}}(\|0\rangle - \|1\rangle)$
Z	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$\boxed{Z}$	$Z\|0\rangle = \|0\rangle$ $Z\|1\rangle = -\|1\rangle$
S	$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$	$\boxed{S}$	$S\|0\rangle = \|0\rangle$ $S\|1\rangle = i\|1\rangle$
T	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$	$\boxed{T}$	$T\|0\rangle = \|0\rangle$ $T\|1\rangle = e^{i\pi/4}\|1\rangle$
Y	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	$\boxed{Y}$	$Y\|0\rangle = i\|1\rangle$ $Y\|1\rangle = -i\|0\rangle$
RZ	$\begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$	$\boxed{R_z(\theta)}$	$RZ(\theta)\|0\rangle = e^{-i\frac{\theta}{2}}\|0\rangle$ $RZ(\theta)\|1\rangle = e^{i\frac{\theta}{2}}\|1\rangle$
RX	$\begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$	$\boxed{R_x(\theta)}$	$RX(\theta)\|0\rangle = \cos\frac{\theta}{2}\|0\rangle - i\sin\frac{\theta}{2}\|1\rangle$ $RX(\theta)\|1\rangle = -i\sin\frac{\theta}{2}\|0\rangle + \cos\frac{\theta}{2}\|1\rangle$
RY	$\begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$	$\boxed{R_y(\theta)}$	$RY(\theta)\|0\rangle = \cos\frac{\theta}{2}\|0\rangle + \sin\frac{\theta}{2}\|1\rangle$ $RY(\theta)\|1\rangle = -\sin\frac{\theta}{2}\|0\rangle + \cos\frac{\theta}{2}\|1\rangle$

Gate	Matrix	Circuit element(s)	Basis state action
$CNOT$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$	$\oplus$ $\boxed{X}$	$CNOT\|00\rangle = \|00\rangle$ $CNOT\|01\rangle = \|01\rangle$ $CNOT\|10\rangle = \|11\rangle$ $CNOT\|11\rangle = \|10\rangle$
CZ	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$	$\boxed{Z}$	$CZ\|00\rangle = \|00\rangle$ $CZ\|01\rangle = \|01\rangle$ $CZ\|10\rangle = \|10\rangle$ $CZ\|11\rangle = -\|11\rangle$
CRZ	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^{-i\frac{\theta}{2}} & 0 \\ 0 & 0 & 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$	$\boxed{R_z(\theta)}$	$CZ\|00\rangle = \|00\rangle$ $CZ\|01\rangle = \|01\rangle$ $CZ\|10\rangle = e^{-i\frac{\theta}{2}}\|10\rangle$ $CZ\|11\rangle = e^{i\frac{\theta}{2}}\|11\rangle$
CRX	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ 0 & 0 & -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$	$\boxed{R_x(\theta)}$	$CRX(\theta)\|00\rangle = \|00\rangle$ $CRX(\theta)\|01\rangle = \|01\rangle$ $CRX(\theta)\|10\rangle = \cos\frac{\theta}{2}\|10\rangle - i\sin\frac{\theta}{2}\|11\rangle$ $CRX(\theta)\|11\rangle = -i\sin\frac{\theta}{2}\|10\rangle + \cos\frac{\theta}{2}\|11\rangle$
CRY	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ 0 & 0 & \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$	$\boxed{R_y(\theta)}$	$CRY(\theta)\|00\rangle = \|00\rangle$ $CRY(\theta)\|01\rangle = \|01\rangle$ $CRY(\theta)\|10\rangle = \cos\frac{\theta}{2}\|10\rangle + \sin\frac{\theta}{2}\|11\rangle$ $CRY(\theta)\|11\rangle = -\sin\frac{\theta}{2}\|10\rangle + \cos\frac{\theta}{2}\|11\rangle$
CU	$\begin{pmatrix} I_2 & 0 \\ 0 & U \end{pmatrix}$	$\boxed{U}$	$CU\|00\rangle = \|00\rangle$ $CU\|01\rangle = \|01\rangle$ $CU\|10\rangle = \|1\rangle \otimes U\|0\rangle$ $CU\|11\rangle = \|1\rangle \otimes U\|1\rangle$
$SWAP$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$		$SWAP\|00\rangle = \|00\rangle$ $SWAP\|01\rangle = \|10\rangle$ $SWAP\|10\rangle = \|01\rangle$ $SWAP\|11\rangle = \|11\rangle$
TOF	$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$	$\oplus$	$TOF\|000\rangle = \|000\rangle$ $TOF\|001\rangle = \|001\rangle$ $\vdots = \vdots$ $TOF\|101\rangle = \|101\rangle$ $TOF\|110\rangle = \|111\rangle$ $TOF\|111\rangle = \|110\rangle$

Special cases of RZ: Z, S, T

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \Rightarrow \boxed{RZ(\omega)|\psi\rangle = \alpha|0\rangle + \beta e^{i\omega}|1\rangle}$$

$$\boxed{RZ(\omega) = \begin{bmatrix} e^{-i\frac{\omega}{2}} & 0 \\ 0 & e^{i\frac{\omega}{2}} \end{bmatrix} \sim \begin{bmatrix} 1 & 0 \\ 0 & e^{i\omega} \end{bmatrix}} \qquad RZ^{\dagger}(\omega) = RZ(-\omega)$$

$Z = RZ(\omega = \pi) =$	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$ZZ = Z^2 = I, Z^{\dagger} = Z$
$S = RZ(\omega = \frac{\pi}{2}) =$	$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$	$SS = Z, SSSS = ZZ = I, S^{\dagger} = SSS$
$T = RZ(\omega = \frac{\pi}{4}) =$	$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}$	$T^{\dagger} = T^7$

One more thing...

When implementing your quantum circuits, you will need to READ the DOCUMENTATION!

`https://pennylane.readthedocs.io/en/stable/code/qml.html`

Fourier Transform

Fourier Transform (FT) – brief review

- Let f be a function of some variable x
- The FT is defined as,

$$\mathcal{F}(f(x)) \equiv \hat{f}(k) \propto \int f(x) e^{\pm i k \cdot x} dx$$

- Inverse transformation,

$$\mathcal{F}^{-1}(\hat{f}(k)) \equiv f(x) \propto \int \hat{f}(k) e^{\mp i k \cdot x} dk$$

- The overall idea is that any function (under certain conditions) can be expressed as a harmonic series
- The FT is a mathematical expression of that
- Constitutes a linear (basis) transformation in *function space*.
- Transforms from spatial to wavenumber, or time to frequency, etc.
- Constants and signs are conventions.

FT – Examples & Applications

Examples

- Double sided exponential: $f(x) = e^{-a|x|} (a > 0) \Rightarrow \hat{f}(k) = \frac{2a}{a^2+k^2}$
- Rectangular pulse: $f(t) = \begin{cases} 1 & -T \leq t \leq T \\ 0 & |t| > T \end{cases} \Rightarrow \hat{f}(\omega) = 2\frac{\sin(\omega T)}{\omega}$
- Unit impulse: $f(t) = \delta(t) \Rightarrow \hat{f}(\omega) = 1$

Applications

- Solve differential equations, integration, polynomials multiplication, ...
- Communications, Signal processing, sampling.
- Harmonic analysis, principal modes, ...

