Quantum Computation

Makoto Yamashita

This is lecture note for the Quantum Computation course (MAT3420) at University of Oslo, 2026 Spring semester.

Our main references are the lecture note by Aaronson [Aar18] and the textbook by Nielsen and Chuang [NC00].

This is the version with rendering of computational notebook included in the relevant sections.

1. Introduction

1.1. **What is quantum computing?** *Classical computation*, the foundation of information processing by computers, is built on *classical physics* such as electricity voltage and magnetic charges, which are then formalized by the concept of *bit-strings*, i.e., sequences of the form $01001011\cdots$. *Quantum computation*, the topic of this course, is built on *quantum mechanics*, which developed as a new paradigm of theoretical physics in the early 20th century to explain phenomena at very small scale like electrons and photons that cannot be explained by classical mechanics.

These phenomena include

- superposition: states of "particles" are given by solutions to linear equations, and in particular we can take *linear combination* of states;
- entanglement: independent measurement of two "particles" that are far apart can give correlated results.

Quantum computation is an attempt to achieve new kinds of "computing" using these features, by using quantum mechanical states to encode information instead of bit-strings. It started as more of an intellectual curiosity and thought experiments to better understand the principles of quantum mechanics, but starting from the 1990's people found out that it can potentially lead to speed-up of certain type of computations that can have real-world consequences such as cryptographyc (secure communication methods).

1.2. **Goal of this course.** There are several goals for this course.

The first is to develop familiarity with the mathematical and physical formalism underlying quantum computation. On the physical side, this includes the language of quantum mechanics (Lectures 4 and 5). In particular, we will cover:

- states modelled by vectors $v, \psi, |01\rangle, \ldots$ in *Hilbert spaces* H;
- transformation of states (quantum gates) modelled by *unitary transforms* on H;
- measurements of the system modelled by *projections* and *inner products*.

On the mathematical side, the course relies heavily on complex linear algebra (Lectures 2 and 3), and in particular ideas like:

- tensor product of vector spaces $V \otimes W$;
- linear maps acting on tensor products $S \otimes T\colon\ V \otimes W \to V' \otimes W'$.

The second goal is to study characteristic quantum phenomena and algorithms. Some of these arise as thought experiments illuminating the counter-intuitive features of quantum mechanics (Lectures 7, 8):

- quantum money scheme (Lecture 9)
- superdense coding (Lecture 11)
- the CHSH game (Lecture 12)

Others offer problem-solving algorithms, including:

Date: Spring 2026.

- quantum algorithms for Boolean functions (Lecture 13);
- Shor's prime factorization algorithm for $pq = N$ which can can break the RSA cryptography; exponential speedup over classical algorithms (Lectures 16 and 17);
- Grover's search algorithm for general search problem; quadratic speedup over classical algorithms (Lecture 21);
- the HHL algorithm for linear equations (Lecture 22).

Finally, we also want to study quantitative aspects of quantum computation, including:

- complexity of algorithms (Lecture 14)
- universality of quantum gates (Lectures 18, 19, and 20): how many basic operations do we need to achieve general computation?

1.3. **History of classical and quantum computation.** Reference: [NC00, Section 1.1]

Quantum computation grew out of a combination of several different threads that merged in the 1990s: quantum mechanics, computability theory, information theory, and cryptography.

- 1920s: invention of quantum mechanics
- 1936: Turing's paper on computability (Turing machine)
- 1940s: building first computers (von Neumann, ...)
- 1948: Shannon's entropy for communication channels; follow von Neumann's entropy for quantum mechanical states from 1920s
- 1970s and onward: technology to control quantum mechanical systems
- mid 1970s: public key cryptography by Diffie–Hellman, Rivest–Shamir–Adleman (Lecture 15)
- 1980s: thought experiments in quantum mechanics about faster-than-light signal transmission; no-cloning theorem (Lecture 9)
- 1984: the Bennett–Brassard quantum key distribution scheme (Lecture 10); secure communication through entanglement, building on Wiesner's work from 1960s
- 1985: Deutsch's thought experiment of quantum computer; simulate quantum system by a "machine"? some problems solvable by such a machine, but not by a classical / probabilistic Turing machine?
- 1994: Shor's algorithm
- 1995: Grover's algorithm
- 1998: two-qubit quantum computer
- 2020s: quantum computer with many qubits (about 150 qubits now)

2. Basics in linear algebra, classes of operators

Reference: [NC00, Section 2]

Our basic number system is *complex numbers*: $z = x + iy$ with x and y usual numbers (real numbers) that we can write with signed digit expansion like $x = -0.1234321\ldots$, and i the imaginary unit satisfying $i^2 = -1$. We write $\mathbb{C}$ for the collection of all complex numbers.

We write the *complex conjugate* as $z^* = x - iy$ for $z = x + iy$. Another common notation is $\bar{z}$.

2.1. **Hilbert spaces.** We will see that the mathematical framework for quantum mechanics is given by *Hilbert spaces* (Lecture 4). A Hilbert space is a collection H of "things" (which we call *vectors* $v, w, \cdots \in H$), with the following operations allowed:

- multiplication by complex numbers gives another vector $zv \in H$;
- sum of two vectors gives another vector $v + w \in H$;
- *inner product* of two vectors $\langle v, w\rangle$ is a complex number.

They should satisfy some consistency conditions. The most important ones are about the inner product:

$$\langle w, v\rangle = \langle v, w\rangle^*, \qquad \langle v, v\rangle \geq 0 \quad \text{(equality happens only for } v = 0),$$

and its interaction with the other two operations:

$$\langle u, v + w\rangle = \langle u, v\rangle + \langle u, w\rangle, \qquad \langle v, zw\rangle = z\langle v, w\rangle.$$

The *norm* (length) of a vector is given by

$$\|v\| = \sqrt{\langle v, v\rangle},$$

which satisfies the *triangle inequality*:

$$\|v + w\| \leq \|v\| + \|w\|.$$

Remark 2.1. To be precise, in the infinite-dimensional case we also require an extra condition called *completeness* of H for the norm $\|v\|$. This is automatic in the finite-dimensional case, and we will stick to this setting in quantum computation.

Remark 2.2. There are other notations / conventions for inner product. For example, it is common to write (v, w) instead of $\langle v, w\rangle$, and / or to ask $\langle zv, w\rangle = z\langle v, w\rangle$ instead of $\langle v, zw\rangle = z\langle v, w\rangle$. (Exercise: what is the difference?) Greek letters like $\alpha, \lambda, \ldots$ for complex numbers, and $\phi, \psi, \ldots$ for vectors are also quite common.

Example 2.3. Let us fix a positive integer n. As the Hilbert space H, we can take the collection of sequences of complex numbers $(z_1, \ldots, z_n)$, i.e., consider

$$\mathbb{C}^n = \{(z_i)_{i=1}^n \mid z_i \in \mathbb{C}\}.$$

Given $v = (y_1, \ldots, y_n)$ and $w = (z_1, \ldots, z_n)$, the three operations are given by

$$zv = (zy_1, \ldots, zy_n), \qquad v + w = (y_1 + z_1, \ldots, y_n + z_n), \qquad \langle v, w\rangle = y_1^* z_1 + \cdots + y_n^* z_n.$$

It is convenient to think of v as a *column vector* and the inner product $\langle v, w\rangle$ as a product of conjugate-transposed vector $v^\dagger$ (which becomes a *row vector*) and a column vector w, so that

$$v = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}, \qquad \langle v, w\rangle = \begin{bmatrix} y_1^*, \ldots, y_n^* \end{bmatrix} \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}.$$

We often write some important vectors with distinguished indices, like v_0, v_1 or v_+, v_-, or even with multiple symbols as indexes like v_{01}, v_{+-+}, etc. In such a situation, we often use the *Dirac notation*

$$|0\rangle, |1\rangle, |+\rangle, |-\rangle, |01\rangle, |+-+\rangle, \ldots,$$

which allows us to write the inner products like $\langle v_0, v_+\rangle, \langle v_{010}, v_{000}\rangle, \ldots$ as

$$\langle 0|+\rangle, \langle 010|000\rangle, \ldots.$$

Remark 2.4. It is often quite common to put symbols for vectors like v, ϕ inside the "kets" and "bras" like $|v\rangle$, $\langle\phi|$, but we will not do that, and only use indices of vectors inside kets and bras. So $|0\rangle$ is a vector with the *index* 0 that we could instead write $v_0 \in H$ (which typically has a nonzero length), and not the *zero vector* $0 \in H$ (which satisfies $v + 0 = v$ for other vectors v, and has the length 0).

2.2. **Linear transforms.** Vectors in Hilbert spaces are transformed by *linear transforms* (also called *linear maps* or *linear operators*). More formally, a linear transform on H is a rule to transform a vector $v \in H$ to another vector $T(v) \in H$, with consistency

$$T(zv) = zT(v), \qquad T(v + w) = T(v) + T(w).$$

It is deterministic, so that there is only one possible output value $T(v)$ for each input value v. If there is no danger of confusion, we can omit parenthesis and write Tv instead. We write the space of linear transforms on H as $\mathscr{B}(H)$.

Remark 2.5. It is possible to take different Hilbert spaces H_I and H_O for inputs and outputs of transforms, so that $v \in H_I$ would give $T(v) \in H_O$ under the transform T. We write $\mathscr{B}(H_I, H_O)$ for the space of such transforms.

Example 2.6. Square matrices $A = [A_{ij}]_{i,j=1}^n$ represent linear transforms on $H = \mathbb{C}^n$. Given an input $v = (y_1, \ldots, y_n)$, the transformed vector $z = A(v)$ is given by

$$z_i = A_{i1} y_1 + \cdots + A_{in} y_n.$$

If we think of v as a column vector, this is the matrix product operation

$$\begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} A_{11}y_1 + \cdots + A_{1n}y_n \\ \vdots \\ A_{n1}y_1 + \cdots + A_{nn}y_n \end{bmatrix}.$$

When we have a linear operator on H, say $T \in \mathscr{B}(H)$, we can consider its *adjoint* $T^\dagger \in \mathscr{B}(H)$. This is characterized by

$$\langle v, T^\dagger(w)\rangle = \langle T(v), w\rangle \quad (v, w \in H).$$

If we consider transforms between different Hilbert spaces as in Remark 2.5, the adjoint of $T \in \mathscr{B}(H_I, H_O)$ would be a transform $T^\dagger \in \mathscr{B}(H_O, H_I)$.

Remark 2.7. It is also quite common to write T^* for the adjoint.

If we have vectors $u, v \in H$, we can consider the linear operator T on H given by

$$T(w) = \langle v, w\rangle u \quad (w \in H).$$

We will write $uv^\dagger$ for this transform, so that interpretation like $v^\dagger w = \langle v, w\rangle$ becomes consistent up to exchanging the position of scalar and vector:

$$uv^\dagger w = u\langle v, w\rangle = \langle v, w\rangle u, \tag{2.1}$$

and we have $(uv^\dagger)^\dagger = vu^\dagger$, etc.

Suppose we have indexed vectors v_i and v_j in H, so that we can use the Dirac notation to represent them as $|i\rangle, |j\rangle$, etc. If T is a linear transform on H, we can write $T|i\rangle$ for the vector $T(v_i)$. For the transform $v_i v_j^\dagger$, we can write $|i\rangle\langle j|$, which allows us to write formulas like

$$(|i\rangle\langle j|)\,|k\rangle = |i\rangle\langle j|k\rangle = \langle j|k\rangle\,|i\rangle$$

corresponding to (2.1) for $u = v_i$, $v = v_j$, and $w = v_k$.

Example 2.8. With a matrix $A = [A_{ij}]_{i,j=1}^n$, the linear transform of Example 2.6 can be written as

$$\sum_{i,j=1}^{n} A_{ij}\,|i\rangle\langle j|$$

for the *standard basis* vectors $|i\rangle$ for $i = 1, \ldots, n$.

2.3. **Unitary transforms.** A *unitary* transform on H is a linear transform U satisfying

$$U(U^\dagger(v)) = v = U^\dagger(U(v)) \quad (v \in H).$$

We write $U^{-1} = U^\dagger$ to express this condition.

An equivalent formulation is

$$\|U(v)\| = \|v\| = \|U^\dagger(v)\| \quad (v \in H).$$

Yet another equivalent way is to ask

$$\langle U(v), U(w)\rangle = \langle v, w\rangle = \langle U^\dagger(v), U^\dagger(w)\rangle \quad (v, w \in H).$$

Example 2.9. Among the 2×2-matrices, rotation matrices, its complex variant, and diagonal matrices with entries z, w such that $|z| = 1 = |w|$ are unitary:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \quad \begin{bmatrix} \cos\theta & i\sin\theta \\ i\sin\theta & \cos\theta \end{bmatrix}, \quad \begin{bmatrix} z & 0 \\ 0 & w \end{bmatrix} \quad (\theta \in \mathbb{R}, z, w \in \mathbb{C}, |z| = |w| = 1),$$

as well as their products.

Remark 2.10. We can also make sense of unitary transforms between different Hilbert spaces by analogous conditions. If there is a unitary transform from H_I to H_O, we have $\dim H_I = \dim H_O$.

2.4. **Selfadjoint and positive transforms.** A *selfadjoint* transform (also called *Hermitian* transform) on H is a linear transform T satisfying $T = T^\dagger$. One formal consequence of $T = T^\dagger$ is

$$\langle v, T(v)\rangle^* = \langle v, T(v)\rangle \quad (v \in H),$$

that is, $\langle v, T(v)\rangle$ must be a real number.

A *positive* transform is a selfadjoint transform A that satisfies

$$\langle v, A(v)\rangle \geq 0 \quad (v \in H).$$

A linear transform A on H is positive if and only if $A = T^\dagger T$ holds for some $T \in \mathscr{B}(H)$.

Remark 2.11. For a matrix, being entry-wise real or positive is very different from being selfadjoint or positive as a linear transform. For example,

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

is an entry-wise positive matrix which does not define a positive linear transform on $\mathbb{C}^2$. Even with selfadjointness added,

$$B = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

does not give a positive transform (although it does give a selfadjoint transform).

3. Functions of operators, Pauli matrices, Bloch sphere, and tensor products

Reference: [NC00, Sections 2.1.3, 2.1.7, 2.1.8]

3.1. **Functions for linear transforms.** The *trace* of a linear transform $T \in \mathscr{B}(H)$ is a complex number $\mathrm{tr}(T) \in \mathbb{C}$ characterized by

$$\mathrm{tr}(S+T) = \mathrm{tr}(S) + \mathrm{tr}(T), \qquad \mathrm{tr}(zT) = z\,\mathrm{tr}(T), \qquad \mathrm{tr}(uv^\dagger) = v^\dagger u = \langle v, u\rangle \quad (S, T \in \mathscr{B}(H), z \in \mathbb{C}).$$

When $H = \mathbb{C}^n$ and T corresponds to a square matrix $A = [A_{ij}]_{i,j=1}^n$, we have

$$\mathrm{tr}(T) = A_{11} + \cdots + A_{nn}.$$

We have

$$\mathrm{tr}(ST) = \mathrm{tr}(TS) \quad (S, T \in \mathscr{B}(H)).$$

Moreover, $\mathrm{tr}(T)$ is a real number if T is selfadjoint, and a positive number if T is positive.

The *exponential* of a linear transform $T \in \mathscr{B}(H)$ is a transform

$$\exp(T) = I + T + \frac{1}{2}T^2 + \frac{1}{6}T^3 + \cdots = \sum_{n=0}^{\infty} \frac{1}{n!}T^n,$$

where I is the *identity transform* of H defined by $I(v) = v$. When T is selfadjoint,

$$U = \exp(iT)$$

is unitary.

Example 3.1. The unitary matrices in Example 2.9 can be presented as exponentials in the following way:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \exp(iS), \qquad \begin{bmatrix} \cos\theta & i\sin\theta \\ i\sin\theta & \cos\theta \end{bmatrix} = \exp(iT), \qquad \begin{bmatrix} e^{ix} & 0 \\ 0 & e^{iy} \end{bmatrix} = \exp(iD),$$

with selfadjoint matrices

$$S = \begin{bmatrix} 0 & i\theta \\ -i\theta & 0 \end{bmatrix}, \qquad T = \begin{bmatrix} 0 & \theta \\ \theta & 0 \end{bmatrix}, \qquad D = \begin{bmatrix} x & 0 \\ 0 & y \end{bmatrix} \quad (\theta, x, y \in \mathbb{R}).$$

3.2. **Pauli matrices.** These 2×2 matrices are called *the Pauli matrices*:

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

They are both unitary and selfadjoint at the same time.