Discrete Fourier Transform (DFT)

- Given a set of n function values on a regular grid: $f_j = f(j\Delta x)$
- Fourier-transform these n values (Fourier series),

$$\hat{f}_k = \sum_{j=0}^{n-1} f_j e^{\pm 2\pi ijk/n}$$

- Easy to inverse-transform (revert),

$$f_j = \frac{1}{n} \sum_{j=0}^{n-1} \hat{f}_k e^{\mp 2\pi ijk/n}$$

- Discrete Fourier transform is a linear transformation.
- In particular, it's a matrix-vector multiplication.
- Slow: naively, costs $\mathcal{O}(n^2)$

Fast Fourier Transform (FFT) i

- Derived in partial form several times before and even after Gauss, because he'd just written it in his diary in 1805 (published later).
- Rediscovered (in general form) by Cooley and Tukey in 1965.

Basic Idea

- Write each n-point FT as a sum of two $n/2$ point FTs.
- Do this recursively $\log n$ times.
- Each level requires $\sim n$ computations: $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n^2)$. Could as easily divide into 3, 5, 7, ... parts

Fast Fourier Transform (FFT) ii

- Define $\omega_n = e^{(2\pi i)/n}$.
- Note that $\omega_n^2 = \omega_{\frac{n}{2}}$.
- DFT takes form of matrix-vector multiplication: $\hat{f}_k = \sum_{j=0}^{n-1} \omega_n^{kj} f_j$
- Rewriting this, assuming n is even, $\hat{f}_k = \underbrace{\sum_{j=0}^{\frac{n}{2}-1} \omega_{\frac{n}{2}}^{kj} f_{2j}}_{\text{FT of even samples}} + \omega_n^k \underbrace{\sum_{j=0}^{\frac{n}{2}-1} \omega_{\frac{n}{2}}^{kj} f_{2j+1}}_{\text{FT of odd samples}}$

Inverse DFT

- Inverse DFT is similar to forward DFT, up to a normalization: almost just as fast.

$$f_j = \frac{1}{n} \sum_{k=0}^{n-1} \hat{f}_k e^{\mp 2\pi i j k / n}$$

Inverse DFT is similar to forward DFT, up to a normalization: almost just as fast.

- FFT allows quick back-and-forth between x and k domain (or e.g. time and frequency domain).
- Allows parts of the computation and/or analysis to be done in the most convenient or efficient domain.

Polynomial Multiplication

Let's consider two polynomials,

$$A(x) = x^2 + 2x + 1$$
$$B(x) = 3x^2 + 2$$

Polynomial Multiplication

Let's consider two polynomials,

$$A(x) = x^2 + 2x + 1$$
$$B(x) = 3x^2 + 2$$

To compute $C(x) = A(x) \cdot B(x)$, we can use the distribute property,

$$\begin{aligned} C(x) &= (x^2 + 2x + 1) \cdot (3x^2 + 2) \\ &= x^2 \cdot (3x^2 + 2) + 2x \cdot (3x^2 + 2) + 1 \cdot (3x^2 + 2) \\ &= 1 + 2x + 4x^2 + 6x^3 + 3x^4 \end{aligned}$$

Coefficient representation: $C(x) \rightsquigarrow [1, 2, 4, 6, 3]$

How difficult is multiplying two polynomials?

Given two polynomials of degree d,

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d = \textstyle\sum_{n=0}^{d} a_j x^j$$
$$B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_d x^d = \textstyle\sum_{n=0}^{d} b_j x^j$$

$\Rightarrow C(x) = A(x) \cdot B(x)$ is a polynomial with degree $2d$,

$$C(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{2d} x^{2d} = \sum_{n=0}^{2d} c_j x^j$$

Using the distribute property, we end up with a complexity $\sim \mathcal{O}(d^2)$

Faster polynomials multiplication

Polynomials can be multiplied faster when they are represented using *values* (a set of $n \geq d+1$ points representing a d-degree polynomial) instead of using coefficients

Polynomials can be converted from the *coefficient representation* to the *value representation* using the Discrete Fourier transform.

Evaluation: coeffs. $\rightarrow$ values

$$A(x) = \{(x_0, A(x_0), (x_1, A(x_1)), \ldots, (x_{2d+1}, A(x_{2d+1})\}$$
$$B(x) = \{(x_0, B(x_0), (x_1, B(x_1)), \ldots, (x_{2d+1}, B(x_{2d+1})\}$$

Multiplication in the values representation

$C(x) =$

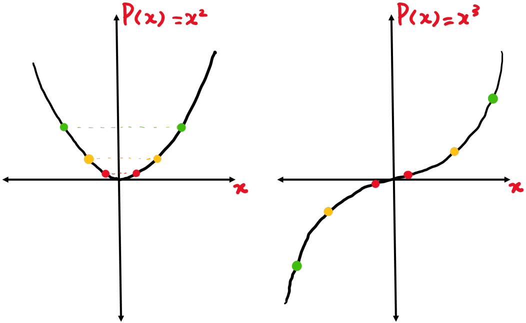
$\{(x_0, A(x_0) \cdot B(x_0), (x_1, A(x_1) \cdot B(x_1)), \ldots, (x_{2d+1}, A(x_{2d+1}) \cdot B(x_{2d+1}))\}$

$\sim \mathcal{O}(d)$!!!

Alternative Algorithm for Multiplying Polynomials

1. *Selection*: choose a set of $n \geq 2d + 1$ points to represent both our polynomials.
   $\sim$ constant linear time
2. *Evaluation*: convert the two polynomials from the coefficient representation to the value representation.
   $\sim$ linear time and that we have to evaluate the polynomial at $n \geq 2d + 1$ points
   $\Rightarrow$ **quadratic** running time for evaluating and multiplying both the polynomials
3. *Multiplication*: multiply element-wise to get the value representation of the product of the polynomials.
   $\sim \mathcal{O}(d)$
4. *Interpolation*: convert the value representation back to the coefficient representation.

The FFT algorithm is an elegant technique that uses the *divide-and-conquer* approach to make evaluation and interpolation faster.



(Credit: Xanadu)

$P(x) = x^2$ is an *even function*; only needs to evaluate the polynomial at only $n/2$ points – $P(x) = P(-x)$
$P(x) = x^3$ is an *odd function*;
$P(x) = -P(-x)$

In general, any given polynomial could be partitioned into an even and odd part,
$P(x) = P_{even}(x) + P_{odd}(x)$

Eg.

$$\boldsymbol{P(x) = 1 + 2x + 4x^2 + 6x^3 + 3x^4}$$

Eg.

$$\boldsymbol{P(x) = 1 + 2x + 4x^2 + 6x^3 + 3x^4 \Rightarrow P(x) = \underbrace{(1 + 4x^2 + 3x^4)}_{P_e(x)} + x\underbrace{(2 + 6x^2)}_{P_o(x)}}$$

Eg.

$$P(x) = 1 + 2x + 4x^2 + 6x^3 + 3x^4 \Rightarrow P(x) = \underbrace{(1 + 4x^2 + 3x^4)}_{P_e(x)} + x\underbrace{(2 + 6x^2)}_{P_o(x)}$$

$u \equiv x^2$,

Eg.

$$P(x) = 1 + 2x + 4x^2 + 6x^3 + 3x^4 \Rightarrow P(x) = \underbrace{(1 + 4x^2 + 3x^4)}_{P_e(x)} + x\underbrace{(2 + 6x^2)}_{P_o(x)}$$

$$u \equiv x^2, \quad \begin{array}{l} P_e(u) = 1 + 4u + 3u^2 \\ P_o(u) = 2 + 6u \end{array} \quad \Rightarrow P(x) = P_e(x^2) + xP_o(x^2)$$

We can think of polynomials of x^2 (rather than x) with a degree $d \leq 2$, we can evaluate them at fewer points.