Since they can be considered as linear transforms on the Hilbert space $\mathbb{C}^2$, we can consider the corresponding eigenequation

$$T(v) = zv \quad (v \in \mathbb{C}^2, z \in \mathbb{C})$$

for $T = X, Y, Z$. Recall that it's natural to think of v as a column vector with 2 components, so that we can think of the left hand side as a matrix product of T and v. The solutions for these equations are

$$v_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad v_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

for $T = Z$,

$$v_+ = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad v_- = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

for $T = X$, and

$$v_i = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ i \end{bmatrix}, \quad v_{-i} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -i \end{bmatrix}$$

for $T = Y$. (Exercise: work out the corresponding eigenvalues z.) In the Dirac notation, we write these as

$$|0\rangle, \quad |1\rangle, \quad |+\rangle, \quad |-\rangle, \quad |i\rangle, \quad |-i\rangle. \tag{3.1}$$

The vectors $|0\rangle, |1\rangle$ are often called the *computational basis* of $\mathbb{C}^2$, since they can be used to encode one bit of data from classical computation.

3.3. **Bloch sphere.** The *Bloch sphere* gives a convenient way to represent unit vectors in the Hilbert space $\mathbb{C}^2$. It looks like Figure 3.1, which is the 2-dimensional unit sphere in the 3-dimensional Euclidean space, i.e., the solution set to $x^2 + y^2 + z^2 = 1$ with real coordinates $(x, y, z) \in \mathbb{R}^3$. Observe that the vectors $|0\rangle$ and $|1\rangle$ are on the opposite sides of the sphere, and similar relation holds for the pairs $|+\rangle$ and $|-\rangle$, and $|i\rangle$ and $|-i\rangle$.

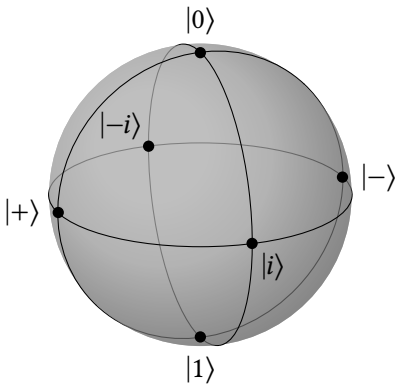


FIGURE 3.1. The Bloch sphere

Since the Hilbert space $\mathbb{C}^2$ is 4-dimensional as a real vector space, the Bloch sphere by itself is not the 3-dimensional unit ball of $\mathbb{C}^2$, which would be the solution set for $|z_1|^2 + |z_2|^2 = 1$ for the complex coordinates $(z_1, z_2) \in \mathbb{C}^2$. Instead, we identify such points $(z_1, z_2), (w_1, w_2)$ whenever we can find a complex number y such that $w_1 = yz_1$ and $w_2 = yz_2$ hold (such y should have absolute value 1). This way we reduce the dimension by 1, and get a 2-dimensional space.

Remark 3.2. Abstractly, the Bloch sphere is the *complex projective line* $\mathbb{P}^1(\mathbb{C})$.

Transformation of vectors by unitary transforms of $\mathbb{C}^2$ induce transformation of the Bloch sphere which are in fact *rotation transforms*. The unitary matrices from Example 2.9 induce rotations along the axes through $|i\rangle$ and $|-i\rangle$, $|+\rangle$ and $|-\rangle$, and $|0\rangle$ and $|1\rangle$, respectively.

Remark 3.3. A quick way to see this is through the eigenequations for the generators S, T, D from Example 3.1. For example, $|i\rangle$ is an eigenvector for S. This means that $\exp(iS)$ also has $|i\rangle$ as an eigenvector, hence does not change the corresponding point of the Bloch sphere.

3.4. **Tensor products.** *Tensor products* of vector spaces gives you a way to think about formal product of vectors, which are again vectors but in a bigger vector space.

Suppose V and W are vector spaces, i.e., certain collection of "vectors". Their tensor product space $V \otimes W$ is another collection of "vectors", that we would get from making sense of product of vectors in V and in W. That is, when v is a vector in V and w is a vector in W, we want a vector $v \otimes w \in V \otimes W$ representing their product. Once we have such vectors, we also need to allow their scalar multiples and sums as vectors in $V \otimes W$, so that formulas like

$$z(v \otimes w), \quad v_1 \otimes w_1 + v_2 \otimes w_2 \quad (z \in \mathbb{C}, v, v_i \in V, w, w_i \in W)$$

also represent vectors in $V \otimes W$.

We should then think about consistency conditions. For example, when $v = v_1 + v_2$ holds in V, we want

$$v \otimes w = v_1 \otimes w + v_2 \otimes w$$

in $V \otimes W$. Similarly, we should require

$$v \otimes (w_1 + w_2) = v \otimes w_1 + v \otimes w_2, \quad (zv) \otimes w = z(v \otimes w) = v \otimes (zw).$$

These are the only relations we impose.

When $v_1, \dots, v_m$ are basis vectors of V and $w_1, \dots, w_n$ are basis vectors of W, then

$$v_1 \otimes w_1, v_1 \otimes w_2, \dots, v_1 \otimes w_n, v_2 \otimes w_1, \dots, v_m \otimes w_n$$

become basis vectors of $V \otimes W$. In particular, we have $\dim V \otimes W = mn = \dim V \times \dim W$.

Now, suppose that H_A and H_B are Hilbert spaces. In order to make sense of the tensor product $H_A \otimes H_B$ as a Hilbert space, we have to determine the inner product between tensor product vectors. We do this by setting

$$\langle v_1 \otimes w_1, v_2 \otimes w_2 \rangle = \langle v_1, v_2 \rangle \langle w_1, w_2 \rangle,$$

and use the consistency conditions between linear combination and inner product to work out the value of inner product for general vectors, so that we have

$$\langle v_1 \otimes w_1 + v_2 \otimes w_2, v_3 \otimes w_3 \rangle = \langle v_1, v_3 \rangle \langle w_1, w_3 \rangle + \langle v_2, v_3 \rangle \langle w_2, w_3 \rangle, \quad \langle v_1 \otimes w_1, z v_2 \otimes w_2 \rangle = z \langle v_1 \otimes w_1, v_2 \otimes w_2 \rangle,$$

etc. We then have to check that this satisfies the required positivity condition $\langle u_1, u_2 \rangle \geq 0$ for $u_1, u_2 \in H_A \otimes H_B$.

If $(v_i)_i$ and $(w_j)_j$ are indexed vectors in H_A and H_B, we can use the index ij to denote $v_i \otimes w_j \in H_A \otimes H_B$. In particular, we can use the Dirac notation $|ij\rangle$ for this tensor product vector. This becomes handy when we want to take an iterated tensor product $H_A \otimes \cdots \otimes H_A$. For example, with the vectors from (3.1), we have

$$|01\rangle, |1+\rangle, |+i\rangle,$$

etc., in $\mathbb{C}^2 \otimes \mathbb{C}^2$, (we should be careful about the interpretation of $|-i\rangle$ though), and

$$|010\rangle, |1++\rangle, |+i1\rangle,$$

etc., in $\mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \mathbb{C}^2$.

If we have linear transforms T_A on H_A and T_B on H_B, we get a linear transform $T_A \otimes T_B$ on $H_A \otimes H_B$ given by

$$(T_A \otimes T_B)(v \otimes w) = T_A(v) \otimes T_B(w).$$

If T_A and T_B are both selfadjoint, $T_A \otimes T_B$ is again selfadjoint. Same relation holds for the positivity and the unitarity.

3.5. **Computational note for Lecture 3.** Exponential of transforms

[6]:
```
# print options
import sympy as sy
sy.init_printing(use_unicode=True)
```

[7]:
```
θ = sy.Symbol('θ', real=True)
```

[8]:
```
S = sy.Matrix([[0, sy.I * θ],
               [-sy.I * θ, 0]])
S, sy.exp(sy.I * S)
```

[8]:
$$\left(\begin{bmatrix} 0 & i\theta \\ -i\theta & 0 \end{bmatrix}, \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}\right)$$

[9]:
```
T = sy.Matrix([[0, θ],
               [θ, 0]])
T, sy.exp(sy.I * T)
```

[9]:
$$\left(\begin{bmatrix} 0 & \theta \\ \theta & 0 \end{bmatrix}, \begin{bmatrix} \frac{e^{i\theta}}{2} + \frac{e^{-i\theta}}{2} & \frac{e^{i\theta}}{2} - \frac{e^{-i\theta}}{2} \\ \frac{e^{i\theta}}{2} - \frac{e^{-i\theta}}{2} & \frac{e^{i\theta}}{2} + \frac{e^{-i\theta}}{2} \end{bmatrix}\right)$$

[10]:
```
sy.simplify(sy.exp(sy.I * T))
```

[10]:
$$\begin{bmatrix} \cos(\theta) & i\sin(\theta) \\ i\sin(\theta) & \cos(\theta) \end{bmatrix}$$

4. Postulates of quantum mechanics, part I: states

Reference: [NC00, Sections 2.2.1, 2.4; Aar18, Section 3.1]

The starting point of mathematical foundation of quantum mechanics is:

> States of a quantum mechanical system are represented by *unit vectors* of a Hilbert space H, i.e., vectors $v \in H$ satisfying
>
> $$\|v\| = \sqrt{\langle v, v\rangle} = 1.$$

To be precise, we consider v and zv to represent the same state if z is a complex number (satisfying $|z| = 1$).

We often cannot decide the state exactly, and instead talk about combination of different states with certain probabilities, like state v_1 with probability p_1, state v_2 with probability p_2, ..., state v_n with probability p_n. We should then have $p_1 + \cdots + p_n = 1$ for the consistency of probability.

One way to encode this situation is to use a *direct sum Hilbert space* $H^{\oplus n} = H \oplus \cdots \oplus H$, with n factors. This is a space of sequence of vectors $(w_1, \dots, w_n)$ with $w_1, \dots, w_n \in H$, and the inner product defined by

$$\langle (v_1, \dots, v_n), (w_1, \dots, w_n)\rangle = \langle v_1, w_1\rangle + \cdots + \langle v_n, w_n\rangle$$

Then, if $p_1, \dots, p_n$ are positive numbers satisfying $p_1 + \cdots p_n = 1$, and $v_1, \dots, v_n$ are unit vectors $\|v_i\| = 1$ as above,

$$v = (\sqrt{p_1}v_1, \sqrt{p_2}v_2, \dots, \sqrt{p_n}v_n)$$

is a unit vector of $H^{\oplus n}$.

Instead of changing the Hilbert space from H to $H^{\oplus n}$, we can use *density operators* on H to represent the same idea. They are linear transforms ρ on H which are:

- positive, i.e., satisfies $\langle v, \rho(v)\rangle \geq 0$ for all vectors v;
- have trace 1.

Concretely, given $p_1, \dots, p_n$ and $v_1, \dots, v_n$ as above, we can set

$$\rho = p_1 v_1 v_1^\dagger + \cdots + p_n v_n v_n^\dagger.$$

Then the positivity can be checked by

$$\begin{aligned}\langle v, \rho(v)\rangle &= p_1 v^\dagger v_1 v_1^\dagger v + \cdots + p_n v^\dagger v_n v_n^\dagger v \\ &= p_1 |\langle v_1, v\rangle|^2 + \cdots + p_n |\langle v_n, v\rangle|^2,\end{aligned}$$

(we used $\langle v, v_1\rangle = \langle v_1, v\rangle^*$ and $z^* z = |z|^2$) while the trace condition can be checked by

$$\operatorname{tr}(\rho) = p_1 v_1^\dagger v_1 + \cdots + p_n v_n^\dagger v_n = p_1 + \cdots + p_n$$

(we used $\langle v_i, v_i\rangle = 1$).

We say that unit vectors $v \in H$ represent *pure states* while density operators represent *mixed states*.

Remark 4.1. Besides being able to talk about mixed states, another advantage of density operator formalism is that a unit vector v and its scalar multiple zv with $|z| = 1$ define the same operator, by

$$\rho_{zv} = zv(zv)^\dagger = zz^* vv^\dagger = vv^\dagger = \rho_v.$$

Since the "physical states" represented by v and zv are the same, this is a more efficient model of states.

Example 4.2. (Exercise: fill in the details.) Density operators on the Hilbert space $\mathbb{C}^2$ form a 3-dimensional ball, with the "evenly mixed" state $\frac{1}{2}I_2$ (the scalar matrix with diagonal entries $\frac{1}{2}$) at the origin. To see this, we first observe that the selfadjoint operators T on $\mathbb{C}^2$, i.e., selfadjoint 2×2-matrices, span a 4-dimensional real inner product space (over real numbers) spanned by the Pauli matrices X, Y, Z, and I_2, with the inner product given by $\langle S, T\rangle = \operatorname{tr}(ST) = \operatorname{tr}(S^\dagger T)$. Imposing $\operatorname{tr}(T) = 1$ means we can write T as

$$T = \frac{1}{2}I_2 + xX + yY + zZ \quad (x, y, z \in \mathbb{R}),$$

since we have $\mathrm{tr}(X) = \mathrm{tr}(Y) = \mathrm{tr}(Z) = 0$. Now, T being a positive operator is the same thing as $T - T^2$ being a positive operator. We have

$$T^2 = \left(\frac{1}{4} + x^2 + y^2 + z^2\right) I_2 + xX + yY + zZ$$

by $XY = -YX$, etc., and this gives $T - T^2 = (\frac{1}{4} - x^2 - y^2 - z^2) I_2$. This is a positive operator if and only if $x^2 + y^2 + z^2 \leq \frac{1}{4}$, which is the defining equation for a ball of radius $\frac{1}{2}$. The boundary points correspond to pure states, and by scaling up by the factor of 2, we get the Bloch sphere. For example, the choice $z = \frac{1}{2}$ and $x = y = 0$ gives

$$T = \frac{1}{2}(I_2 + Z) = v_0 v_0^\dagger = \rho_{v_0},$$

while $z = -\frac{1}{2}$ gives

$$T = \frac{1}{2}(I_2 - Z) = v_1 v_1^\dagger = \rho_{v_1},$$

which are orthogonal projections to the (span of) eigenvectors of Z. Similar relations hold for $x = \pm\frac{1}{2}$ and $y = \pm\frac{1}{2}$.

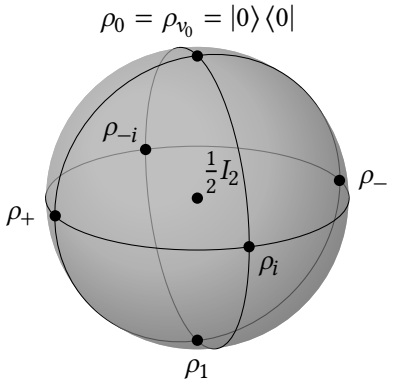


FIGURE 4.1. The refined Bloch sphere, bounding the density operators on $\mathbb{C}^2$

We will need to talk about the composite of two or more systems. Let's say we have two quantum mechanical systems A and B (think of individual information units like *bits* and *registers* in a quantum computer, or separate "laboratories" that will share an entangled quantum mechanical state), each modeled by Hilbert spaces H_A and H_B. We take the tensor product $H_A \otimes H_B$ as the Hilbert space representing the composite system AB, made of subsystems A and B. In this formalism, if we separately give the state $v \in H_A$ for the system A is the state $w \in H_B$ for the system B, then the overall state of the AB system is given by $v \otimes w \in H_A \otimes H_B$. However, $H_A \otimes H_B$ has more interesting unit vectors that can be used to model entangled states, which will come in Lecture 8 and beyond.

4.1. Computational note for Lecture 4.

[1]:
```
# print options
import sympy as sy
sy.init_printing(use_unicode=True)
```

Parametrize density operators on $\mathbb{C}^2$

[2]:
```
# Pauli matrices
X = sy.Matrix([[0, 1],
               [1, 0]])
Y = sy.Matrix([[0, -sy.I],
               [sy.I, 0]])
Z = sy.Matrix([[1, 0],
               [0, -1]])
```

[3]:
```
x, y, z = sy.symbols('x y z', real=True)
```

[4]:
```
# 'evenly mixed' state represented by scalar matrix
HlfI2 = sy.Matrix([[sy.Rational(1, 2), 0],
                   [0, sy.Rational(1, 2)]])
```

2×2 matrices which are selfadjoint and has trace 1 can be written as linear combination of I_2, X, Y, Z, with coefficient $\frac{1}{2}$ for I_2

[5]:
```
T = HlfI2 + x * X + y * Y + z * Z
T
```

[5]:
$$\begin{bmatrix} z + \frac{1}{2} & x - iy \\ x + iy & \frac{1}{2} - z \end{bmatrix}$$

given $T = T^\dagger$ and $\mathrm{tr}(T) = 1$, we expect the positivity of T to be equivalent to the positivity of $T - T^2$

[6]:
```
sy.simplify(T - T * T)
```

[6]:
$$\begin{bmatrix} -x^2 - y^2 - z^2 + \frac{1}{4} & 0 \\ 0 & -x^2 - y^2 - z^2 + \frac{1}{4} \end{bmatrix}$$

T has determinant 0 when $x^2 + y^2 + z^2 = \frac{1}{4}$, meaning it has rank 1 on the sphere defined by this equation

[7]:
```
T.det()
```

[7]:
$$-x^2 - y^2 - z^2 + \frac{1}{4}$$

5. Postulates of quantum mechanics, part II: time evolution and measurement

5.1. **Time evolution of states.** Reference: [NC00, Section 2.2.2; Aar18, Section 3.2]

The next thing to understand is how a quantum mechanical system can develop. Since states are represented by unit vectors, we should model model this by a transform that preserves these vectors. This leads to:

> When a quantum mechanical system represented by a Hilbert space H, its time evolution is given by a unitary transform on H.