We are not reducing the number of points, rather just the number of evaluations based on the relationship between positive and negative pairs of points.

For a general polynomial,

$$\begin{aligned} P(x) &= P_e(x^2) + xP_o(x^2) \\ P(-x) &= P_e(x^2) - xP_o(x^2) \end{aligned}$$

To evaluate a polynomial of degree $(n-1)$, we need to evaluate it at n points.

Recursively, we can evaluate $P_e(x^2)$ and $P_o(x^2)$ at each $[x_1^2, x_2^2, \dots, x_{n/2}^2] \rightarrow$ using the parity relations between the polynomials, we end up with two polynomials evaluated at $\frac{n}{2}$ points.

However, the $\{x_i^2\}$ points posse a problem... if $x_i \in \mathbb{R} \Rightarrow \{x_i^2\} > 0$

n-th Roots of Unity

The way to fix the issue of being left out of negative numbers can be solved by considering the n-th roots of the eqn. $\boxed{x^n = 1}$.

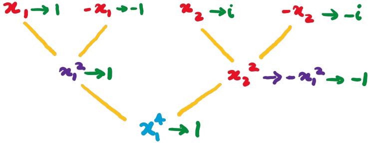
The way to fix the issue of being left out of negative numbers can be solved by considering the n-th roots of the eqn. $\boxed{x^n = 1}$.

Eg. for $n = 4$,

n-th Roots of Unity

The way to fix the issue of being left out of negative numbers can be solved by considering the n-th roots of the eqn. $\boxed{x^n = 1}$.

Eg. for $n = 4$,
we have $[1, -1, i, -i]$.



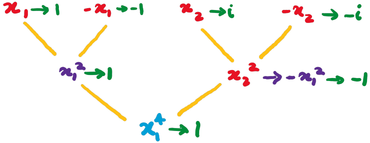
(Credit: Xanadu)

n-th Roots of Unity

The way to fix the issue of being left out of negative numbers can be solved by considering the n-th roots of the eqn. $\boxed{x^n = 1}$.

Eg. for $n = 4$,

we have $[1, -1, i, -i]$.

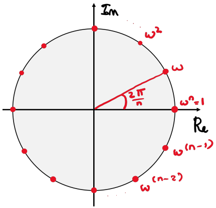


(Credit: Xanadu)

In general, for $x^n = 1$, there are n complex roots: $[\omega^0, \omega^1, \ldots, \omega^{(n-1)}]$ where

$$\omega = e^{\frac{2\pi i}{n}} = \cos\left(\frac{2\pi}{n}\right) + i\sin\left(\frac{2\pi}{n}\right)$$

For any polynomial of degree d, we choose $n \geq (d+1)$ roots of unity so that the polynomial can be evaluated at these points – **with n a power of 2**.



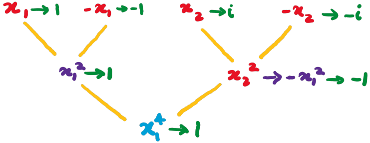
(Credit: Xanadu)

n-th Roots of Unity

The way to fix the issue of being left out of negative numbers can be solved by considering the n-th roots of the eqn. $\boxed{x^n = 1}$.

Eg. for $n = 4$,

we have $[1, -1, i, -i]$.

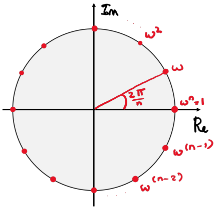


(Credit: Xanadu)

In general, for $x^n = 1$, there are n complex roots: $[\omega^0, \omega^1, \ldots, \omega^{(n-1)}]$ where

$$\omega = e^{\frac{2\pi i}{n}} = \cos\left(\frac{2\pi}{n}\right) + i\sin\left(\frac{2\pi}{n}\right)$$

For any polynomial of degree d, we choose $n \geq (d+1)$ roots of unity so that the polynomial can be evaluated at these points – **with n a power of 2**.



(Credit: Xanadu)

Some properties:

$$\omega^0 + \omega^1 + \ldots + \omega^{(n-1)} = 0 \qquad \sum_{k=0}^{n-1} \omega^{xk} = 0, \text{ for } x \neq 0$$

Interpolation

Given any polynomial $P(x) = (x_0, P(x_0)), (x_1, P(x_1)), \ldots, (x_d, P(x_d))$

$$P(x_0) = p_0 + p_1 x_0 + p_2 x_0^2 + \cdots + p_d x_0^d$$
$$P(x_1) = p_0 + p_1 x_1 + p_2 x_1^2 + \cdots + p_d x_1^d$$
$$\vdots$$
$$P(x_d) = p_0 + p_1 x_d + p_2 x_d^2 + \cdots + p_d x_d^d$$

Interpolation

Given any polynomial $P(x) = (x_0, P(x_0)), (x_1, P(x_1)), \ldots, (x_d, P(x_d))$

$$\begin{aligned} P(x_0) &= p_0 + p_1 x_0 + p_2 x_0^2 + \cdots + p_d x_0^d \\ P(x_1) &= p_0 + p_1 x_1 + p_2 x_1^2 + \cdots + p_d x_1^d \\ &\vdots \\ P(x_d) &= p_0 + p_1 x_d + p_2 x_d^2 + \cdots + p_d x_d^d \end{aligned} \qquad \begin{bmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_d) \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^d \\ 1 & x_1 & x_1^2 & \cdots & x_1^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_d & x_d^2 & \cdots & x_d^d \end{bmatrix}}_{\text{Vandermonde matrix}} \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_d \end{bmatrix}$$

Interpolation

Given any polynomial $P(x) = (x_0, P(x_0)), (x_1, P(x_1)), \ldots, (x_d, P(x_d))$

$$
\begin{array}{l}
P(x_0) = p_0 + p_1 x_0 + p_2 x_0^2 + \cdots + p_d x_0^d \\
P(x_1) = p_0 + p_1 x_1 + p_2 x_1^2 + \cdots + p_d x_1^d \\
\vdots \\
P(x_d) = p_0 + p_1 x_d + p_2 x_d^2 + \cdots + p_d x_d^d
\end{array}
\qquad
\begin{bmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_d) \end{bmatrix}
=
\underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^d \\ 1 & x_1 & x_1^2 & \cdots & x_1^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_d & x_d^2 & \cdots & x_d^d \end{bmatrix}}_{\text{Vandermonde matrix}}
\begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_d \end{bmatrix}
$$

Evaluating in the set of $n \geq (d+1)$ points as the n-th roots of unity, Given any polynomial $P(x) = (x_0, P(x_0)), (x_1, P(x_1)), \dots, (x_d, P(x_d))$

$$\begin{aligned} P(x_0) &= p_0 + p_1 x_0 + p_2 x_0^2 + \cdots + p_d x_0^d \\ P(x_1) &= p_0 + p_1 x_1 + p_2 x_1^2 + \cdots + p_d x_1^d \\ &\vdots \\ P(x_d) &= p_0 + p_1 x_d + p_2 x_d^2 + \cdots + p_d x_d^d \end{aligned} \qquad \begin{bmatrix} P(x_0) \\ P(x_1) \\ \vdots \\ P(x_d) \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^d \\ 1 & x_1 & x_1^2 & \cdots & x_1^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_d & x_d^2 & \cdots & x_d^d \end{bmatrix}}_{\text{Vandermonde matrix}} \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_d \end{bmatrix}$$

Evaluating in the set of $n \geq (d+1)$ points as the n-th roots of unity,

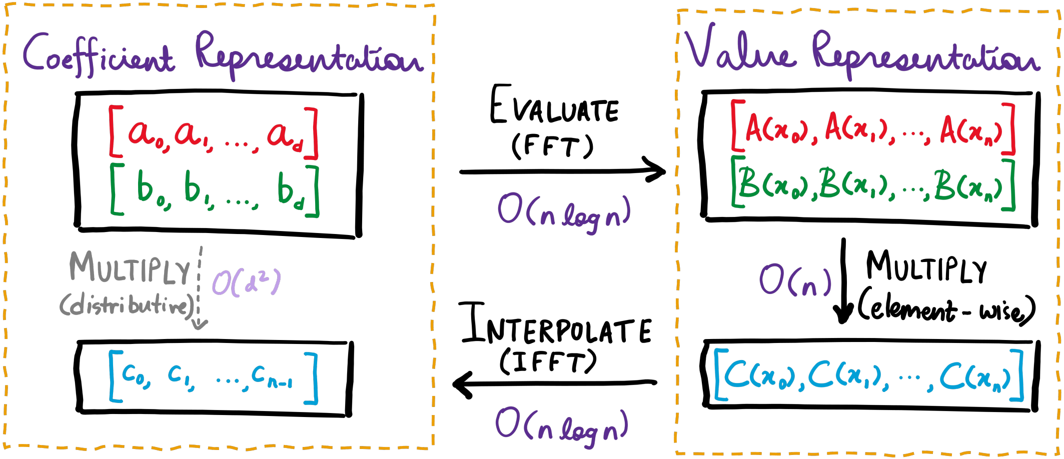
$$\begin{bmatrix} P(\omega^0) \\ P(\omega^1) \\ \vdots \\ P(\omega^{n-1}) \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}}_{\text{DFT matrix}} \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{n-1} \end{bmatrix}$$

Unitary DFT Matrix & IDFT

- Interpolation, as matrix-vector multiplication $\rightsquigarrow \mathcal{O}(n^2) \Rightarrow$ FFT $\sim \mathcal{O}(n \log n)$
- The DFT matrix, is unitary up to a factor of n, i.e. $U_{DFT} U_{DFT}^{\dagger} = nI$
- The DFT matrix is invertible, $U_{DFT}^{-1} = \frac{1}{n} U_{DFT}^{\dagger}$
- The Inverse Discrete Fourier transform (IDFT) is essentially just the DFT but with a factor of $\frac{1}{n}$ and the inverse roots of unity

$$U_{DFT} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}$$

$$U_{IDFT} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$



(Credit: Xanadu)

Numpy FFT implementation of Polynomials Multiplication i

Given a polynomial in its coefficient representation, convert it into a value representation using NumPy's DFT/FFT module.

```
def coefficients_to_values(coefficients):
    """Returns the value representation of a polynomial

    Args:
        coefficients (array[complex]): a 1-D array of complex
            coefficients of a polynomial with
            index i representing the i-th degree coefficient

    Returns:
        array[complex]: the value representation of the
            polynomial
    """
    # apply FFT
    return np.fft.fft(coefficients)


A = [4, 3, 2, 1]
print(coefficients_to_values(A))
```

Numpy FFT implementation of Polynomials Multiplication ii

Given a polynomial in its value representation, use the NumPy's DFT/FFT module to convert from the value representation to the coefficient representation.

```python
def values_to_coefficients(values):
    """Returns the coefficient representation of a polynomial

    Args:
        values (array[complex]): a 1-D complex array with
            the value representation of a polynomial

    Returns:
        array[complex]: a 1-D complex array of coefficients
    """

    # apply inverse-FFT
    return np.fft.ifft(values)


A = [10.+0.j,  2.-2.j,  2.+0.j,  2.+2.j]
print(values_to_coefficients(A))
```

Implement a helper function `nearest_power_of_2` that calculates a power of 2 that is greater than a given number.

```python
def nearest_power_of_2(x):
    """Given an integer, return the nearest power of 2.

    Args:
        x (int): a positive integer

    Returns:
        int: the nearest power of 2 of x
    """

    return int(2**np.ceil(np.log2(x)))
```

Numpy FFT implementation of Polynomials Multiplication iv

Given two polynomials in their coefficient representation, write a function to multiply them both using the functions coefficients_to_values, nearest_power_of_2, and values_to_coefficients

```
def fft_multiplication(poly_a, poly_b):
    """Returns the result of multiplying two polynomials

    Args:
        poly_a (array[complex]): 1-D array of coefficients
        poly_b (array[complex]): 1-D array of coefficients

    Returns:
        array[complex]: complex coefficients of the product
            of the polynomials
    """

    # Calculate the number of values required
    # polynomial degree
    d = (len(poly_a)-1)+(len(poly_b)-1) + 1

    # Figure out the nearest power of 2
    d=nearest_power_of_2(d)
```

Numpy FFT implementation of Polynomials Multiplication v

```python

    # Pad zeros to the polynomial
    # padding: 2nd arg a list with nbr of elements before and after
    pad_poly_a=np.pad(poly_a,(0,d-len(poly_a)),'constant',constant_values=(0))
    pad_poly_b=np.pad(poly_b,(0,d-len(poly_b)),'constant',constant_values=(0))

    # Convert the polynomials to value representation
    poly_a_values = coefficients_to_values(pad_poly_a)
    poly_b_values = coefficients_to_values(pad_poly_b)

    # Multiply
    result = np.multiply(poly_a_values,poly_b_values)

    # Convert back to coefficient representation
    return values_to_coefficients(result)
```

Quantum Fourier Transform

Quantum Fourier Transform

The Quantum Fourier transform (QFT) is the quantum version of the discrete Fourier transform (DFT).

The transformation is applied to the amplitudes of a quantum state, rotating the state vectors from any given basis (e.g., the computational basis) into the Fourier basis.

$$U_{QFT} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{(N-1)} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{(N-1)} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)^2} \end{bmatrix}$$

with $N = 2^n, \omega = e^{\frac{2\pi i}{N}}$.

One qubit, $n = 1$

$N = 2^1 = 2, \omega = e^{\pi i} = -1$

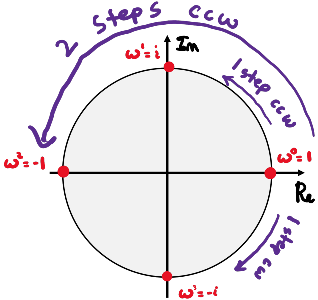
$$U_{QFT} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Two qubits, $n = 2$

$N = 2^2 = 4, \omega = e^{\pi i/2} = i$

$$U_{QFT} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

Looking at the structure of the QFT matrix, eg. consider the columns, its values are cycling the roots of unity.
The particular columns represent different speeds at which we cycle around.