In other words, if U is a unitary transform on H, then we can think of it as an "evolution rule" that says: "if the system is in a state v at a given step, at the next step it will be in the state $U(v)$." We will use this formalism to model "programs" in quantum computing.

How about mixed states? If ρ is a density operator and U is a unitary operator, both on the same Hilbert space H, then $\rho' = U\rho U^\dagger$ is again a density operator on H. This is consistent with the transform of unit vectors, since the density operator for pure states $\rho_v = vv^\dagger$ satisfy

$$\rho_{Uv} = (Uv)(Uv)^\dagger = Uvv^\dagger U^\dagger = U\rho_v U^\dagger.$$

While the above "discrete time" picture is good enough for most of quantum computing, traditional quantum mechanics is built on the idea of continuous time parameter, so we should say a word about the corresponding continuous time evolution.

In such a setting, we want to model the state of the system at time t by a unit vector-valued function $v(t)$, and write the differential equation

$$\frac{dv(t)}{dt} = Tv(t)$$

using a linear operator T (which can depend on t) on H. The consistency condition for the norm of $v(t)$ becomes skew-adjointness condition $T^\dagger = -T$, i.e., $T = iS$ for some selfadjoint operator S. If T does not depend on t, we can write down the solution for above equation as

$$v(t) = \exp(tT)(v(0)).$$

If we have a way to specify the "Hamiltonian operator" S to achieve a desired unitary operator U by $U = \exp(iS)$ (at the "unit time" $t = 1$), we can let the system evolve for the unit time to get the desired effect of U. Then we will choose another S' to get another kind of unitary U', and so on, to implement a more complicated program built out of simpler steps represented by $U, U', \ldots$.

5.2. **Measurement.** Reference: [NC00, Sections 2.2.3–2.2.6; Aar18, Section 4.1]

To formulate the concept of *measurement* in quantum mechanics, we have to use positive transforms. Suppose that we want to measure some quantity of the system represented by a Hilbert space H, whose possible values are labeled by some indices $i, j, \ldots$ over some collection I (this can be some integers like 0, 1, or symbols like +, −, or strings of such). The *positive operator-valued measure formalism* (POVM formalism) of measurement is the following postulate:

> there is a positive operator E_i on H for each index i, such that the outcome with index i happens with probability
>
> $$p(i) = \langle v, E_i(v)\rangle$$
>
> if the system is in a state represented by unit vector $v \in H$. Moreover, after the outcome m is observed, the system will be in a new state
>
> $$v' = \frac{1}{p(i)}\sqrt{E_i}(v),$$
>
> where $\sqrt{E_i}$ is the positive operator satisfying $\sqrt{E_i}^2 = E_i$.

The right hand side of the above formula is guaranteed to be nonnegative by the positivity condition on E_i. The consistency for probability is $p(i) + p(j) + \cdots = 1$ if we look at all possibilities. To achieve this for any state vector v, we should ask

$$E_1 + E_j + \cdots = I_H,$$

where the right hand side is the identity transform of H.

For mixed states, it is more sensible to carry around different possibilities of outcomes i together with probabilities $p(i)$. The corresponding operation for density operators can be written as

$$\rho' = \sum_{i \in I} \sqrt{E_i} \rho \sqrt{E_i}.$$

(Exercise: check that this will be a density operator whenever ρ is.)

Most of the time we work with a special class of POVM formalism, called projection-valued measure formalism (PVM formalism), where we assume that E_i are orthogonal projections on H, meaning we have $E_i^2 = E_i$ in addition to $E_i^\dagger = E_i$. Then the new system after getting outcome of index i will be

$$v' = \frac{1}{\|E_i(v)\|} E_i(v),$$

and the subspace $\{u \in H \mid E_i(u) = u\} \subset H$ is the Hilbert space spanned by the state vectors that give outcome i.

Remark 5.1. The PVM formalism has to do with *distinguishing* different cases of measurement outcomes. If we have a PVM $(E_i)_{i \in I}$ as above, so that we have $E_i^2 = E_i$, then the Hilbert space H is a direct sum of its subspaces $H_i = \{u \in H \mid E_i(u) = u\}$, meaning any $u \in H$ can be written as

$$u = u_i + u_j + \cdots \quad (u_i \in H_i, \ldots)$$

in a unique way. If we have a state represented by a unit vector $u_k \in H_k$ for some index k, the corresponding probability would be $p(k) = 1$ and $p(i) = 0$ for all $i \neq k$, so that we are guaranteed to get the value k by measurement. Of course, for a general state vector $u \in H$, we might have nontrivial probability distribution of $p(i)$.

5.3. **Observables.** In a more traditional quantum mechanical setting, we often talk about measurements whose outcome are numbers with decimal expansions (real numbers), like measurement of position, momentum, energy, etc. This calls for two new ingredients:

(1) we can talk about expectation value of a quantity (measurement outcome) X,

$$\mathbb{E}[X] = \sum_{\lambda} \lambda \mathbb{P}[X = \lambda],$$

where λ represents possible values of X and $\mathbb{P}[X = \lambda]$ is the probability of X taking the value λ;

(2) we might want to allow continuum possibilities (all real numbers) for possible values of λ.

If we have finitely many possibilities $\lambda_1, \ldots, \lambda_k$ for λ (so that point 2 does not matter), me have projection valued measure $E_1, \ldots, E_k$ by saying $E_i(v) = v$ means v represents a state in which X takes the value λ_i for certainty. We then have a selfadjoint operator

$$T = \sum_{m=1}^{k} \lambda_i E_i,$$

(recall that λ_i is a real number) which we call the *observable* representing X. When the system is in the state represented a unit vector $v \in H$, the expectation of X is given by

$$\mathbb{E}[X] = \langle v, T(v) \rangle.$$

To take care of point 2, we need to consider infinite-dimensional Hilbert spaces and unbounded selfadjoint operators, which is outside of the scope of this course. (It will be a part of mathematics called *functional analysis*, which developed around the same time as the foundation of quantum mechanics and von Neumann played a significant role there too.)

5.4. **More "realistic" model of measurement.** The formalism of measurement as explained in Section 5.2 is, while good enough for many parts of quantum computation, is not enough to capture the following more realistic scenario of measurement:

- we have a primary system (modelled by a Hilbert space H_S) which we want to measure, and another auxiliary system (modelled by H_A) of "measurement apparatus" (like thermometer);
- the apparatus system has k different distinguishable states, represented by orthonormal basis vectors $w_1, \dots, w_k \in H_A$;
- before measurement, we initialize the apparatus to a fixed state $w \in H_A$;
- to measure the primary system, we take the state vector $v \in H_S$ and form the tensor product $v \otimes w \in H_{SA} = H_S \otimes H_A$ in the tensor product Hilbert space (which models the composite system);
- we then apply some unitary transform U on H_{SA} to get a new state $u = U(v \otimes w)$ (this means let the apparatus interact with the primary system, like sticking the thermometer probe into the gas);
- then we do the measurement on the apparatus side.

If we look at what happens to the state vectors of the primary system, we get the operators

$$M_m(v) = (I_S \otimes w_m^\dagger)U(v \otimes w) \quad (m = 1, \dots, k)$$

where I_S is the identity transform of H_S. The probability of getting the outcome m form a given input state $v \in H_S$ is given by

$$p(m) = \langle U(v \otimes w), (I_S \otimes w_m w_m^\dagger)U(v \otimes w)\rangle = \langle M_m(v), M_m(v)\rangle,$$

since $(I_S \otimes w_m w_m^\dagger)_{m=1}^k$ would be the PVM for measurement considered on the composite system SA.

We call the operators $(M_m)_m$ *measurement operators* for such measurement scheme. We get the consistency condition

$$M_1^\dagger M_1 + \cdots + M_k^\dagger M_k = I_S$$

from $w_1 w_1^\dagger + \cdots + w_k w_k^\dagger$ from the assumption that $(w_m)_{m=1}^k$ form an orthonormal basis of H_A.

6. Quantum gates and circuits

6.1. **Quantum gates.** Reference: [NC00, Sections I.3.1–I.3.3; Aar18, Section 4.1]

Suppose we have a Hilbert space H representing a quantum mechanical system. Unitary operators on H represent time development of the system, or manipulation of states in this system. We use them to implement algorithms of quantum computation, and call them *quantum gates* (or *unitary gates*), borrowing name from *logic gates* in computer science.

Typically, the Hilbert space will be of the form $\mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2$, a tensor product of several copies of $\mathbb{C}^2$. We start with a handful of quantum gates on $\mathbb{C}^2$ (single-qubit system) and $\mathbb{C}^2 \otimes \mathbb{C}^2$ (two-qubit system), then make a more complicated gates through compositions UV (when U and V are transforms on the same space) and tensor products $U \otimes V$ (U and V can be transforms on different spaces) from known ones to create a more complicated quantum gate.

We already know a few quantum gates on the single-qubit system: rotation matrices (Exercise 3.1)

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}, \qquad \begin{bmatrix} \cos\theta & i\sin\theta \\ i\sin\theta & \cos\theta \end{bmatrix}, \qquad D(e^{ix}, e^{iy}) = \begin{bmatrix} e^{ix} & 0 \\ 0 & e^{iy} \end{bmatrix};$$

and the Paul matrices

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

We often single out some special cases of the diagonal matrix $D(e^{ix}, e^{iy})$, in particular the ones $D(1, e^{iy})$ corresponding to $x = 0$. The case of $e^{iy} = i$ (corresponding to $y = \frac{\pi}{2}$) is called the *S gate* or *phase gate*,

while the case of $e^{iy} = \frac{1}{\sqrt{2}}(1+i)$ (corresponding to $y = \frac{\pi}{4}$) is called the *T gate*. Yet another quantum gate on $\mathbb{C}^2$ is the *Hadamard gate*

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Going to the two-qubit system $\mathbb{C}^2 \otimes \mathbb{C}^2$, there is the *controlled-NOT gate* (*CNOT gate*) defined by the rule

$$|00\rangle \mapsto |00\rangle, \qquad |01\rangle \mapsto |01\rangle, \qquad |10\rangle \mapsto |11\rangle, \qquad |11\rangle \mapsto |10\rangle$$

on the *computational basis* vectors $|ij\rangle = |i\rangle \otimes |j\rangle$. This rule says "if the first bit is 0, do nothing; if it is 1, then flip the second bit". More generally, if U is a quantum gate on some H (think $\mathbb{C}^2 \otimes \cdots \mathbb{C}^2$), we can have the *controlled-U gate* on $\mathbb{C}^2 \otimes H$ defined by the rule

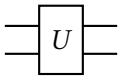
$$|0\rangle \otimes v \mapsto |0\rangle \otimes v, \qquad |1\rangle \otimes v \mapsto |1\rangle \otimes U(v)$$

for $v \in H$.

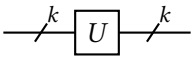
6.2. **Quantum circuit notation.** Reference: [NC00, Section I.3.4; Aar18, Section 4.2]

Complicated composite of quantum gates can be represented as diagrams, called *quantum circuit notation*. The general premise is:

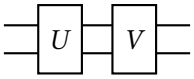
- quantum gates on $(\mathbb{C}^2)^{\otimes k} = \mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2$ (tensor product of k factors) is represented by a box with k input wires and k output wires (input to the left, output to the right), like



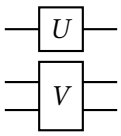
  for a quantum gate on $\mathbb{C}^2 \otimes \mathbb{C}^2$. If there are many wires, we can write like



  to say "k wires bundled together".
- composition of transforms are represented by connecting wires, like

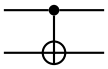


  for the composite transform VU (do U first, then V) of U an V both defined on $\mathbb{C}^2 \otimes \mathbb{C}^2$. Note that the order convention is reversed between the diagram and mathematical formula.
- tensor product of transforms are represented by vertical stackings, like

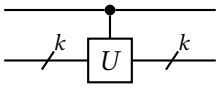


  for $U \otimes V$, with a single-qubit gate U and a two-qubit gate V.

We have a special notation for the CNOT gate:

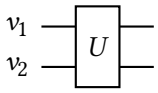


and similar one for controlled-U gate:



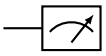
when U is a quantum gate on the k-qubit space $(\mathbb{C}^2)^{\otimes k}$.

When we take $v_1 \otimes \cdots \otimes v_k$ with $v_i \in \mathbb{C}^2$ as the initial input, we put the vectors v_i on the left end of wires (which was meant to be the input side), like



representing $U(v_1 \otimes v_2)$.

Measurement of one qubit is represented by the "meter" symbol



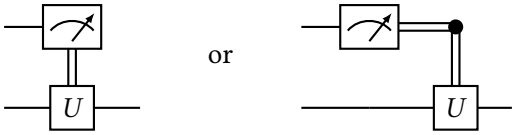
connected to the corresponding wire. That is, if we want to measure the k-th qubit ("decide if it is $|0\rangle$ or $|1\rangle$") of a state v in the N-qubit system, we use the projection-valued measure

$$E_0 = I_{(\mathbb{C}^2)^{\otimes k-1}} \otimes |0\rangle\langle 0| \otimes I_{(\mathbb{C}^2)^{\otimes N-(k+1)}}, \qquad E_1 = I_{(\mathbb{C}^2)^{\otimes k-1}} \otimes |1\rangle\langle 1| \otimes I_{(\mathbb{C}^2)^{\otimes N-(k+1)}}$$

so that probabilities of measuring the two possibilities are

$$p(0) = \langle v, E_0(v)\rangle, \qquad p(1) = \langle v, E_1(v)\rangle.$$

To use the outcome of measurement for control of another quantum gate, we can write like:



for the measurement outcome of first qubit controlling whether we want to apply the single-qubit quantum gate U on the second qubit or not. To be precise, this circuit on a state vector $v \in \mathbb{C}^2 \otimes \mathbb{C}^2$ will give the state vector

$$\frac{v'}{\|v'\|}, \quad v' = (I_2 \otimes \langle 0|)v$$

with probability

$$p(0) = \langle v, E_0(v)\rangle = v^\dagger (I_2 \otimes |0\rangle\langle 0|)v = \|v'\|^2,$$

and

$$\frac{U(v'')}{\|v''\|}, \quad v'' = (I_2 \otimes \langle 1|)v$$

with probability

$$p(1) = \langle v, E_1(v)\rangle = v^\dagger (I_2 \otimes |1\rangle\langle 1|)v = \|v''\|^2.$$

7. Quantum effect for measurement of single qubit

Measurement of single qubit, i.e., a quantum mechanical system modeled by the Hilbert space $\mathbb{C}^2$ can have some counter-intuitive consequences.

7.1. **Quantum Zeno effect.** Reference: [Aar18, Section 4.3]

Suppose we initialize the system in the state represented by $|0\rangle$, but it will start deviating due to some external effect. If the deviation is a constant drift, we can model the state at the time ϵ by the state vector

$$w_\epsilon = \cos\epsilon\,|0\rangle + \sin\epsilon\,|1\rangle\,.$$

(Of course, ϵ is a small number.) In terms of the rotation matrix (Exercise 2.9) R_θ, we can write $w_\epsilon = R_\epsilon\,|0\rangle$.

Let us perform the measurement with projection-valued measure $E_0 = |0\rangle\langle 0|$ and $E_1 = |1\rangle\langle 1|$. This means that we try to "decide" if the system is in the original state $v_0 = |0\rangle$, or is in the complement state $v_1 = |1\rangle$ distinguishable from the initial state (see Remark 5.1). The case 0 happens (we find the state $|0\rangle$) with probability

$$p(0) = w_\epsilon^\dagger E_0 w_\epsilon = \langle w_\epsilon, v_0\rangle\langle v_0, w_\epsilon\rangle = (\cos\epsilon)^2 \sim 1 - \epsilon^2,$$

and the case 1 happens (we find the state $|1\rangle$) with probability

$$p(1) = w_\epsilon^\dagger E_1 w_\epsilon = \langle w_\epsilon, v_1\rangle\langle v_1, w_\epsilon\rangle = (\sin\epsilon)^2 \sim \epsilon^2,$$

see Figure 7.1.