(Credit: Xanadu)

QFT Circuit

Qn: which gate operations will result in operator/matrix as the QFT one?

QFT Circuit

Qn: which gate operations will result in operator/matrix as the QFT one?

$$\frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \xrightarrow{\text{SWAP}} \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \\ 1 & -i & -1 & i \end{bmatrix}$$

(Credit: Xanadu)

QFT Circuit

Qn: which gate operations will result in operator/matrix as the QFT one?

the two upper blocks $\rightsquigarrow$ Hadamard

$$\frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \xrightarrow{\text{SWAP}} \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \\ 1 & -i & -1 & i \end{bmatrix}$$

(Credit: Xanadu)

QFT Circuit

Qn: which gate operations will result in operator/matrix as the QFT one?

the two upper blocks $\rightsquigarrow$ Hadamard

bottom:

$$\frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \xrightarrow{\text{SWAP}} \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \\ 1 & -i & -1 & i \end{bmatrix}$$

(Credit: Xanadu)

Qn: which gate operations will result in operator/matrix as the QFT one?

$$\frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \xrightarrow{\text{SWAP}} \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \\ 1 & -i & -1 & i \end{bmatrix}$$

(Credit: Xanadu)

the two upper blocks $\rightsquigarrow$ Hadamard

bottom: $\boldsymbol{HS} =$

$$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} = \begin{bmatrix} 1 & i \\ 1 & -i \end{bmatrix}$$

QFT Circuit

Qn: which gate operations will result in operator/matrix as the QFT one?

$$\frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \xrightarrow{\text{SWAP}} \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \\ 1 & -i & -1 & i \end{bmatrix}$$

(Credit: Xanadu)

the two upper blocks $\rightsquigarrow$ Hadamard

bottom: $\boldsymbol{HS} =$

$$\frac{\mathbf{1}}{\sqrt{\mathbf{2}}}\begin{bmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{-1} \end{bmatrix}\begin{bmatrix} \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \boldsymbol{i} \end{bmatrix} = \begin{bmatrix} \mathbf{1} & \boldsymbol{i} \\ \mathbf{1} & \boldsymbol{-i} \end{bmatrix}$$

$$\Rightarrow \boxed{\boldsymbol{U_{QFT}} = \frac{\mathbf{1}}{\sqrt{\mathbf{2}}}\begin{bmatrix} \boldsymbol{H} & \boldsymbol{H} \\ \boldsymbol{HS} & \boldsymbol{-HS} \end{bmatrix}}$$

QFT Circuit

Qn: which gate operations will result in operator/matrix as the QFT one?

$$\frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & i & -1 & -i\\ 1 & -1 & 1 & -1\\ 1 & -i & -1 & i\end{bmatrix} \xrightarrow{\text{SWAP}} \frac{1}{2}\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & i & -1 & -i\\ 1 & -i & -1 & i\end{bmatrix}$$

(Credit: Xanadu)

the two upper blocks $\rightsquigarrow$ Hadamard

bottom: $HS = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}1 & 0\\ 0 & i\end{bmatrix} = \begin{bmatrix}1 & i\\ 1 & -i\end{bmatrix}$

$$\Rightarrow \boxed{U_{QFT} = \frac{1}{\sqrt{2}}\begin{bmatrix}H & H\\ HS & -HS\end{bmatrix}}$$

$$\Rightarrow U^S_{QFT} = (I\otimes H)(I\otimes|0\rangle\langle 0| + S\otimes|1\rangle\langle 1|)(H\otimes I)$$

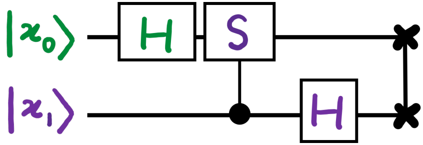
$$U_{QFT}^{S} = (I \otimes H)(I \otimes |0\rangle\langle 0| + S \otimes |1\rangle\langle 1|)(H \otimes I)$$

is a tensor factorized version of the modified QFT matrix.

$$U_{QFT}^{S} = (I \otimes H)(I \otimes |0\rangle\langle 0| + S \otimes |1\rangle\langle 1|)(H \otimes I)$$

is a tensor factorized version of the modified QFT matrix.

Since we swapped the inner rows, we need to swap them back!

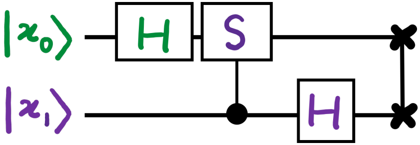


(Credit: Xanadu)

$$U^S_{QFT} = (I \otimes H)(I \otimes |0\rangle\langle 0| + S \otimes |1\rangle\langle 1|)(H \otimes I)$$

is a tensor factorized version of the modified QFT matrix.

Since we swapped the inner rows, we need to swap them back!



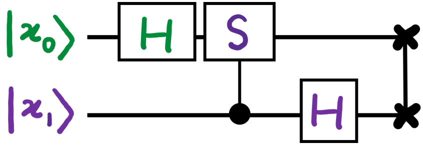
(Credit: Xanadu)

It's possible to show that the SWAP-gate applied to $U^S_{QFT} \rightsquigarrow U_{QFT}$, i.e.
$U_{QFT} = SWAP \cdot U^S_{QFT}$

$$U^S_{QFT} = (I \otimes H)(I \otimes |0\rangle\langle 0| + S \otimes |1\rangle\langle 1|)(H \otimes I)$$

is a tensor factorized version of the modified QFT matrix.

Since we swapped the inner rows, we need to swap them back!

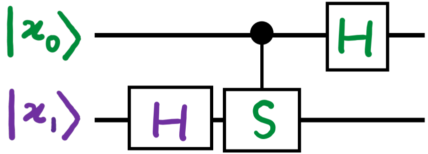


(Credit: Xanadu)

It's possible to show that the SWAP-gate applied to $U^S_{QFT} \rightsquigarrow U_{QFT}$, i.e.

$$U_{QFT} = SWAP \cdot U^S_{QFT}$$

An alternative, to reduce the circuit depth, is reversing the operations on the first and the second qubit to get rid of the SWAP-gate



(Credit: Xanadu)

Properties of the QFT

Properties of the QFT

Unitarity
QFT (more generally the discrete Fourier transformation matrix) is unitary

Unitarity
QFT (more generally the discrete Fourier transformation matrix) is unitary

Convolution-Multiplication
Given an n-qubit state: $[\alpha_0, \alpha_1, \ldots, \alpha_{(N-1)}] \xrightarrow{QFT} [\beta_0, \beta_1, \ldots, \beta_{(N-1)}]$ a new vector in the Fourier basis, with prob. $|\beta_j|^2$

If input amplitudes are shifted *cyclically*, the output distribution remains the same

Properties of the QFT

Unitarity
QFT (more generally the discrete Fourier transformation matrix) is unitary