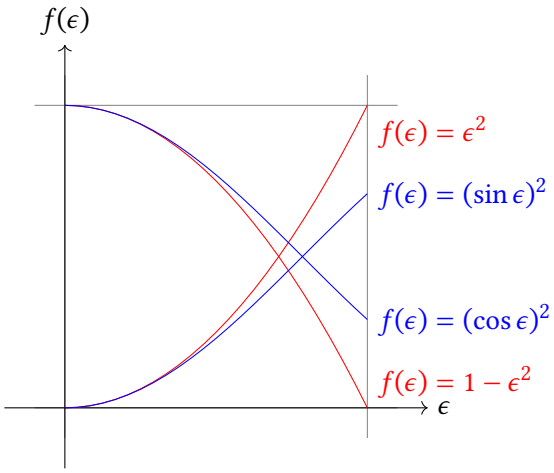


FIGURE 7.1. Probabilities of measurement under constant drift

What if we do the same after a smaller time, at $\frac{\epsilon}{N}$ for some integer N, and do this measurement N times? According to the postulate about measurement (Section 5.2), when the system is in some state v and if the measurement gives the result i ($i = 0, 1$), the system will be in a new state $|i\rangle$. Under this scheme, the probability of *not* observing the case 1 after the total time $\frac{\epsilon}{N} \times N = \epsilon$ is

$$\left(1 - \left(\frac{\epsilon}{N}\right)^2\right) \times \cdots \times \left(1 - \left(\frac{\epsilon}{N}\right)^2\right) \sim 1 - N\left(\frac{\epsilon}{N}\right)^2 = 1 - \frac{\epsilon^2}{N}.$$

In other words, it looks like the deviation became slower if we observe the quantum mechanical system more frequently. This was experimentally confirmed in 1990[1].

7.2. **Elitzur–Vaidman bomb thought experiment.** Reference: [Aar18, Section 4.4]

This is yet another thought experiment about single-qubit system proposed in 1993[2] that was soon confirmed with a beam splitter experiment[3].

Consider the following "bomb" machine:

- it accepts a qubit state $v \in \mathbb{C}^2$;
- when it receives v, it performs measurement with PVM $E_0 = |0\rangle\langle 0|$ and $E_1 = |1\rangle\langle 1|$;
- if the case 0 is observed, it outputs the state $|0\rangle$;
- if the case 1 is observed, it explodes!

We are given a suitcase that might or might not contain this machine, and we want to decide if it contains the bomb or not. To be precise, the suitcase receives a qubit state $v \in \mathbb{C}^2$, and

- it gives back v unchanged if there is no bomb inside;
- if there is bomb inside, the machine works with v as above, and if it did not explode then gives back the output state $|0\rangle$.

We fix a small number ϵ, and take an integer N approximately equal to $\frac{\pi}{2\epsilon}$. We then produce qubit states $w_0, w_1, \ldots, w_N$ in the following way:

- start with the initial state $w_0 = |0\rangle$;

[1] W. M. Itano, D. J. Heinzen, J. J. Bollinger, and D. J. Wineland, "Quantum Zeno effect", *Phys. Rev. A* **41** (5), 2295–2300 (1990).

[2] A. C. Elitzur and L. Vaidman, "Quantum mechanical interaction-free measurements", *Found. Phys.* **23**, 987–997 (1993).

[3] P. G. Kwiat, H. Weinfurter, T. Herzog, A. Zeilinger, and M. A. Kasevich, "Interaction-free measurement", *Phys. Rev. Lett.* **74** (24), 4763–4766 (1995).

- loop with the following steps, start with $i = 0$ and finish at $i = N$:
  - feed w_i to the suitcase;
  - if it exploded, end of the story;
  - if it did not explode, we get back the output vector v (it could be w_i or $|0\rangle$)
  - we then consider the rotation transform
$$R_\epsilon = \begin{bmatrix} \cos\epsilon & -\sin\epsilon \\ \sin\epsilon & \cos\epsilon \end{bmatrix},$$
    and set $w_{i+1} = R_\epsilon v$;
  - repeat the loop with the next value of i.

If the suitcase does not contain the bomb, at each step we get back the input vector w_i as v, so $w_{i+1} = R_\epsilon w_i$. After the N steps, we have

$$R_\epsilon^N w_0 = R_{N\epsilon} w_0 \simeq R_{\frac{\pi}{2}} |0\rangle = |1\rangle .$$

Then the measurement for PVM E_0 and E_1 on this state gives the case 1 almost certainly.

If the suitcase does contain the bomb, at each step we die or otherwise we try our luck next time with the state $|\epsilon\rangle = R_\epsilon |0\rangle = \cos\epsilon |0\rangle + \sin\epsilon |1\rangle$. The probability of dying with this state is

$$p(1) = \langle\epsilon|E_1|\epsilon\rangle = (\sin\epsilon)^2 \sim \epsilon^2.$$

If we do this N times, the probability of dying is about

$$N\epsilon^2 \sim \frac{\pi}{2\epsilon}\epsilon^2 = \frac{\pi}{2}\epsilon,$$

which is still small. By doing the measurement for PVM E_0 and E_1, we get the case 0.

Overall, the smaller the ϵ is, the more likely that we survive, and be able to tell with smaller error that the suitcase contains the bomb or not.

7.3. **Computational note for Lecture 7.** Elitzur-Vaidman bomb experiment

[1]:
```
import numpy as np
np.set_printoptions(suppress=True, legacy='1.25')
rng = np.random.default_rng()
```

[2]:
```
def col_vec(components):
    """Return a column vector (as a NumPy array) with the given␣
 ↪components.

    In a matrix context, this should be treated as a column vector.
    """
    return np.array(components)


def col_vec_inn_prod(v1, v2):
    """Hilbert space inner product of column vectors."""
    return np.conjugate(v1).dot(v2)


zeroket = col_vec([1, 0])
oneket = col_vec([0, 1])
plusket = col_vec([1, 1]) * 1/np.sqrt(2)
minusket = col_vec([1, -1]) * 1/np.sqrt(2)
eyeket = col_vec([1, 1j]) * 1/np.sqrt(2)
zeroket, oneket, eyeket
```

[2]:
```
(array([1, 0]),
 array([0, 1]),
 array([0.70710678+0.j        , 0.        +0.70710678j]))
```

[3]:
```
col_vec_inn_prod(zeroket, eyeket), col_vec_inn_prod(oneket, eyeket), \
    col_vec_inn_prod(plusket, eyeket)
```

[3]:
```
((0.7071067811865475+0j),
 0.7071067811865475j,
 (0.4999999999999999+0.4999999999999999j))
```

measurement of qubit state $v = (z, w)$ in "computational basis" should give the case 0 with probability $|z|^2$, and the case 1 with probability $|w|^2$

[4]:
```
def measure_qubit(v):
    """
    Measure a single-qubit state vector in the computational basis.

    Parameters
    ----------
    v : numpy.ndarray
        Length-2 complex NumPy array representing the qubit state vector.
        The vector need not be normalized.
```

```
    Returns
    -------
    int
        Measurement outcome in the computational basis.

    * 0 with probability |v0|^2, and
    * 1 with probability |v1|^2 = 1 - |v0|^2.

    Notes
    -----
    The state vector is allowed to be unnormalized. The probabilities are
    computed using the normalized state implied by `v`.
    """
    # Compute the threshold value |v0|^2 / ||v||^2
    # Do not assume ||v|| == 1
    # hence threshold = |v0|^2 / ||v||^2
    norm_v_sq = np.real(col_vec_inn_prod(v, v))
    v1_sq = np.abs(v[0])**2
    threshold = v1_sq / norm_v_sq
    rand_num = rng.random()
    if rand_num < threshold:
        return 0
    else:
        return 1
```

[5]:
```
# if we measure |0⟩, expect the outcome 0 with probability 1
[measure_qubit(zeroket) for _ in range(1000)].count(0)
```

[5]:
```
1000
```

[6]:
```
# if we measure |1⟩, expect the outcome 1 with probability 1
[measure_qubit(oneket) for _ in range(1000)].count(0)
```

[6]:
```
0
```

[7]:
```
# if we measure |+⟩, expect the outcome 0 with probability 0.5
[measure_qubit(plusket) for _ in range(1000)].count(0)
```

[7]:
```
489
```

[8]:
```
def bomb_machine(v):
    """
    Implement the Elitzur–Vaidman bomb test for a single qubit.

    Parameters
    ----------
    v : numpy.ndarray
        Length-2 complex NumPy array representing the qubit state vector
        (a single-qubit column vector). The vector need not be normalized.
```

```
    Returns
    -------
    numpy.ndarray or str
        If a projective measurement of `v` in the computational basis
        yields 0, returns the state |0⟩.
        If the measurement yields 1, returns the string "💣".
    """
    if measure_qubit(v) == 0:
        return zeroket
    else:
        return "💣"


def empty_suitcase(v):
    """An empty suitcase returns the input unchanged."""
    return v
```

[9]:
```
# with |+⟩ as input, expect the explosion half of the time, otherwise |0⟩
[bomb_machine(plusket) for _ in range(10)]
```

[9]:
```
[array([1, 0]),
 array([1, 0]),
 array([1, 0]),
 array([1, 0]),
 '💣',
 '💣',
 '💣',
 '💣',
 '💣',
 array([1, 0])]
```

[10]:
```
def EV_algo(ε, suitcase):
    """    Determine whether an Elitzur–Vaidman (EV) suitcase contains a␣
 ↪bomb.

    Parameters
    ----------
    ε : float
        Control parameter (rotation angle), assumed to be small.
    suitcase : collable
        a function implementing the EV-bomb, or `empty_suitcase` function

    Returns
    -------
    str
        A string describing the outcome:

    * `"we died"` if the bomb explodes during any suitcase interaction
```

```
    * `"guess bomb"` if the final measurement suggests a bomb is present
    * `"guess not bomb"` if the final measurement suggests no bomb

    """
    # work out the iteration count N and the rotation matrix
    N = int(np.floor(np.pi / (2 * ε)))
    R_ε = np.array([[np.cos(ε), -np.sin(ε)],
                    [np.sin(ε), np.cos(ε)]])
    # initial vector is |0⟩
    w_vec = zeroket
    for _ in range(N):
        v = suitcase(w_vec)
        if isinstance(v, str) and (v == "💣"):
            return "we died"
        # otherwise continue with the loop
        w_vec = R_ε @ v
    # if the suitcase did not explode, do measurement to decide our guess
    last_qubit_meas = measure_qubit(w_vec)
    if last_qubit_meas == 0:
        return "guess bomb"
    else:
        return "guess not bomb"
```

when the suitcase contains the EV-bomb, we expect explosion with probability $\sim \frac{\pi}{2}\epsilon$, and if we survive, get the correct guess with probability $\sim 1-\epsilon^2$

[11]:
```
ε = 0.1
np.pi * ε / 2, 1 - ε**2
```

[11]:
```
(0.15707963267948966, 0.99)
```

with $\epsilon = 0.1$, we expect to die 15% of the times, and otherwise expect the correct guess 99% of the times

[12]:
```
test_results = [EV_algo(0.1, bomb_machine) for _ in range(100)]
test_results.count("guess bomb"), test_results.count("guess not bomb"), \
    test_results.count("we died")
```

[12]:
```
(83, 1, 16)
```

[13]:
```
ε = 0.01
np.pi * ε / 2, 1 - ε**2
```

[13]:
```
(0.015707963267948967, 0.9999)
```

with $\epsilon = 0.01$, we expect to die 1.5% of the times, and otherwise expect the correct guess 99.99% of the times

[14]:
```
test_results = [EV_algo(0.01, bomb_machine) for _ in range(100)]
test_results.count("guess bomb"), test_results.count("guess not bomb"), \
    test_results.count("we died")
```

[14]:
```
(100, 0, 0)
```

when the suitcase does not contain the EV-bomb, we survive for sure, and get the correct guess with probability (at least) $\sim 1 - \epsilon^2$

[15]:
```
test_results = [EV_algo(0.1, empty_suitcase) for _ in range(100)]
test_results.count("guess bomb"), test_results.count("guess not bomb"), \
    test_results.count("we died")
```

[15]:
```
(0, 100, 0)
```

[16]:
```
test_results = [EV_algo(0.01, empty_suitcase) for _ in range(100)]
test_results.count("guess bomb"), test_results.count("guess not bomb"), \
    test_results.count("we died")
```

[16]:
```
(0, 100, 0)
```

8. Two-qubit system and entanglement

Having seen some quantum effects for single qubits, let us look at the two-qubit system next. The corresponding Hilbert space is the tensor product space $\mathbb{C}^2 \otimes \mathbb{C}^2$, which has vectors like

$$|00\rangle = |0\rangle \otimes |0\rangle, |10\rangle, |+0\rangle, |1-\rangle, \ldots$$

and their linear combinations like $|00\rangle + (1 + 0.3i)\,|{+}{+}\rangle$, etc.

8.1. **Deutsch's algorithm.** Reference: [NC00, Section 1.4.3; Aar18, Section 17.3]

With two qubits at hand, we can consider a slightly more practical problem than detecting the Elitzur–Vaidman bomb machine, call the *Deutsch problem*, proposed by Deutsch in the same paper as his proposal of quantum computation[4]. The problem is the following:

> Given a function f with input values 0, 1 and possible output values 0, 1, decide if $f(0) = f(1)$ or $f(0) \neq f(1)$.

To answer this problem in a usual computation scheme, we need to know the values of $f(0)$ and $f(1)$, and compare them; if the value of $f(i)$ is computed by some program, we need to run it twice before making the comparison. Deutsch's quantum algorithm works by running the "program" just once for the input, as soon as we can construct a quantum gate (unitary transform) modelling f and have two-qubit state as input.

Consider the following *exclusive OR* function

$$\mathrm{XOR}(0,0) = 0, \qquad \mathrm{XOR}(0,1) = 1, \qquad \mathrm{XOR}(1,0) = 1, \qquad \mathrm{XOR}(1,1) = 0.$$

We then define the quantum gate U_f on two-qubit system by

$$U_f|ij\rangle = |ik\rangle, k = \mathrm{XOR}(f(i), j) \quad (i, j = 0, 1).$$

(Exercise: write down this for the cases $f(i) = 0$ and $f'(i) = i$.) To check that U_f is a unitary transform, you can first compute $U_f^2 = I$ that follows from the relation $\mathrm{XOR}(i, \mathrm{XOR}(i, j)) = j$ valid for all choices of i and j.

Remark 8.1. The XOR operation is related to the *arithmetic modulo* 2, where we only care about integers being odd or even. In that framework XOR is the same thing as addition, and some people write $i \oplus j$ for $\mathrm{XOR}(i, j)$ because of this. For now we will avoid using $\oplus$ to avoid confusion with the direct of Hilbert spaces, but we will probably use it later for both when we get used to both concepts.

The algorithm works in this way:

- we prepare the state $v = |+-\rangle$;
- apply the quantum gate U_f to v, and set $v' = U_f(v)$;
- perform measurement for the PVM $E_+ = |+\rangle\langle+| \otimes I_2$ and $E_- = |-\rangle\langle-| \otimes I_2$;
- if the case "+" is observed, reply "$f(0) = f(1)$";
- if the case "−" is observed, reply "$f(0) \neq f(1)$".

Why does this work? First of all, the input vector v can be expanded as

$$|+-\rangle = |+\rangle \otimes |-\rangle = \left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right) \otimes \left(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right) = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle).$$

We then need to chase down what happens to the state vector in different cases.

Suppose we have $f(0) = f(1)$. We then have $v' = v$ if $f(0) = 0 = f(1)$ (because $\mathrm{XOR}(0, j) = 0$) or else $v' = -v$ if $f(0) = 1 = f(1)$ (because $\mathrm{XOR}(1, j) = \mathrm{NOT}(j)$). In any case, we get $E_+(v') = v'$ hence $p(+) = 1$.

Suppose instead we have $f(0) \neq f(1)$. If $f(0) = 0$ and $f(1) = 1$, then we have

$$v' = \frac{1}{2}(|00\rangle - |01\rangle + |11\rangle - |10\rangle)$$

[4] D. Deutsch, "Quantum theory, the Church–Turing principle and the universal quantum computer", *Proc. R. Soc. Lond. A* **400** (1818), 97–117 (1985).

but this agrees with $|{-}{-}\rangle$ by

$$|{-}{-}\rangle = \left(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right) \otimes \left(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right) = \frac{1}{2}(|00\rangle - |01\rangle - |10\rangle + |11\rangle).$$

Finally, if $f(0) = 1$ and $f(1) = 0$, then we have

$$v' = \frac{1}{2}(|01\rangle - |00\rangle + |10\rangle - |11\rangle),$$

but this is $-|{-}{-}\rangle$. In both cases we have $E_-(v') = v'$, hence $p(-) = 1$.

In the quantum circuit notation, a direct transcription of the above algorithm would be Figure 8.1, where the "+/−" label above the meter indicates we perform measurement with the PVM E_+ and E_-.

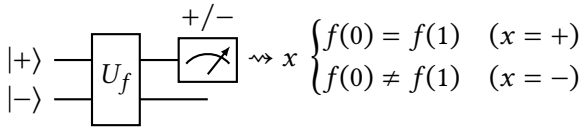


FIGURE 8.1. Deutsch's algorithm in quantum circuit notation

If the implementation only allows computational basis as input and the measurement with the PVM E_0, E_1, we can use Figure 8.2:

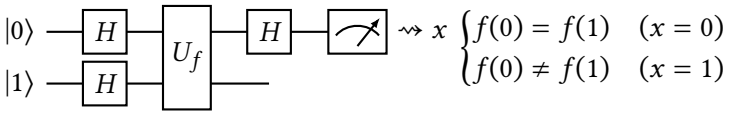


FIGURE 8.2. Equivalent quantum circuit notation for Deutsch's algorithm

8.2. **Separable states and entangled states.** Reference: [NC00, Section 2.2.8; Aar18, Section 5.3.2]

If $u, v \in \mathbb{C}^2$ are unit vectors, then their tensor product $u \otimes v \in \mathbb{C}^2 \otimes \mathbb{C}^2$ is again a unit vector:

$$\|u \otimes v\|^2 = \langle u \otimes v, u \otimes v\rangle = \langle u, u\rangle\langle v, v\rangle = 1 \cdot 1 = 1.$$

Such two-qubit states are called the *separable states* (also called the *product states*). (This makes sense for any composite system $H_{AB} = H_A \otimes H_B$.)