Convolution-Multiplication
Given an n-qubit state: $[\alpha_0, \alpha_1, \ldots, \alpha_{(N-1)}] \xrightarrow{QFT} [\beta_0, \beta_1, \ldots, \beta_{(N-1)}]$ a new vector in the Fourier basis, with prob. $|\beta_j|^2$

If input amplitudes are shifted *cyclically*, the output distribution remains the same

Periodicity
For periodic functions, $|\alpha\rangle = (\alpha_0, \alpha_1, \ldots, \alpha_{(N-1)})$, whose period r divides N

$$\Rightarrow |\alpha\rangle = (\alpha_0, \alpha_1, \ldots, \alpha_{(r-1)}, \alpha_0, \alpha_1, \ldots, \alpha_{(r-1)}, \cdots)$$

$\Rightarrow |\alpha\rangle = \sqrt{\frac{r}{N}} \sum_{j=0}^{\frac{N}{r}-1} |jr\rangle \implies$ the QFT is also **periodic** with period $\frac{N}{r}$

QFT – Hands-on i

Implement the circuit that performs the single-qubit QFT operation, i.e., for $n = 1$.

```
dev = qml.device("default.qubit", wires=1)

@qml.qnode(dev)
def one_qubit_QFT(basis_id):
    """A circuit that computes the QFT on a single qubit.

    Args:
        basis_id (int): An integer value identifying
            the basis state to construct.

    Returns:
        array[complex]: The state of the qubit after applying QFT.
    """
    # Prepare the basis state |basis_id>
    bits = [int(x) for x in np.binary_repr(basis_id, width=dev.num_wires)]
    qml.BasisStatePreparation(bits, wires=[0])

    ### YOUR CODE HERE ###
```

QFT – Hands-on ii

```python
dev = qml.device("default.qubit", wires=1)

@qml.qnode(dev)
def one_qubit_QFT(basis_id):
    """A circuit that computes the QFT on a single qubit.

    Args:
        basis_id (int): An integer value identifying
            the basis state to construct.

    Returns:
        array[complex]: The state of the qubit after applying QFT.
    """
    # Prepare the basis state |basis_id>
    bits = [int(x) for x in np.binary_repr(basis_id, width=dev.num_wires)]
    qml.BasisStatePreparation(bits, wires=[0])

    # The QFT on a single qubit can be performed using the Hadamard gate.
    qml.Hadamard(wires=0)

    return qml.state()
```

QFT – Hands-on iii

Implement a circuit that performs the two-qubit QFT operation.

```
n_bits = 2
dev = qml.device("default.qubit", wires=n_bits)

@qml.qnode(dev)
def two_qubit_QFT(basis_id):
    """A circuit that computes the QFT on two qubits using qml.QubitUnitary.

    Args:
        basis_id (int): An integer value identifying the basis state to construct.

    Returns:
        array[complex]: The state of the qubits after the QFT operation.
    """

    # Prepare the basis state |basis_id>
    bits = [int(x) for x in np.binary_repr(basis_id, width=dev.num_wires)]
    qml.BasisStatePreparation(bits, wires=[0, 1])

    ### YOUR CODE HERE ###
```

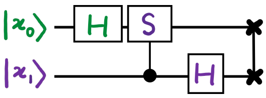
QFT – Hands-on iv

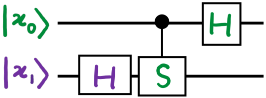
```
n_bits = 2
dev = qml.device("default.qubit", wires=n_bits)

@qml.qnode(dev)
def two_qubit_QFT(basis_id):
    """A circuit that computes the QFT on two qubits using qml.QubitUnitary.

    Args:
        basis_id (int): An integer value identifying the basis state to construct.

    Returns:
        array[complex]: The state of the qubits after the QFT operation.
    """

    # Prepare the basis state |basis_id>
    bits = [int(x) for x in np.binary_repr(basis_id, width=dev.num_wires)]
    qml.BasisStatePreparation(bits, wires=[0, 1])

    # define U_QFT matrix for n=2
    U_QFT=0.5 * np.array([[1,1,1,1], [1,1j,-1,-1j], [1,-1,1,-1], [1,-1j,-1,1j]])

    # Apply U_QFT
    qml.QubitUnitary(U_QFT,wires=[0,1])
```

QFT – Hands-on v

```
    return qml.state()
```

Implement the two-qubit QFT using a combination of gates (**without** using `qml.QubitUnitary`).

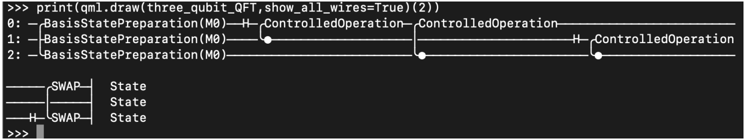


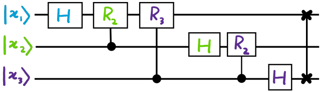



```python
dev = qml.device("default.qubit", wires=2)

@qml.qnode(dev)
def decompose_two_qubit_QFT(basis_id):
    """A circuit that computes the QFT on two qubits using elementary gates.

    Args:
        basis_id (int): An integer value identifying the basis state to construct.

    Returns:
        array[complex]: The state of the qubits after the QFT operation.
    """
    # Prepare the basis state |basis_id>
    bits = [int(x) for x in np.binary_repr(basis_id, width=dev.num_wires)]
    qml.BasisStatePreparation(bits, wires=[0, 1])

    ### YOUR CODE HERE ###
```

Visualizing your circuits...

```
# print/draw circuit
# print(qml.draw(decompose_two_qubit_QFT,show_all_wires=True)(2))
# https://pennylane.readthedocs.io/en/stable/code/api/pennylane.draw.html
```





(Credit: Xanadu)

https://pennylane.ai/blog/2021/05/how-to-visualize-quantum-circuits-in-pennylane/

Hadamard Transform/QFT

The Quantum Fourier transform (QFT) is closely related to the Hadamard transform.

The Hadamard transform takes a system of qubits from the computational basis (for example) to the Hadamard basis,

$$\boxed{H^{\otimes n} |x\rangle = \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} (-1)^{x \cdot y} |y\rangle}$$

$N = 2^n$

The Quantum Fourier transform can be represented as a unitary matrix,

$$\boxed{U_{QFT} |x\rangle = \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} \omega^{x \cdot y} |y\rangle}$$

$\omega = e^{\frac{2\pi i}{N}}, N = 2^n$

The Quantum Fourier transform (QFT) is closely related to the Hadamard transform.

The Hadamard transform takes a system of qubits from the computational basis (for example) to the Hadamard basis,

$$\boxed{H^{\otimes n}|x\rangle = \frac{1}{\sqrt{N}}\sum_{y=0}^{N-1}(-1)^{x\cdot y}|y\rangle}$$

$N = 2^n$

The Quantum Fourier transform can be represented as a unitary matrix,

$$\boxed{U_{QFT}|x\rangle = \frac{1}{\sqrt{N}}\sum_{y=0}^{N-1}\omega^{x\cdot y}|y\rangle}$$

$\omega = e^{\frac{2\pi i}{N}}, N = 2^n$

About representations

$$|x\rangle = \underbrace{|110\rangle}_{binary} = \underbrace{|6\rangle}_{decimal} \mapsto |1\rangle \otimes |1\rangle \otimes |0\rangle$$

A bitstring, $x_1, x_2, \ldots, x_n \rightsquigarrow$

$$x_1 2^{n-1} + x_2 2^{n-2} + x_3 2^{n-3} + \ldots + x_n 2^0 = \sum_{k=1}^{n} x_k 2^{n-k}$$

Designing the n-qubit QFT circuit

1. For a single qubit, the QFT is performed just using the Hadamard gate.
2. For $\boldsymbol{n > 1}$ qubits, we may need to apply a Hadamard gate to produce a superposition, along with some kind of rotations to account for the added phases $\boldsymbol{\omega^{x \cdot y}}$

Designing the n-qubit QFT circuit

1. For a single qubit, the QFT is performed just using the Hadamard gate.
2. For $n > 1$ qubits, we may need to apply a Hadamard gate to produce a superposition, along with some kind of rotations to account for the added phases $\omega^{x \cdot y}$

it is possible to write the QFT in a way that is tensor-factorized,

$$\boxed{\begin{aligned} U_{QFT} |x_1 x_2 \dots x_n\rangle \; = \; \frac{1}{\sqrt{N}} \Big[\; & \left(|0\rangle + e^{2\pi i 0.x_n} |1\rangle\right) \left(|0\rangle + e^{2\pi i 0.x_{n-1} x_n} |1\rangle\right) \cdots \\ & \cdots \left(|0\rangle + e^{2\pi i 0.x_1 x_2 \dots x_n} |1\rangle\right) \Big] \end{aligned}}$$

with *fractional binary*, $\frac{x_l}{2} + \frac{x_{l+1}}{2^2} + \ldots + \frac{x_m}{2^{m-l+1}} \equiv 0.x_l x_{l+1} \ldots x_m$

Designing the n-qubit QFT circuit

1. For a single qubit, the QFT is performed just using the Hadamard gate.
2. For $n > 1$ qubits, we may need to apply a Hadamard gate to produce a superposition, along with some kind of rotations to account for the added phases $\omega^{x \cdot y}$

it is possible to write the QFT in a way that is tensor-factorized,

$$\boxed{\begin{aligned} U_{QFT}|x_1 x_2 \dots x_n\rangle \;=\; \frac{1}{\sqrt{N}}\Big[\; & \left(|0\rangle + e^{2\pi i 0.x_n}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.x_{n-1}x_n}|1\rangle\right)\cdots \\ & \cdots\left(|0\rangle + e^{2\pi i 0.x_1 x_2 \dots x_n}|1\rangle\right)\Big] \end{aligned}}$$

with *fractional binary*, $\frac{x_l}{2} + \frac{x_{l+1}}{2^2} + \dots + \frac{x_m}{2^{m-l+1}} \equiv 0.x_l x_{l+1} \dots x_m$

One can prove, $\Rightarrow$ $\boxed{U_{QFT}|x_1 x_2 \dots x_n\rangle = \frac{1}{\sqrt{N}} \bigotimes_{k=1}^{n} \left[|0\rangle + e^{\frac{2\pi i}{2^k}x}|1\rangle\right]}$

Designing the n-qubit QFT circuit

1. For a single qubit, the QFT is performed just using the Hadamard gate.
2. For $n > 1$ qubits, we may need to apply a Hadamard gate to produce a superposition, along with some kind of rotations to account for the added phases $\omega^{x \cdot y}$

it is possible to write the QFT in a way that is tensor-factorized,

$$\boxed{\begin{aligned} U_{QFT} |x_1 x_2 \ldots x_n\rangle \; = \; \frac{1}{\sqrt{N}} \Big[\; & \left(|0\rangle + e^{2\pi i 0.x_n} |1\rangle\right) \left(|0\rangle + e^{2\pi i 0.x_{n-1}x_n} |1\rangle\right) \cdots \\ & \cdots \left(|0\rangle + e^{2\pi i 0.x_1 x_2 \ldots x_n} |1\rangle\right) \Big] \end{aligned}}$$

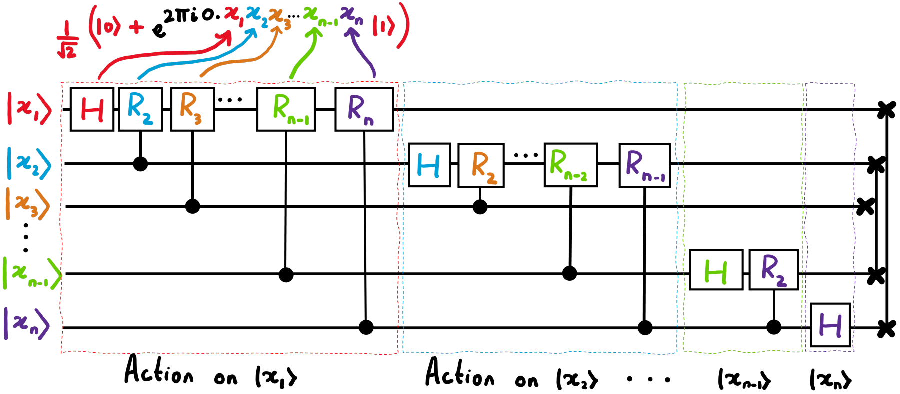
with *fractional binary*, $\frac{x_l}{2} + \frac{x_{l+1}}{2^2} + \ldots + \frac{x_m}{2^{m-l+1}} \equiv 0.x_l x_{l+1} \ldots x_m$

One can prove, $\Rightarrow$ $\boxed{U_{QFT} |x_1 x_2 \ldots x_n\rangle = \frac{1}{\sqrt{N}} \bigotimes_{k=1}^{n} \left[|0\rangle + e^{\frac{2\pi i}{2^k} x} |1\rangle \right]}$

QN: which gates will produce this state?

$$U_{QFT}|x_1x_2\dots x_n\rangle = \frac{1}{\sqrt{N}}\Big[\ \left(|0\rangle + e^{2\pi i 0.x_n}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.x_{n-1}x_n}|1\rangle\right)\cdots \cdots \left(|0\rangle + e^{2\pi i 0.x_1x_2\dots x_n}|1\rangle\right)\Big]$$

with *fractional binary*, $\frac{x_l}{2} + \frac{x_{l+1}}{2^2} + \dots + \frac{x_m}{2^{m-l+1}} \equiv 0.x_lx_{l+1}\dots x_m$



(Credit: Xanadu)

QFT Complexity

Number of gates (more generally the gate count) required for the QFT on n-qubits:

QFT Complexity

Number of gates (more generally the gate count) required for the QFT on n-qubits:
$n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2}$

QFT Complexity

Number of gates (more generally the gate count) required for the QFT on n-qubits: $n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2}$

SWAP gates, of which there are at most $n/2$ since a SWAP works on two qubits at once, $\Rightarrow \frac{n(n+1)}{2} + \frac{n}{2} \rightsquigarrow \mathcal{O}(n^2)$

QFT Complexity

Number of gates (more generally the gate count) required for the QFT on n-qubits:
$n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2}$

SWAP gates, of which there are at most $n/2$ since a SWAP works on two qubits at once, $\Rightarrow \frac{n(n+1)}{2} + \frac{n}{2} \rightsquigarrow \mathcal{O}(n^2)$

n-qubits, $N = 2^n$ amplitudes

$N = 2^n \Rightarrow n = \log(N) \Rightarrow$

$\mathcal{O}((\log(N)\log(N)))$

QFT Complexity

Number of gates (more generally the gate count) required for the QFT on n-qubits: $n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n+1)}{2}$