However, these are not all of the possible two-qubit states. For example, three real parameters are enough to specify a unit vector in $\mathbb{C}^2$, and considering $(zu) \otimes v = u \otimes zv$, separable states form a space of dimension $5 = 3 + 3 - 1$. But unit vectors in the (complex) 4-dimensional Hilbert space $\mathbb{C}^2 \otimes \mathbb{C}^2$ require seven real parameters to describe.

Two-qubit states which are not separable in the above sense, are called *entangled states*. (Again there is an analogue for more general composite systems.) One important example of entangled state is the unit vector

$$w = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \tag{8.1}$$

often called the *Einstein–Podolsky–Rosen pair* state (the EPR pair), or the *Bell pair* state because of its connection to famous thought experiments[5,6] about quantum mechanics. The relation between such entangled states and measurement for projection-valued measures of the form

$$E_i \otimes E_j = |i\rangle\langle i| \otimes |j\rangle\langle j| = (|i\rangle \otimes |j\rangle)(\langle i| \otimes \langle j|) = |ij\rangle\langle ij| \quad (i, j = 0, 1)$$

will be crucial to find interesting quantum computation schemes.

[5] A. Einstein, B. Podolsky, and N. Rosen, "Can quantum-mechanical description of physical reality be considered complete?", *Phys. Rev.* **47**, 777–780 (1935).

[6] J. S. Bell, "On the Einstein–Podolsky–Rosen paradox", *Physics* **1**, 195–200 (1964).

Remark 8.2. The EPR pair state w looks like a mixed state we talked in Lecture 4. To be more precise, we can identify tensor product space $\mathbb{C}^2 \otimes \mathbb{C}^2$ with the direct sum space $\mathbb{C}^2 \oplus \mathbb{C}^2$ by the correspondence

$$\mathbb{C}^2 \otimes \mathbb{C}^2 \ni u \otimes |0\rangle + v \otimes |1\rangle \leftrightarrow (u, v) \in \mathbb{C}^2 \oplus \mathbb{C}^2.$$

Then w represents the even mix of $|0\rangle$ and $|1\rangle$, and the corresponding density operator is $\frac{1}{2} I_2$. This means we can also write

$$w = \frac{1}{\sqrt{2}}(|{+}{+}\rangle + |{-}{-}\rangle),$$

or any other way using an orthonormal basis of $\mathbb{C}^2$.

We can produce entangled states from separable states by quantum circuits made up of components we talked in Lecture 6. For example, the above EPR pair state is given by

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \mathrm{CNOT}(H \otimes I_2)\,|00\rangle\,, \tag{8.2}$$

with the Hadamard gate

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

and the CNOT gate

$$\mathrm{CNOT}: \quad |00\rangle \mapsto |00\rangle\,, \quad |01\rangle \mapsto |01\rangle\,, \quad |10\rangle \mapsto |11\rangle\,, \quad |11\rangle \mapsto |10\rangle\,.$$

In the quantum circuit notation, we can use the diagram of Figure 8.3 to express the right hand side of (8.2).

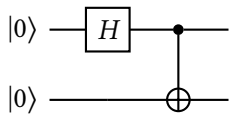


FIGURE 8.3. Quantum circuit representation of the EPR pair state

Let's check the claim (8.2). We first expand $(H \otimes I_2)\,|00\rangle$ as

$$(H \otimes I_2)\,|00\rangle = (H \otimes I_2)(|0\rangle \otimes |0\rangle) = H|0\rangle \otimes I_2\,|0\rangle\,,$$

then use $H|0\rangle = |+\rangle$ and $I_2\,|0\rangle = |0\rangle$ to get

$$(H \otimes I_2)\,|00\rangle = |+\rangle \otimes |0\rangle = \frac{1}{\sqrt{2}}(|0\rangle \otimes |0\rangle + |1\rangle \otimes |0\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle).$$

Since CNOT preserves $|00\rangle$ while changes $|10\rangle$ to $|11\rangle$, we get

$$\mathrm{CNOT}(H \otimes I_2)\,|00\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

by linearity.

8.3. **Spooky action at a distance.** Let us consider the projection-valued measure

$$E_{ij} = E_i \otimes E_j \quad (i, j = 0, 1)$$

on $\mathbb{C}^2 \otimes \mathbb{C}^2$. If we perform the measurement when the system is in the state described by the EPR pair state (8.1), the probability of observing the case ij is

$$p(ij) = w^\dagger E_{ij} w.$$

When $i = j = 0$, we get

$$p(00) = w^\dagger E_{00} w = w^\dagger\Big(\frac{1}{\sqrt{2}}\,|00\rangle\Big) = \frac{1}{\sqrt{2}^2}(\langle 00|00\rangle + \langle 11|00\rangle) = \frac{1}{2} \cdot 1 = \frac{1}{2}.$$

We get $p(11) = \frac{1}{2}$ by a similar calculation, hence the other cases have zero probability: $p(01) = p(10) = 0$. This means that the first bit and the second bit of the measurement outcome will be *correlated* if we measure the EPR pair state.

Let us think in terms of the composite system interpretation of tensor product: think of $\mathbb{C}^2 \otimes \mathbb{C}^2$ as $H_{AB} = H_A \otimes H_B$ with $H_A = H_B = \mathbb{C}^2$, i.e., the composite system with two qubit subsystems A and B. Doing measurement of the subsystem A (in the computational basis) corresponds to using the PVM

$$E_0^A = E_0 \otimes I_2, \qquad E_1^A = E_1 \otimes I_2,$$

while doing measurement of the subsystem B corresponds to using

$$E_0^B = I_2 \otimes E_0, \qquad E_1^B = I_2 \otimes E_1.$$

Now, suppose that the system was in the EPR pair state at the beginning, and the A-subsystem measurement gave the case 0. This means the state has changed to

$$v = \frac{w'}{\|w'\|}, \quad (w' = E_0^A(w))$$

according to the postulate about measurement. Computing $E_0^A(w)$, we get

$$E_0^A(w) = (|0\rangle\langle 0| \otimes I_2)\Big(\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)\Big) = \frac{1}{\sqrt{2}}\Big((|0\rangle\langle 0| \otimes I_2)(|00\rangle + |11\rangle)\Big) = \frac{1}{\sqrt{2}}|00\rangle$$

from

$$(|0\rangle\langle 0| \otimes I_2)(|00\rangle) = (|0\rangle\langle 0| \otimes I_2)(|0\rangle \otimes |0\rangle) = |0\rangle\langle 0|0\rangle \otimes I_2|0\rangle = |0\rangle \otimes |0\rangle$$

and

$$(|0\rangle\langle 0| \otimes I_2)(|11\rangle) = (|0\rangle\langle 0| \otimes I_2)(|1\rangle \otimes |1\rangle) = |0\rangle\langle 0|1\rangle \otimes I_2|1\rangle = 0 \otimes |1\rangle = 0.$$

(Note that the last two 0's are *zero vectors*, and not the *index zero*). This means $v = |00\rangle$, and if we perform the B-subsystem measurement, we are guaranteed to have the case 0. Similarly, if the A-subsystem measurement gave the case 1, then the next B-subsystem measurement is guaranteed to give the case 1. (At the end of the day, this is just another way of working out the correlation about $p(ij)$ we discussed above.)

This thought experiment implies that, what we do in a part (measurement in the subsystem A) of a bigger system (the composite system AB) apparently affects outcome of what we can do *independently* in another part (measurement in the subsystem B), even when the two subsystems are far apart, as long as there is an entangled state between the two. This is often called the *EPR paradox*, or *spooky action at a distance*.

This paradox has can be "resolved" in the following way: while there is such a correlation between two independent measurements, this is still not enough to send "information" from subsystem A to B because the measurement gives the case 0 or 1 with equal probability, and that outcome cannot be known beforehand. What we can proactively do is to choose a different PVM, for example

$$E_+^A = |+\rangle\langle +| \otimes I_2, \qquad E_-^A = |-\rangle\langle -| \otimes I_2,$$

instead of E_0^A, E_1^A. If we start with the EPR pair state and do measurement for $E_\pm^A$, and after measuring the case "+", the state would change to $|{+}{+}\rangle$. This can be seen from

$$E_+^A(w) = \frac{1}{\sqrt{2}}\Big((|+\rangle\langle +| \otimes I_2)(|00\rangle + |11\rangle)\Big) = \frac{1}{2\sqrt{2}}(\sum_{i,j=0}^{1}|ij\rangle) = \frac{1}{\sqrt{2}}|{+}{+}\rangle.$$

However, if we do the measurement of this state in subsystem B with PVM E_0^B, E_1^B, we get the cases 0 and 1 with equal probabilities.

8.4. Computational note for Lecture 8.

[1]:
```
# print options
import sympy as sy
sy.init_printing(use_unicode=True)
```

[2]:
```
from sympy.physics.quantum import TensorProduct
```

Deutsch's algorithm

[3]:
```
# we work with symbolic computation, so use sympy
zeroket = sy.Matrix([1, 0])
oneket = sy.Matrix([0, 1])
plusket = sy.Matrix([1/sy.sqrt(2), 1/sy.sqrt(2)])
minusket = sy.Matrix([1/sy.sqrt(2), -1/sy.sqrt(2)])
```

[4]:
```
zerozeroket = TensorProduct(zeroket, zeroket)
zerooneket = TensorProduct(zeroket, oneket)
onezeroket = TensorProduct(oneket, zeroket)
oneoneket = TensorProduct(oneket, oneket)
```

Helper functions for two-qubit computational basis; see Exercise 1 for the convention

[5]:
```
def comp_basis_coeff(v, ij):
    """Return the coefficient of a two-qubit state in the computational␣
 ↪basis.

    Parameters
    ----------
    v : sympy.Matrix
        SymPy vector with 4 components representing a two-qubit state in␣
 ↪the
        computational basis
    ij : tuple of int
        Pair of integers. Must be one of `(0, 0)`, `(0, 1)`,
        `(1, 0)`, `(1, 1)`.

    Returns
    -------
    sympy.Expr
        Coefficient of `v` corresponding to the basis vector |ij⟩.
    """
    if ij == (0, 0):
        return v[0]
    elif ij == (0, 1):
        return v[1]
    elif ij == (1, 0):
        return v[2]
    return v[3]


def comp_basis_for_index(ij):
```

```
    """Return the two-qubit computational basis vector for a given index.

    Parameters
    ----------
    ij : tuple of int
        Pair of integers. Must be one of `(0, 0)`, `(0, 1)`,
        `(1, 0)`, `(1, 1)`.

    Returns
    -------
    sympy.Matrix
        The corresponding computational basis vector |ij⟩.
    """
    if ij == (0, 0):
        return zerozeroket
    elif ij == (0, 1):
        return zerooneket
    elif ij == (1, 0):
        return onezeroket
    return oneoneket
```

[6]:
```
a, b, c, d = sy.symbols('a00 a01 a10 a11')
test_vec = a * zerozeroket + b * zerooneket + c * onezeroket + d *␣
 ↪oneoneket
test_vec
```

[6]:
$$\begin{bmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{bmatrix}$$

[7]:
```
index_list = [(0, 0), (0, 1), (1, 0), (1, 1)]
[comp_basis_coeff(test_vec, index) for index in index_list]
```

[7]:
$$[a_{00}, a_{01}, a_{10}, a_{11}]$$

[8]:
```
# XOR operation is represented by the ^ operator
0 ^ 0, 0 ^ 1, 1 ^ 0, 1 ^ 1
```

[8]:
$$(0, 1, 1, 0)$$

[9]:
```
def U_f_gate(f, v):
    """Apply the Deutsch oracle gate $U_f$ to a two-qubit state.

    Parameters
    ----------
    f : callable
        Boolean function which takes input 0 or 1, returns 0 or 1.
    v : sympy.Matrix
        SymPy vector with 4 components.

    Returns
```

```
    -------
    sympy.Matrix
        The result of applying the linear transform
        |ij⟩ → |ik⟩ with k = XOR(f(i), j) to v.
    """
    ij_list = [(0, 0), (0, 1), (1, 0), (1, 1)]
    a, b, c, d = [comp_basis_coeff(v, index) for index in ij_list]
    ik_list = [(p[0], f(p[0]) ^ p[1]) for p in ij_list]
    va, vb, vc, vd = [comp_basis_for_index(ik_pair) for ik_pair in␣
↪ik_list]
    return a * va + b * vb + c * vc + d * vd
```

with the constant function $f(i) = 0$, expect $U_f(v) = v$ from $\mathrm{XOR}(0, j) = j$

[10]:
```
def f0(i):
    return 0
```

[11]:
```
U_f_gate(f0, test_vec)
```

[11]:
$$\begin{bmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{bmatrix}$$

with the constant function $f(i) = 1$, expect flip on the second bit from $\mathrm{XOR}(1, j) = \mathrm{NOT}(j)$

[12]:
```
def f1(i):
    return 1
```

[13]:
```
U_f_gate(f1, test_vec)
```

[13]:
$$\begin{bmatrix} a_{01} \\ a_{00} \\ a_{11} \\ a_{10} \end{bmatrix}$$

with the identity function $f(i) = i$, expect CNOT

[14]:
```
def f_id(i):
    return i
```

[15]:
```
U_f_gate(f_id, test_vec)
```

[15]:
$$\begin{bmatrix} a_{00} \\ a_{01} \\ a_{11} \\ a_{10} \end{bmatrix}$$

[16]:
```
# initial vector for Deutsch's algorithm
plusminusket = TensorProduct(plusket, minusket)
plusminusket
```

[16]:
$$\begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \end{bmatrix}$$

U_f gate with f_1 on the input vector $v = |+-\rangle$ gives the output vector $-v$

[17]:
```
U_f_gate(f1, plusminusket)
```

[17]:

$$\begin{bmatrix} -\frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}$$

U_f gate with f_{id} on the input vector $|+-\rangle$ gives the output vector $|--\rangle$

[18]:
```
U_f_gate(f_id, plusminusket), TensorProduct(minusket, minusket)
```

[18]:

$$\left(\begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \end{bmatrix}, \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \end{bmatrix} \right)$$

[19]:
```
E_plus = TensorProduct(plusket * plusket.T, sy.eye(2))
E_minus = TensorProduct(minusket * minusket.T, sy.eye(2))
E_plus, E_minus
```

[19]:

$$\left(\begin{bmatrix} \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix}, \begin{bmatrix} \frac{1}{2} & 0 & -\frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 & -\frac{1}{2} \\ -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & -\frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} \right)$$

[20]:
```
# apply U_f gate for the constant function f1 to the initial input
v_prime = U_f_gate(f1, plusminusket)
# then estimate the probability of observing the + case
p_plus = v_prime.T * E_plus * v_prime
p_plus
```

[20]:

$$\begin{bmatrix} 1 \end{bmatrix}$$

[21]:
```
# apply U_f gate for the identity function f_id to the initial input
v_prime = U_f_gate(f_id, plusminusket)
# then estimate the probability of observing the + case
p_plus = v_prime.T * E_plus * v_prime
p_plus
```

[21]:

$$\begin{bmatrix} 0 \end{bmatrix}$$

9. No-cloning theorem and quantum money scheme

9.1. **No-cloning theorem.** Reference: [NC00, p. 532; Aar18, Section 7.2]

The *no-cloning theorem*[7] from quantum mechanics says:

> we cannot copy a quantum mechanical state, i.e, we cannot construct a transform that takes a state vector v and gives the output $v \otimes v$.

If we try to know the information about v, we end up doing some kind of measurement, and the postulate about measurement says that the state will change to a new one w depending on the outcome of measurement.

More formally, we have the following mathematical claim.

Proposition 9.1. *There is no unitary operator U on $\mathbb{C}^2 \otimes \mathbb{C}^2$ (i.e., 4×4-unitary matrix) satisfying*

$$U(v \otimes |0\rangle) = v \otimes v \tag{9.1}$$

for all unit vectors $v \in \mathbb{C}^2$. (In fact, it is enough to consider two vectors.)

The second component $|0\rangle$ in the input of U above is there to help us formulate the claim about unitary transforms, but it is not particularly important that we use $|0\rangle$, and we could have used any other state vector like $|+\rangle$. A more honest formulation of the above physical idea would be the following.

Theorem 9.2. *Let H be a Hilbert space of dimension at least 2. There is no linear operator S from H to $H \otimes H$ that satisfies*

$$S\rho_v S^\dagger = \rho_v \otimes \rho_v \tag{9.2}$$

with $\rho_v = vv^\dagger$ for all unit vectors $v \in H$. (In fact, it is enough to consider four vectors.)

Proposition 9.1 can be proved in several ways, with different degrees of sophistication (and different possibilities for counterarguments "what if we change the formulation to ...").

Proof 1 for Proposition 9.1. The transform U is a linear operator. Consider $-v$ instead of v. We then have

$$U((-v) \otimes |0\rangle) = U(-(v \otimes |0\rangle)) = -U(v \otimes |0\rangle),$$

while

$$(-v) \otimes (-v) = (-1) \cdot (-1) \cdot v \otimes v = v \otimes v.$$

This shows that there is no linear operator satisfying (9.1). □

Proof 2 for Proposition 9.1. A unitary operator must preserve the inner product:

$$\langle U(v), U(w)\rangle = \langle v, w\rangle.$$

We would then get

$$(U(v \otimes |0\rangle))^\dagger U(w \otimes |0\rangle) = (v \otimes |0\rangle)^\dagger (w \otimes |0\rangle) = v^\dagger w \cdot \langle 0|0\rangle = \langle v, w\rangle \cdot 1,$$

while we also have

$$\langle v \otimes v, w \otimes w\rangle = \langle v, w\rangle \cdot \langle v, w\rangle.$$

Therefore we cannot have (9.1). □

Theorem 9.2 reflects the split of the no-cloning theorem in a better way, since it does not use an auxiliary vector like $|0\rangle$, and avoids "unphysical" argument like comparison of vectors v and $-v$ that should represent the same physical state. The proof also works a bit more generally, for the *quantum channels*, that is, trace-preserving positive linear transforms acting on "the space of linear transforms"; there would be no such transform that can convert ρ_v to $\rho_v \otimes \rho_v$.

[7] J. L. Park, "The concept of transition in quantum mechanics", *Found. Phys.* **1** (1), 23–33 (1970).

Proof of Theorem 9.2. Since H has dimension at least 2, we can choose orthogonal unit vectors v_0 and v_1. We then set

$$v_+ = \frac{1}{\sqrt{2}}(v_0 + v_1), \qquad v_- = \frac{1}{\sqrt{2}}(v_0 - v_1),$$

which are again unit vectors. Of course, these are analogues of the vectors $|0\rangle, |1\rangle, |+\rangle, |-\rangle$ in $\mathbb{C}^2$, and satisfy the same formal relations.

On one hand, the density operators $\rho_0 = v_0 v_0^\dagger$, etc., satisfy

$$\rho_0 + \rho_1 = \rho_+ + \rho_-.$$

(See Figure 4.1 to get the idea.) If we had S satisfying (9.2), the linearity of the correspondence $T \mapsto STS^\dagger$ would give

$$\rho_0 \otimes \rho_0 + \rho_1 \otimes \rho_1 = \rho_+ \otimes \rho_+ + \rho_- \otimes \rho_-,$$

but this equality does not hold.

Concretely, consider the vector $v_0 \otimes v_0$ as the input. The left hand side gives $v_0 \otimes v_0$ as the output by $\rho_0(v_0) = v_0$ and $\rho_1(v_0) = 0$. As for the right hand side, first we observe

$$\rho_+(v_0) = \frac{1}{\sqrt{2}} v_+, \qquad \rho_-(v_0) = \frac{1}{\sqrt{2}} v_-.$$

Consequently, the effect of the right hand side becomes

$$\frac{1}{2}(v_+ \otimes v_+ + v_- \otimes v_-) = \frac{1}{2}(v_0 \otimes v_0 + v_1 \otimes v_1),$$

(this is the analogue of the EPR pair state) which is indeed different from $v_0 \otimes v_0$. □

9.2. **Wiesner's quantum money scheme.** Reference: [Aar18, Section 7.3]

The no-cloning theorem guarantees a type of quantum "authentication scheme"[8] that lets trusted actors to check if the information encoded by quantum states are "tampered" by a third party or not. The basic idea is:

- trusted actors impose a certain predefined consistency condition of the "valid" states;
- the third party will have to perform a measurement with respect to some PVM in order to "look into" the state;
- this measurement would then break the consistency with high probability;
- the trusted actors can check the consistency using a (secret) measurement scheme.

More concretely, this scheme works in the following way:

- a "banknote" has the following data:
  - a string of 0's and 1's: $i_1 \cdots i_N$, to be interpreted as the "serial number" of the note;
  - a N-qubit state $v_1 \otimes \cdots \otimes v_N$, with $v_k \in \mathbb{C}^2$.
- the "bank" knows the following secret: for each $k = 1, \dots N$, the k-th qubit v_k should be measured either in the $(|0\rangle, |1\rangle)$-basis (i.e., with respect to the PVM E_0, E_1), or in the $(|+\rangle, |-\rangle)$-basis (i.e., with respect to the PVM E_+, E_-).
- the "public" knowledge:
  - if the k-th qubit v_k should be measured in the $(|0\rangle, |1\rangle)$-basis, the measurement outcome should directly correspond to i_k;
  - if v_k should be measured in the $(|+\rangle, |-\rangle)$-basis, then the case "+" of measurement outcome should correspond to $i_k = 0$, and the case "−" should correspond to $i_k = 1$.

If you are an honest citizen, you don't mess with the qubit state on the banknote, but you can bring the note to the bank and ask them to check if it is an authentic note or not. Then the bank would measure the qubit on the banknote using the secret measurement scheme: for example if $N = 2$, and if they know that the first qubit should be measured in the $(|0\rangle, |1\rangle)$-basis while the second qubit should be measured in the $(|+\rangle, |-\rangle)$-basis, then they use the PVM

$$E_{0+}, \quad E_{0-}, \quad E_{1+}, \quad E_{1-}$$

[8] S. Wiesner, "Conjugate Coding", *SIGACT News* **15** (1), 78–88 (1983).

and compare the measurement outcome with the serial number on the note. If the outcome is consistent, then the bank can be confident that it is authentic, and return the banknote to the customer.

Suppose you are counterfeiter, and you try to forge a banknote with the serial number 01. Since you don't know the bank's secret, you have to guess measurement schemes for each qubit, and let's say you guessed that both qubits shall be measured in the $(|0\rangle, |1\rangle)$-basis. Then you put the two-qubit state $|01\rangle$ on your counterfeit note. When the bank checks this note, they get the cases $0+$ and $0-$ with equal probability. The case $0+$ corresponds to the serial number 00, so they would reject your note in this case, while if they get the case $0-$ then they are going to be fooled.

If we increase N, the counterfeiter have to assume that they guess the wrong measurement scheme for half of the qubits. Among those $\frac{N}{2}$ qubits, each qubit has the $\frac{1}{2}$ probability of getting flagged as incompatible with the corresponding digit of the serial number. Since measurement of each qubit needs to be consistent with the corresponding bit in the serial number, the rough estimate of the probability of a counterfeit bill passing the bank's check is $(\frac{1}{2})^{\frac{N}{2}}$, which becomes below 0.1% at $N = 20$. (A more rigorous calculation gives the probability estimate $(\frac{3}{4})^N$, but it is still an exponential decay with a similar base because of $(\frac{3}{4})^2 = \frac{9}{16} \sim \frac{1}{2}$.)

If the qubits can be cloned (which is forbidden by the no-cloning theorem), the counterfeiter can do a better job at estimating the bank's measurement scheme. Suppose the authentic banknote with serial number 01 has a state $v = v_0 \otimes v_1$, which is $|0-\rangle$ under the above scheme. The cloned state would look like

$$|0\rangle \otimes |-\rangle \otimes \cdots \otimes |-\rangle = |0-\cdots-\rangle .$$

The counterfeiter doing measurement by PVM E_0, E_1 on the second, third, ..., k-th qubit would give results $i_2, \ldots, i_k = 0, 1$ and the new state vector

$$|0\rangle \otimes |i_2\rangle \otimes \cdots \otimes |i_k\rangle \otimes |-\rangle \otimes \cdots \otimes |-\rangle ,$$

with equal probability $(\frac{1}{2})^{k-1}$ for all possible choices of i_j's. Thus, typically the counterfeiter observes random behavior, and can conclude that the second qubit should be instead measured in $E_\pm$. If the counterfeiter uses the PVM $E_\pm$ to do the measurement of cloned state, they see the consistent result "$-$" and then concludes that the second qubit should be measured $E_\pm$.

10. Quantum key distribution

Reference: [NC00, Section 12.6.3; Aar18, Section 8.2]

We get more interesting multi-qubit problems when two actors cooperate to get a desired outcome. The *key-distribution problem* asks how two parties (that we call Alice and Bob) can establish a shared secret through insecure communication channel, which can be eavesdropped by another party (that we call Eve). Let us look at the first algorithm by Bennett and Brassard[9] that solved this problem using quantum mechanical ideas, which is often called the *BB84 protocol*.

Here is the setting:

- Alice can send both quantum mechanical states (unit vectors of a Hilbert space) and classical information (string of bits) to Bob, without error;
- Bob can send classical information to Alice, again without error;
- they can acknowledge receiving such communication;
- Eve can eavesdrop the communication and look into the classical information and forward it without change;
- if she tries to look into the quantum mechanical states, she ends up performing a measurement;
- Alice and Bob can be confident that the classical information came from each other without change, although they don't not know whether Eve looked into it or not.

The protocol starts with Alice randomly choosing a candidate of shared secret and a encoding / decoding method, while Bob also randomly choosing a encoding / decoding method:

[9] C. H. Bennett and G. Brassard. "Quantum cryptography: Public key distribution and coin tossing". In Proceedings of IEEE International Conference on Computers, Systems and Signal Processing, pp. 175–179, IEEE, New York, 1984. Bangalore, India, December 1984.

(1) control parameters are integers $m < n$, that Alice and Bob share;
(2) Alice chooses two strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_n$ of 0's and 1's. This choice is random;
(3) she then prepares an n-qubit state $v = v_1 \otimes \cdots \otimes v_n$ representing x according to the following rule:
   - if $y_j = 0$, then $v_j = |0\rangle$ or $|1\rangle$ depending on $x_j = 0$ or not;
   - if $y_j = 1$, then $v_j = |+\rangle$ or $|-\rangle$ depending on $x_j = 0$ or not.
(4) on Bob's side, he chooses a string $y' = y'_1 \cdots y'_n$ of 0's and 1's, again in a random way.

Then Alice sends the n-qubit encoding of the candidate for secret, and next they exchange the encoding / decoding methods to settle on an actual candidate of shared secret:

(5) Alice sends the state v to Bob;
(6) Bob makes the measurement of the factors v_j by the following rule:
   - if $y'_j = 0$, the j-th qubit is measured in the computational basis, i.e., with the PVM $E_0 = |0\rangle\langle 0|$ and $E_1 = |1\rangle\langle 1|$;
   - if $y'_j = 1$, the j-th qubit is measured in the $(|+\rangle, |-\rangle)$-basis, i.e., with $E_+ = |+\rangle\langle +|$ and $E_- = |-\rangle\langle -|$.
(7) he records the result of measurement as the string $x' = x'_1 \cdots x'_n$ of 0's and 1's, where the case "+" corresponds to 0 and the case "−" corresponds to 1 in the case of $y'_j = 1$;
(8) Alice and Bob exchange the strings y and y';
(9) now they know the j's satisfying $y_j = y'_j$. Let's write $I_=$ for the collection of such j's;
(10) now Alice's string $(x_j)_{j \in I_=}$ and Bob's string $(x'_j)_{j \in I_=}$ are candidates of their shared secret.

Let us analyze what this protocol does for Alice and Bob up to this point:

- typically, the length of the shared secret is about $\frac{n}{2}$, since each j has the probability $\frac{1}{2}$ of having $y_j = y'_j$;
- if Eve did not look into v, we have $x_j = x'_j$ for $j \in I_=$ because v arrived unchanged and Bob is using the right measurement scheme on v_j for such j's.

Eve can transparently intercept the classical communication at step 8 and know the collection $I_=$. However, in order to have known the corresponding bits x_j, she will have to intercept the state v at step 5 and perform measurements. At this point the strings y and y' are not sent yet, so she needs to guess which PVM to use for each v_j. (If Eve could copy the state v, she can keep the copy until she knows y and y', and then do the right measurement, but this is forbidden by the no-cloning theorem.)

For example, suppose we had $y_1 = 0 = y'_1$ and $x_1 = 0$ so $v_1 = |0\rangle$, but Eve guessed that this qubit shall be measured in the $(|+\rangle, |-\rangle)$-basis. Then she observes the case "+" or the case "−", and the state v_1 ends up being $|+\rangle$ or $|-\rangle$. In each case, Bob will measure the case 0 or case 1 with probability $\frac{1}{2}$ for each. If Eve makes the right guess, then she will observe the case 0, and v_1 remains as $|0\rangle$, so Bob will also observe the case 0. That means, for each j such that $y_j = y'_j$, the probability of having $x_j \neq x'_j$ is $\frac{1}{4}$.

Now, here is the remaining part of the protocol, which lets them decide if the candidate of the shared secret is good or not:

(11) Alice randomly chooses some $j_1, \ldots, j_m$ from $I_=$;
(12) she sends the pairs (j_i, x_{j_i}) to Bob;
(13) Bob checks these x_{j_i} against the corresponding terms in his x';
(14) if he sees any discrepancy, he reports it to Alice. (Then they know that the shared secret is bad and should be discarded.)
(15) otherwise they can keep the remaining terms of $(x_j)_{j \in I_=}$ and $(x'_j)_{j \in I_=}$ as valid shared secret which is not known to Eve.

Now, if Eve has eavesdropped at step 5, then discrepancy for the j_i-th bit happens with probability $\frac{1}{4}$, or in other words Bob does not see discrepancy for this bit with probability $\frac{3}{4}$. That means the overall probability of Bob overlooking Eve's snooping is $(\frac{3}{4})^m$, which can be exponentially small for large m. If n is even larger, they can expect to end up with a shared secret string of length about $\frac{n}{2} - m$.

10.1. **Computational note for Lecture 10.** The BB84 key distribution protocol

[1]:
```
import numpy as np
np.set_printoptions(suppress=True, legacy='1.25')
rng = np.random.default_rng()
```

[2]:
```
# from Lecture 7
def col_vec(components):
    """Return a column vector (as a NumPy array) with the given␣
  ↪components.

    In a matrix context, this should be treated as a column vector.
    """
    return np.array(components)


def col_vec_inn_prod(v1, v2):
    """Hilbert space inner product of column vectors."""
    return np.conjugate(v1).dot(v2)


zeroket = col_vec([1, 0])
oneket = col_vec([0, 1])
plusket = col_vec([1, 1]) * 1/np.sqrt(2)
minusket = col_vec([1, -1]) * 1/np.sqrt(2)
eyeket = col_vec([1, 1j]) * 1/np.sqrt(2)
```

10.1.1. *preparing strings and states.* control parameter is an integer n, that Alice and Bob share;

[3]:
```
m = 5
n = 20
```

Alice chooses two strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_n$ of 0's and 1's. This choice is random;

[4]:
```
x = [rng.integers(2) for _ in range(n)]
y = [rng.integers(2) for _ in range(n)]
x, y
```

[4]:
```
([0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0],
 [0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0])
```

she then prepares an n-qubits $v_1, \ldots, v_n$ representing x according to the following rule: - if $y_j = 0$, then $v_j = |0\rangle$ or $|1\rangle$ depending on $x_j = 0$ or not; - if $y_j = 1$, then $v_j = |+\rangle$ or $|-\rangle$ depending on $x_j = 0$ or not;

[5]:
```
def v_enc(x_val, y_val):
    if y_val == 0:
        if x_val == 0:
            return zeroket
        else:
            return oneket
    else:
        if x_val == 0:
```

```
            return plusket
    return minusket
```

[6]:
```
v = [v_enc(x[i], y[i]) for i in range(n)]
v
```

[6]:
```
[array([1, 0]),
 array([0.70710678, 0.70710678]),
 array([0.70710678, 0.70710678]),
 array([0.70710678, 0.70710678]),
 array([0, 1]),
 array([0.70710678, 0.70710678]),
 array([1, 0]),
 array([1, 0]),
 array([0, 1]),
 array([1, 0]),
 array([0.70710678, 0.70710678]),
 array([1, 0]),
 array([0, 1]),
 array([0, 1]),
 array([0.70710678, 0.70710678]),
 array([0.70710678, 0.70710678]),
 array([0, 1]),
 array([0.70710678, 0.70710678]),
 array([0.70710678, 0.70710678]),
 array([1, 0])]
```

on Bob's side, he chooses a string $y' = y'_1 \cdots y'_n$ of 0's and 1's, again in a random way.