SWAP gates, of which there are at most $n/2$ since a SWAP works on two qubits at once, $\Rightarrow \frac{n(n+1)}{2} + \frac{n}{2} \rightsquigarrow \mathcal{O}(n^2)$

n-qubits, $N = 2^n$ amplitudes

$N = 2^n \Rightarrow n = \log(N) \Rightarrow$

$\mathcal{O}((\log(N)\log(N)))$

The transformation is encoded into the amplitudes of the qubits; $\Rightarrow$ measuring in an arbitrary basis state

QFT Complexity

Number of gates (more generally the gate count) required for the QFT on n-qubits: $n+(n-1)+(n-2)+\ldots+1=\frac{n(n+1)}{2}$

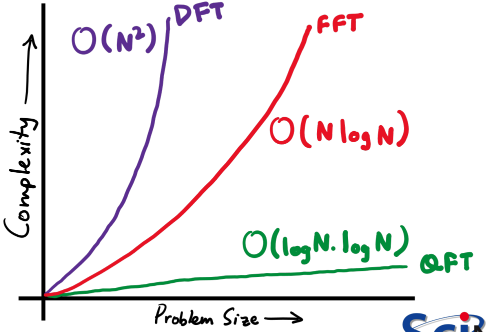
SWAP gates, of which there are at most $n/2$ since a SWAP works on two qubits at once, $\Rightarrow \frac{n(n+1)}{2}+\frac{n}{2} \rightsquigarrow \mathcal{O}(n^2)$

n-qubits, $N=2^n$ amplitudes

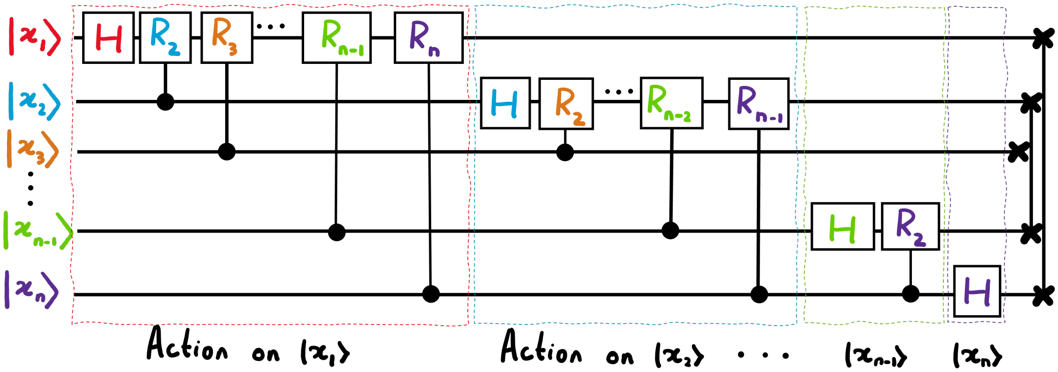
$N=2^n \Rightarrow n=\log(N) \Rightarrow$
$\mathcal{O}((\log(N)\log(N)))$

The transformation is encoded into the amplitudes of the qubits; $\Rightarrow$ measuring in an arbitrary basis state

Using the *Fourier basis*, we can solve classically intractable problems such as factoring in polynomial time.



(Credit: Xanadu)



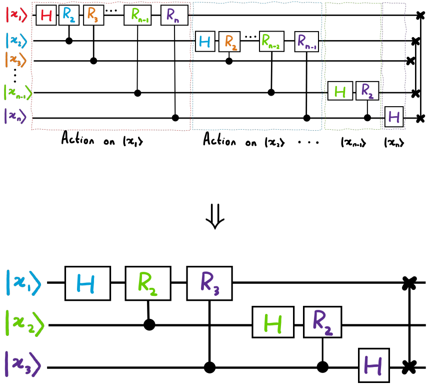
(Credit: Xanadu)

Implement the QFT for three qubits.

```
dev = qml.device("default.qubit", wires=3)

@qml.qnode(dev)
def three_qubit_QFT(basis_id):
    """A circuit that computes the QFT on three qubits.

    Args:
        basis_id (int): An integer value identifying the basis state to
            construct.

    Returns:
        array[complex]: The state of the qubits after the QFT operation.
    """
    # Prepare the basis state |basis_id>
    bits = [int(x) for x in np.binary_repr(basis_id, width=dev.num_wires)]
    qml.BasisStatePreparation(bits, wires=[0, 1, 2])
```

QFT – Hands-on iii

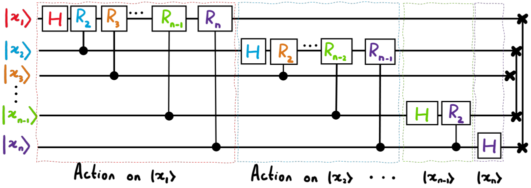
```
    # Rk gates // NOT used!
    R2=np.array([[1,0],[0,np.exp(np.pi*0.5j)]])
    R3=np.array([[1,0],[0,np.exp(np.pi*0.25j)]])
    # R2 -> TT -> S
    # R3 -> T
    # R2R3 -> TTT -> ST

    # on |0>
    qml.Hadamard(wires=0)
    qml.ctrl(qml.S,control=1)(wires=0)
    qml.ctrl(qml.T,control=2)(wires=0)

    # on |1>
    qml.Hadamard(wires=1)
    qml.ctrl(qml.S,control=2)(wires=1)

    # on |2>
    qml.Hadamard(wires=2)

    qml.SWAP(wires=[0,2])

    return qml.state()
```



(Credit: Xanadu)

QFT – Hands-on iv

Implement a circuit that reverses the order of n qubits using SWAP gates.



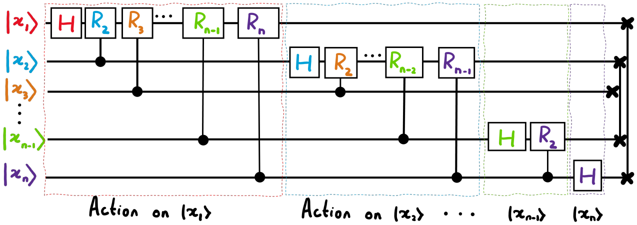
(Credit: Xanadu)

```python
def swap_bits(n_qubits):
    """A circuit that reverses the order of qubits, i.e.,
    performs a SWAP such that [q1, q2, ..., qn] -> [qn, ... q2, q1].

    Args:
        n_qubits (int): An integer value identifying the number of qubits.
    """

    # loop over pair of wires: 0,n-1; 1,n-2, ...
    for i in range(int(n_qubits/2)):
        qml.SWAP(wires=[i,(n_qubits-1)-i])
```

Exercise II: TO BE COMPLETED and submitted!

Implement the n-qubit QFT using the circuit that performs the Hadamards and controlled rotations on n qubits using `qml.ControlledPhaseShift`. Recall that you must read the documentation, e.g. see

https://pennylane.readthedocs.io/en/stable/code/api/pennylane.ControlledPhaseShift.html



(Credit: Xanadu)