[7]:
```
yp = [rng.integers(2) for _ in range(n)]
yp
```

[7]:
```
[1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1]
```

10.1.2. *transmit states and strings, without eavesdropping.* Alice sends the state v to Bob;

Bob makes the measurement of the factors v_j by the following rule: - if $y'_j = 0$, the j-th qubit is measured in the computational basis, i.e., with the PVM $E_0 = |0\rangle\langle 0|$ and $E_1 = |1\rangle\langle 1|$; - if $y'_j = 1$, the j-th qubit is measured in the $(|+\rangle, |-\rangle)$-basis, i.e., with $E_+ = |+\rangle\langle +|$ and $E_- = |-\rangle\langle -|$.

he records the result of measurement as the string $x' = x'_1 \cdots x'_n$ of 0's and 1's, where the case "+" corresponds to 0 and the case "−" corresponds to 1 in the case of $y'_j = 1$;

[8]:
```
def measure_qubit(v, basis_choice):
    """Measure a single-qubit state vector in a specified basis.

    Parameters
    ----------
    v : numpy.ndarray
        Qubit state, a length-2 NumPy array (complex amplitudes).
    basis_choice : int
        0 for computational basis, 1 for the |+⟩ / |−⟩ basis
```

```
    Returns
    -------
    int
        0 or 1, according to the measurement of v in the specified basis.
        The case "+" corresponds to 0.
    """
    # to be on the safe side, do not assume ||v|| = 1
    # normalize the input vector
    norm_v = np.sqrt(np.real(col_vec_inn_prod(v, v)))
    v_normalized = v / norm_v
    if basis_choice == 0:
        # the case 0 happens with probability |v0|^2
        threshold = np.real(np.conjugate(v_normalized[0]) *␣
↪v_normalized[0])
        rand_num = rng.random()
        if rand_num < threshold:
            return 0
        else:
            return 1
    else:
        # the case "+" happens with probability |⟨|+⟩,v⟩|^2
        inn_prod_with_plus_ket = col_vec_inn_prod(plusket, v_normalized)
        threshold = np.abs(inn_prod_with_plus_ket)**2
        rand_num = rng.random()
        if rand_num < threshold:
            return 0
        else:
            return 1
```

[9]:
```
xp = [measure_qubit(v[i], yp[i]) for i in range(n)]
xp
```

[9]:
```
[0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1]
```

Alice and Bob exchange the strings y and y';
now they know the j's satisfying $y_j = y'_j$, call them $j_1, \dots, j_k$; now Alice's string $x_{j_1} \cdots x_{j_k}$ and Bob's string $x'_{j_1} \cdots x'_{j_k}$ are candidates of their shared secret.

[10]:
```
matching_indices = [i for i in range(n) if y[i] == yp[i]]
shared_secret_Alice = [x[i] for i in matching_indices]
shared_secret_Bob = [xp[i] for i in matching_indices]
matching_indices, shared_secret_Alice, shared_secret_Bob
```

[10]:
```
([1, 2, 3, 4, 5, 6, 8, 9, 10, 13, 15, 17],
 [0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0])
```

10.1.3. *transmit states and strings, with eavesdropping.* Alice sends the state v to Bob;
Eve looks into the state, but needs to make guess about measurement schemes

[11]:
```
y_eve = [rng.integers(2) for _ in range(n)]
```

when she performs the measurement, the state changes according to the outcome

[12]:
```
x_eve = [measure_qubit(v[i], y_eve[i]) for i in range(n)]
v = [v_enc(x_eve[i], y_eve[i]) for i in range(n)]
v
```

[12]:
```
[array([0.70710678, 0.70710678]),
 array([0, 1]),
 array([0, 1]),
 array([0.70710678, 0.70710678]),
 array([0, 1]),
 array([0, 1]),
 array([ 0.70710678, -0.70710678]),
 array([ 0.70710678, -0.70710678]),
 array([0, 1]),
 array([ 0.70710678, -0.70710678]),
 array([0, 1]),
 array([1, 0]),
 array([0.70710678, 0.70710678]),
 array([0, 1]),
 array([0.70710678, 0.70710678]),
 array([0, 1]),
 array([0, 1]),
 array([0.70710678, 0.70710678]),
 array([1, 0]),
 array([0.70710678, 0.70710678])]
```

Bob makes the measurement of the factors v_j by the following rule: - if $y'_j = 0$, the j-th qubit is measured in the computational basis, i.e., with the PVM $E_0 = |0\rangle\langle 0|$ and $E_1 = |1\rangle\langle 1|$; - if $y'_j = 1$, the j-th qubit is measured in the $(|+\rangle, |-\rangle)$-basis, i.e., with $E_+ = |+\rangle\langle +|$ and $E_- = |-\rangle\langle -|$.

he records the result of measurement as the string $x' = x'_1 \cdots x'_n$ of 0's and 1's, where the case "+" corresponds to 0 and the case "−" corresponds to 1 in the case of $y'_j = 1$;

[13]:
```
xp = [measure_qubit(v[i], yp[i]) for i in range(n)]
xp
```

[13]:
```
[0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0]
```

Alice and Bob exchange the strings y and y';

now they know the j's satisfying $y_j = y'_j$, call them $j_1, \ldots, j_k$; now Alice's string $x_{j_1} \cdots x_{j_k}$ and Bob's string $x'_{j_1} \cdots x'_{j_k}$ are candidates of their shared secret.

[14]:
```
matching_indices = [i for i in range(n) if y[i] == yp[i]]
shared_secret_Alice = [x[i] for i in matching_indices]
shared_secret_Bob = [xp[i] for i in matching_indices]
matching_indices, shared_secret_Alice, shared_secret_Bob
```

[14]:
```
([1, 2, 3, 4, 5, 6, 8, 9, 10, 13, 15, 17],
 [0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0],
 [1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0])
```

10.1.4. *verify if there was eavesdropping or not.* Alice randomly chooses some $j_{a_1}, \ldots, j_{a_m}$ among $j_1, \ldots, j_k$; she sends the pairs (j_{a_s}, z_{a_s}) to Bob;

[15]:
```
sample_indices = rng.choice(matching_indices, m, replace=False)
sample_secret_Alice = [x[i] for i in sample_indices]
sample_indices, sample_secret_Alice
```

[15]:
```
(array([17, 10,  9,  1, 15]), [0, 0, 0, 0, 0])
```

Bob checks these z_{a_s} against the corresponding terms in his z'; if he sees any discrepancy, he reports it to Alice. (Then they know that the shared secret is bad and should be discarded.)

[16]:
```
sample_secret_Bob = [xp[i] for i in sample_indices]
sample_secret_Bob, sample_secret_Alice == sample_secret_Bob
```

[16]:
```
([0, 1, 1, 1, 0], False)
```

otherwise they can keep the remaining terms of z and z' as valid shared secret which is not known to Eve.

[17]:
```
remaining_secret_Alice = [x[i] for i in matching_indices
                          if i not in sample_indices]
remaining_secret_Bob = [xp[i] for i in matching_indices
                        if i not in sample_indices]
remaining_secret_Alice, remaining_secret_Bob
```

[17]:
```
([0, 0, 1, 0, 0, 1, 1], [1, 0, 1, 0, 0, 1, 1])
```

11. Superdense coding and quantum teleportation

Recall that entangled states are given by unit vectors $w \in \mathbb{C}^2 \otimes \mathbb{C}^2$ that cannot be written as $w = u \otimes v$ for some $u, v \in \mathbb{C}^2$ (Section 8.2). Such states can be used for interesting communication protocols.

11.1. **Superdense coding.** Reference: [NC00, Section 2.3; Aar18, Sections 9]

The *superdense coding* protocol developed by Wiesner and Bennett around 1970[10], and experimentally confirmed in 1996[11], was the first "communication protocol" that used the idea of entanglement to get something better than the classical schemes.

Similar to the quantum key distribution protocol, there are two players in the protocol that we call Alice and Bob. Their goal is to transmit classical information encoded by *two bits* $s = s_1 s_2$ ($s_i = 0, 1$) from Alice to Bob. The protocol achieves this task by

- initial entangled state between Alice and Bob;
- operations that can be independently performed by them;
- transfer of one qubit from Alice to Bob.

So it is enough to send one qubit state instead of two classical bits.

Here is how the protocol is defined:

(1) Alice and Bob prepares the EPR pair state (8.1)
$$w = w_{\text{EPR}} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \in H_A \otimes H_B$$
with $H_A = H_B = \mathbb{C}^2$;
(2) the message string Alice wants to send out is $s = s_1 s_2$, with $s_i = 0, 1$;
(3) if $s_1 = 1$, Alice applies the Pauli X transform
$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$
on her part of w (the H_A factor);
(4) if $s_2 = 1$, she then applies the Pauli Z transform
$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$
on her part of the state;
(5) then she sends her part of the state to Bob (this means Bob now has access to the overall state in $H_A \otimes H_B$);
(6) Bob does measurement for the projection-valued measure corresponding to the basis
$$w_{00} = w, \qquad w_{10} = (X \otimes I_2)w, \qquad w_{01} = (Z \otimes I_2)w, \qquad w_{11} = (ZX \otimes I_2)w;$$
(7) the (index of) result of the measurement is guaranteed to be $s_1 s_2$, what Alice wanted to send.

To make sure that the above protocol is well-defined, we want to check that Bob has a good PVM at step 6. Expanding the actions of X and Z, we get

$$w_{00} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \quad w_{10} = \frac{1}{\sqrt{2}}(|10\rangle + |01\rangle), \quad w_{01} = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle), \quad w_{11} = \frac{1}{\sqrt{2}}(-|10\rangle + |01\rangle),$$

which form an orthonormal basis of the 4-dimensional Hilbert space $\mathbb{C}^2 \otimes \mathbb{C}^2$. This means that the operators $E_{ij} = w_{ij} w_{ij}^\dagger$ indeed form a PVM. Then, at step 7 he is guaranteed to get $s_1 s_2$ since the basis vectors he used was exactly what he has after step 5.

Why does this protocol work? We see that Alice is using one of the four unitary transforms I_2, X, Z, ZX on her qubit according to the string $s = s_1 s_2$. Since ZX is equal to $-iY$ for the Pauli matrix Y, we are working with the three Pauli matrices and the identity transform on $\mathbb{C}^2$.

---

[10] O. Sattath, "Stephen Wiesner" (2021), blog post at *My quantum thoughts*, Or Sattath's blog.

[11] K. Mattle, H. Weinfurter, P. G. Kwiat, and A. Zeilinger, "Dense coding in experimental quantum communication", *Phys. Rev. Lett.* **76** (25), 4656–4659 (1996).

In fact, we can make the space of complex 2×2-matrices $M_2(\mathbb{C})$ a Hilbert space with the inner product (the normalized *Hilbert–Schmidt inner product*)

$$\langle S, T \rangle = \frac{1}{2} \operatorname{tr}(S^\dagger T).$$

(The normalization factor $\frac{1}{2}$ makes unitary matrices unit vectors.)

Then, on the one hand, the correspondence $T \mapsto (T \otimes I_2)(w_{\text{EPR}})$ is a unitary transform from $M_2(\mathbb{C})$ to $\mathbb{C}^2 \otimes \mathbb{C}^2$:

$$\langle S, T \rangle = \langle (S \otimes I_2)(w_{\text{EPR}}), (T \otimes I_2)(w_{\text{EPR}}) \rangle.$$

On the other hand, the matrices I_2, X, Y, Z form an orthonormal basis of $M_2(\mathbb{C})$. Combining these two, we see that Alice's "local operation" can indeed create enough interesting states in the two-qubit system. This can be generalized to higher-dimensional Hilbert spaces $\mathbb{C}^n$ by looking at *orthonormal basis of unitary matrices* in $M_n(\mathbb{C})$.

11.2. **Comparison with classical communication channels.** While the "speedup" of the above protocol may not sound impressive, if we can do this at a large scale, it does beat the central concept of *channel capacity* of communication channels in information theory[12], which roughly says the following:

> in any "classical" communication channel, in order to send n bits worth of information we need to use at least n bits of transfer.

More formally, a *compression scheme of rate R* is a way to take a string of "message letters" $x_1 \cdots x_n$ of arbitrary length n, and get sequence of bits $C(x) = s_1 \cdots s_m$ with $m \sim Rn$, and a *decompression scheme of rate R* is a way to take an arbitrary sequence of bits $s_1 \cdots s_m$ with $m \sim Rn$ and get a string $D(s) = x_1 \cdots x_n$. We say such a compression-decompression scheme is *reliable* if we have $D(C(x)) = x$ with probability arbitrarily close to 1, as n approaches ∞.

When X is a discrete random variable, its (base 2) *entropy* is defined as

$$H(X) = -\sum_x p_X(x) \log_2 p_X(x),$$

where we take the sum for all possible values x for X, and $p_X(x) = \mathbb{P}[X = x]$ is the probability of X taking the value x.

Theorem 11.1 ([NC00, Theorem 12.4])**.** *Given $R > H(X)$, there is a reliable compression-decompression scheme of rate R. On the other hand, there is no reliable compression-decompression scheme of rate R for $R < H(X)$.*

For example, if the input letters are also bits, 0 or 1, and if we expect them to happen at equal probability $\frac{1}{2}$, then the entropy would be $H(X) = 1$. The above theorem implies there is no reliable encoding-decoding scheme of rate $\frac{1}{2}$, so we cannot just use $\frac{n}{2}$ bits of transmission to reliably transfer n bits. However, the superdense coding protocol does let us transfer n bits of information with $\frac{n}{2}$-qubit transfer!

Of course, you might want to object that a qubit probably has more "information capacity" than a classical bit, equivalent of several bits bundled together. For example, in a classical simulation of qubit $v = (z_1, z_2) \in \mathbb{C}^2$, we can use single-precision floating-point numbers (typically represented by 32 bits each) x_i and y_i for $i = 1, 2$ representing the real and imaginary parts of z_i. However, the *Holevo bound* inequality [NC00, Theorem 12.1] implies that Alice sending qubits to Bob without background entanglement will not improve the above rate for classical communication.

[12] C. E. Shannon, *A mathematical theory of communication*, Bell System Tech. J. **27** (1948), 379–423, 623–656.

11.3. **Quantum teleportation.** Reference: [NC00, Section 1.3.7; Aar18, Sections 10.1]

The entanglement can be also used to "transport" a quantum mechanical state by transmitting classical information. The *quantum teleportation* protocol[13] is a concrete protocol to send a qubit state through a shared EPR state and two classical bits of communication. This was experimentally achieved in 1997[14,15].

Let us describe the protocol:

(1) Alice and Bob prepares the EPR pair state
$$w = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \in H_{A2} \otimes H_B$$
with $H_{A2} = H_B = \mathbb{C}^2$;
(2) Alice wants to send a qubit state $v \in H_{A1} = \mathbb{C}^2$ to Bob;
(3) we now have $v \otimes w$ in the composite system $H_{A1} \otimes H_{A2} \otimes H_B$;
(4) on her part of the system $H_{A1} \otimes H_{A2}$, Alice first performs the CNOT transform;
(5) then she performs the Hadamard transform on the H_{A1} part;
(6) Alice measures her part $H_{A1} \otimes H_{A2}$ with PVM for the computational basis, i.e., $E_{ij} = |i\rangle\langle i| \otimes |j\rangle\langle j|$ for $i, j = 0, 1$;
(7) she sends the outcome $s_1 s_2$ to Bob;
(8) Bob first looks at s_1, and if it is 1, then he applies the Pauli Z transform to his part of the state;
(9) he next looks at s_2, and if it is 1, then he applies the Pauli X transform;
(10) now Bob's state is guaranteed to be v.

Let us analyze what happens under this protocol. We write the qubit v as $v = (\alpha, \beta)$, or in other words $v = \alpha|0\rangle + \beta|1\rangle$. At step 3, the composite state is
$$v \otimes w = \frac{1}{\sqrt{2}}(\alpha(|000\rangle + |011\rangle) + \beta(|100\rangle + |111\rangle)).$$
Then after step 4, the state becomes
$$\frac{1}{\sqrt{2}}(\alpha(|000\rangle + |011\rangle) + \beta(|110\rangle + |101\rangle)).$$
Since the Hadamard gate transforms $|0\rangle$ to $|+\rangle$ and $|1\rangle$ to $|-\rangle$, after step 5 the state is
$$u = \frac{1}{\sqrt{2}}(\alpha(|+00\rangle + |+11\rangle) + \beta(|-10\rangle + |-01\rangle)).$$

Suppose Alice's measurement gave the result 00, so Bob will not do anything. After the measurement the state became the normalization of $(E_{00} \otimes I_2)(u)$. We can compute
$$(E_{00} \otimes I_2)(u) = \frac{1}{2}(\alpha|000\rangle + \beta|001\rangle),$$
so its normalization is
$$(\alpha|000\rangle + \beta|001\rangle) = |00\rangle \otimes v.$$
This means Bob's state is v.

Next consider the case 10 for Alice's measurement, so Bob will apply the Z gate. Then the state became the normalization of $(E_{10} \otimes I_2)(u)$, which can be computed as
$$(E_{10} \otimes I_2)(u) = \frac{1}{2}(\alpha|000\rangle - \beta|001\rangle).$$
Then the normalization removes the factor $\frac{1}{2}$. The Z gate corrects the sign for $|001\rangle$, and again we get the state $|00\rangle \otimes v$.

[13] C. H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, and W. K. Wootters, "Teleporting an unknown quantum state via dual classical and Einstein–Podolsky–Rosen channels", *Phys. Rev. Lett.* **70** (13), 1895–1899 (1993).

[14] D. Boschi, S. Branca, F. De Martini, L. Hardy, and S. Popescu, "Experimental realization of teleporting an unknown pure quantum state via dual classical and Einstein–Podolsky–Rosen channels", *Phys. Rev. Lett.* **80** (6), 1121–1125 (1998).

[15] D. Bouwmeester, J.-W. Pan, K. Mattle, M. Eibl, H. Weinfurter, and A. Zeilinger, "Experimental quantum teleportation", *Nature* **390** (6660), 575–579 (1997).

Next consider the case 01 for Alice's measurement, so Bob will apply the X gate. In this case the state became the normalization of $(E_{01} \otimes I_2)(u)$. We compute

$$(E_{01} \otimes I_2)(u) = \frac{1}{2}\left(\alpha|011\rangle + \beta|010\rangle\right),$$

so its normalization is $|01\rangle \otimes (\alpha|1\rangle + \beta|0\rangle)$. Then the X gate swaps $|0\rangle$ and $|1\rangle$ on the last factor, so we have $|01\rangle \otimes v$ at the end, and Bob has the state v again.

Finally, suppose Alice got the case 11. Then the state became the normalization of

$$(E_{11} \otimes I_2)(u) = \frac{1}{2}\left(\alpha|111\rangle - \beta|110\rangle\right),$$

which is $|11\rangle \otimes (\alpha|1\rangle - \beta|0\rangle)$. When Bob applies the Z gate, the state becomes $|11\rangle \otimes (-\alpha|1\rangle - \beta|0\rangle) = -|11\rangle \otimes (\alpha|1\rangle + \beta|0\rangle)$. Then he applies the X gate, and the final state is $-|11\rangle \otimes v$, so his part of the state is indeed v.

11.4. Computational note for Lecture 11.

11.4.1. *superdense coding.*

[1]:
```
import sympy as sy
sy.init_printing(use_unicode=True)
```

[2]:
```
from sympy.physics.quantum import TensorProduct
```

[3]:
```
zeroket = sy.Matrix([1, 0])
oneket = sy.Matrix([0, 1])
EPR_state = (TensorProduct(zeroket, zeroket) +
             TensorProduct(oneket, oneket)) * (1/sy.sqrt(2))
```

[4]:
```
# keep track of the state
w = EPR_state
```

message is a string of two bits

[5]:
```
s = [0, 1]
```

if $s_1 = 1$, Alice applies the Pauli X transform on her part of w (the H_A factor)

[6]:
```
X = sy.Matrix([[0, 1],
               [1, 0]])
# indexing starts with 0
if s[0] == 1:
    w = TensorProduct(X, sy.eye(2)) * w
```

if $s_2 = 1$, she then applies the Pauli Z transform on her part of the state

[7]:
```
Z = sy.Matrix([[1, 0],
               [0, -1]])
# indexing starts with 0
if s[1] == 1:
    w = TensorProduct(Z, sy.eye(2)) * w
```

then she sends her part of the state to Bob; this means Bob now has access to the overall state in $H_A \otimes H_B$

Bob does measurement for the projection-valued measure corresponding to the basis

$$w_{00} = w_{\text{EPR}}, \quad w_{10} = (X \otimes I_2)w_{\text{EPR}}, \quad w_{01} = (Z \otimes I_2)w_{\text{EPR}}, \quad w_{11} = (ZX \otimes I_2)w_{\text{EPR}}$$

[8]:
```
w00 = EPR_state
w10 = TensorProduct(X, sy.eye(2)) * w00
w01 = TensorProduct(Z, sy.eye(2)) * w00
w11 = TensorProduct(Z * X, sy.eye(2)) * w00
```

the (index of) result of the measurement is guaranteed to be $s_1 s_2$, what Alice wanted to send

[9]:
```
def inn_prod_abs_sq(w1, w2):
    """Return the squared absolute value of the inner product ⟨w1, w2⟩.

    Parameters
    ----------
    w1 : sympy.Matrix or sympy.ImmutableMatrix
```

```
        SymPy column matrix representing a (possibly complex) vector.
    w2 : sympy.Matrix or sympy.ImmutableMatrix
        SymPy column matrix representing a (possibly complex) vector.

    Returns
    -------
    sympy.Expr
        The scalar |⟨w1, w2⟩|² as a SymPy expression.
    """
    inn_prod = sy.conjugate(w1).T * w2
    abs_sq = sy.conjugate(inn_prod) * inn_prod
    return abs_sq[0]
```

[10]:
```
# measurement probabilities are square of absolute value of inner product
inn_prod_abs_sq(w00, w), inn_prod_abs_sq(w10, w), \
    inn_prod_abs_sq(w01, w), inn_prod_abs_sq(w11, w)
```

[10]: $(0,\ 0,\ 1,\ 0)$

check consistency of the above scheme; we want Bob's basis to be orthogonal

[11]:
```
inn_prod_abs_sq(w00, w10), inn_prod_abs_sq(w00, w01), \
    inn_prod_abs_sq(w00, w11), inn_prod_abs_sq(w10, w01), \
    inn_prod_abs_sq(w10, w11), inn_prod_abs_sq(w01, w11)
```

[11]: $(0,\ 0,\ 0,\ 0,\ 0,\ 0)$

11.4.2. *quantum teleportation.* Alice wants to send a qubit state $v \in H_{A1} = \mathbb{C}^2$ to Bob

[12]:
```
α, β = sy.symbols("α β")
v = sy.Matrix([α, β])
```

we take $v \otimes w_{\text{EPR}}$ in the composite system $H_{A1} \otimes H_{A2} \otimes H_B$;

[13]:
```
tot_state = TensorProduct(v, EPR_state)
```

on her part of the system $H_{A1} \otimes H_{A2}$, Alice first performs the CNOT transform;

[14]:
```
CNOT = sy.Matrix([[1, 0, 0, 0],
                  [0, 1, 0, 0],
                  [0, 0, 0, 1],
                  [0, 0, 1, 0]])
tot_state = TensorProduct(CNOT, sy.eye(2)) * tot_state
```

then she performs the Hadamard transform on the H_{A1} part;

[15]:
```
H = sy.Matrix([[1, 1],
               [1, -1]]) * (1/sy.sqrt(2))
tot_state = TensorProduct(H, sy.eye(4)) * tot_state
```

Alice measures her part $H_{A1} \otimes H_{A2}$ with PVM for the computational basis

[16]:
```
zerozeroket = TensorProduct(zeroket, zeroket)
onezeroket = TensorProduct(oneket, zeroket)
zerooneket = TensorProduct(zeroket, oneket)
oneoneket = TensorProduct(oneket, oneket)
E00 = zerozeroket.reshape(4, 1) * zerozeroket.reshape(1, 4)
```

```
E01 = zerooneket.reshape(4, 1) * zerooneket.reshape(1, 4)
E10 = onezeroket.reshape(4, 1) * onezeroket.reshape(1, 4)
E11 = oneoneket.reshape(4, 1) * oneoneket.reshape(1, 4)
```

[17]:
```
def E(s):
    """Convenience function to get E_ij for s = ij."""
    if s == [0, 0]:
        return E00
    elif s == [0, 1]:
        return E01
    elif s == [1, 0]:
        return E10
    return E11
```

she sends the outcome $s_1 s_2$ to Bob;

[18]:
```
s = [1, 0]  # choose s
# the factor of 2 is to get normalized vector
new_state = TensorProduct(E(s), sy.eye(2)) * tot_state * 2
```

Bob first looks at s_1, and if it is 1, then he applies the Pauli Z transform to his part of the state

[19]:
```
if s[0] == 1:
    new_state = TensorProduct(sy.eye(4), Z) * new_state
```

he next looks at s_2, and if it is 1, then he applies the Pauli X transform;

[20]:
```
if s[1] == 1:
    new_state = TensorProduct(sy.eye(4), X) * new_state
```

now Bob's state is guaranteed to be v.

[21]:
```
new_state
```

[21]:
$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \alpha \\ \beta \\ 0 \\ 0 \end{bmatrix}$$

12. The Clauser–Horne–Shimony–Holt game

Reference: [Aar18, Sections 13.2, 14]

We can understand the power of entangled states by designing "quantum strategies" for two cooperating players that tries to win a game in a better way than the usual algorithms by using entangled states. The most famous proposal is the *CHSH game* after a work of Clauser, Horne, Shimony, and Holt from 1969[16] that refined an earlier thought experiment by Bell (footnote 8.2) on the EPR paradox. This was realized in experiments in various settings since early 80's[17,18], and more recently realizing Bell's original vision for entangled electrons over a long distance[19].

The game is set up as follows:

- there are two (cooperating) players, Alice and Bob;
- there is a "referee", Charlie, who asks them questions and evaluates their answers;
- Alice and Bob can set up a strategy using classical random variables ("coin toss" to decide something) and operations on a prepared quantum mechanical states before the game starts;
- but they cannot communicate information (neither classical nor quantum) once the game starts;
- Charlie's question is one bit $x = 0, 1$ to Alice, and another bit $y = 0, 1$ to Bob;
- Alice answers with one bit $a = 0, 1$, and Bob similarly answers with one bit $b = 0, 1$ (without communication);
- they "win" if $a + b = xy$ holds modulo 2 (meaning the even / odd parity matches between $a + b$ and xy), "lose" if this equality does not hold.

If they can only use classical random variables, they cannot have a meaning strategy beyond deciding x and y randomly (and separately). Whatever the strategy is, if Charlie is choosing the question pair (x, y) in a random way, Alice and Bob's winning probability will be at most 75% since $xy = 0$ will happen 75% of the time (either $x = 0$ or $y = 0$) and their answers should match, while $xy = 1$ will happen 25% of the time ($x = y = 1$) then their answers should be different.

12.1. **The CHSH strategy.** The proposal by the CHSH paper uses the EPR pair state, and wins the game about 85% of the time. Bob will also use the following qubit states:

$$v = \cos\frac{\pi}{8}|0\rangle + \sin\frac{\pi}{8}|1\rangle, \qquad v' = \cos\frac{5\pi}{8}|0\rangle + \sin\frac{5\pi}{8}|1\rangle,$$
$$w = \cos\frac{-\pi}{8}|0\rangle + \sin\frac{-\pi}{8}|1\rangle, \qquad w' = \cos\frac{3\pi}{8}|0\rangle + \sin\frac{3\pi}{8}|1\rangle.$$

See Figure 12.1 for their arrangement in the usual Euclidean plane. Now, here is the strategy:

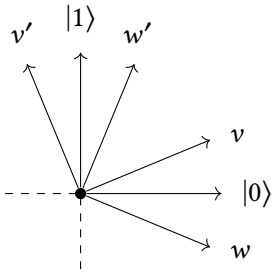


Figure 12.1. Bob's vectors and the computational basis vectors

[16] J. F. Clauser, M. A. Horne, A. Shimony, and R. A. Holt, "Proposed experiment to test local hidden-variable theories", *Phys. Rev. Lett.* **23** (15), 880–884 (1969).

[17] A. Aspect, P. Grangier, and G. Roger, "Experimental tests of realistic local theories via Bell's theorem", *Phys. Rev. Lett.* **47** (7), 460–463 (1981).

[18] A. Aspect, P. Grangier, and G. Roger, "Experimental realization of Einstein–Podolsky–Rosen–Bohm Gedankenexperiment: A new violation of Bell's inequalities", *Phys. Rev. Lett.* **49** (2), 91–94 (1982).

[19] B. Hensen, H. Bernien, A. E. Dréau, A. Reiserer, N. Kalb, M. S. Blok, J. Ruitenberg, R. F. L. Vermeulen, R. N. Schouten, C. Abellán, W. Amaya, V. Pruneri, M. W. Mitchell, M. Markham, D. J. Twitchen, D. Elkouss, S. Wehner, T. H. Taminiau, and R. Hanson, "Loophole-free Bell inequality violation using electron spins separated by 1.3 kilometres", *Nature* **526** (7575), 682–686 (2015).

(1) as preparation, Alice and Bob shares the EPR pair state $u = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \in H_A \otimes H_B$ for $H_A = \mathbb{C}^2 = H_B$;
(2) given the question x, Alice computes $a = 0, 1$ as follows:
   - if $x = 0$, then she measures her part of the state in the computational basis, i.e., with the PVM $E_0 = |0\rangle\langle 0|$ and $E_1 = |1\rangle\langle 1|$. The outcome will be a.
   - if $x = 1$, then she measures her part in the $(|+\rangle, |-\rangle)$-basis, i.e., with the PVM $E_+ = |+\rangle\langle +|$ and $E_- = |-\rangle\langle -|$. The case "+" corresponds to $a = 0$, while the case "−" corresponds to $a = 1$.
(3) given the question y, Bob computes $b = 0, 1$ as follows:
   - if $y = 0$, then he measures his part of the state in the (v, v')-basis, i.e., with the PVM $E_v = vv^\dagger$ and $E_{v'} = v'v'^\dagger$. The case "v" corresponds to $b = 0$, while the case "v'" corresponds to $b = 1$.
   - if $y = 1$, then he measures his part in the (w, w')-basis, i.e., with the PVM $E_w = ww^\dagger$ and $E_{w'} = w'w'^\dagger$. The case "w" corresponds to $b = 0$, while the case "w'" corresponds to $b = 1$.

How does this strategy work? We can look at what the state becomes after Alice's measurement, and estimate the probability for the desirable outcome for Bob. (If Bob does his measurement before Alice, the mathematics will be similar.)

Suppose she had received $x = 0$, so that she did computational basis measurement. Then the state will be either $|00\rangle$ or $|11\rangle$, depending of the outcome. So the states will look like the computational basis for Bob. In this case, he wants to match the outcome of Alice. We see that whichever y is, as we can see from Figure 12.1, his choice of basis will be $\frac{\pi}{8}$-rotation of Alice's basis $(|0\rangle, |1\rangle)$. This means that he will get a consistent answer with probability $\left(\cos\frac{\pi}{8}\right)^2$.

Next, suppose Alice had received $x = 1$. Since the EPR pair state w satisfies

$$u = \frac{1}{\sqrt{2}}(|++\rangle + |--\rangle),$$

the state after her measurement will be either $|++\rangle$ or $|--\rangle$ depending on the outcome, so the state looks like $|+\rangle$ or $|-\rangle$ for Bob. Now, the relation between these and Bob's measurement vector are in the configuration of Figure 12.2.

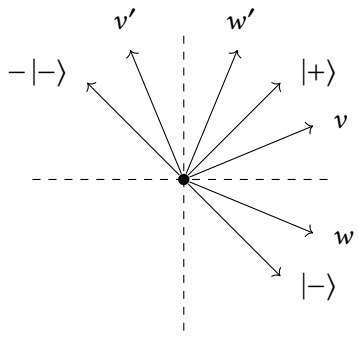


FIGURE 12.2. Bob's vectors and the $(|+\rangle, |-\rangle)$-basis vectors

If $y = 0$, then he wants to match Alice's measurement outcome. Since he will use (v, v'), the probability of achieving this is again $\left(\cos\frac{\pi}{8}\right)^2$. (Note that v' is close to $-|-\rangle$ instead of $v_- = |-\rangle$, and the inner product itself is a negative number. But the probability is given by $\mathbb{P}[b = 1] = |\langle v', v_-\rangle|^2$ and is close to 1.) If $y = 1$, then he does not want to match her measurement outcome. This time he is using w and w', and they are respectively closer to the desired vectors $|-\rangle$ and $|+\rangle$.

Summarizing the above discussion, in all cases Bob will get the winning outcome with probability $\left(\cos\frac{\pi}{8}\right)^2 = 0.8535\ldots$.

12.2. **Einstein-certified randomness.** The CHSH game can be used to *certify* that one has a machine that can generate a random sequence of bits.

What would be a *fake machine* in this context? It can be a deterministic program that holds an internal state s, and outputs sequence $a_1 a_2 \cdots$ of bits by updating the internal state to s_k after the step k, and compute the next output a_{k+1} using some function as $a_{k+1} = f(s_k)$. If an adversary gives you such a machine, the generated sequence might have a subtle bias that can be exploited if you are using it for encryption. This is called *random number generator attack*, which was exploited in real life (see Wikipedia article).

What you can do is to ask your vendor produce a machine with two parts that play the roles of Alice and Bob in the CHSH game. In other words, you switch on the machine, then separate the two parts in a secure way so that they cannot communicate, and give each parts inputs $x, y = 0, 1$. Then the two parts should answer with $a, b = 0, 1$ respectively. (Then you turn off the machine.)

Now you play the CHSH game with the provided machine many times and estimate the winning probability. If the machine is tampered during production so that it will try to give you a fake random sequence, it relies on a classical algorithm and cannot win the game more than 75% of the time. If the machine is winning more than that (and close to 85% of the time), then you can be confident that the generated sequence of a's is a truly random sequence.

12.3. **Computational note for Lecture 12.**

[1]:
```
import numpy as np
np.set_printoptions(suppress=True, legacy='1.25')
rng = np.random.default_rng()
```

The CHSH game: two cooperating players Alice and Bob plays the following game - Charlie (referee) gives them input bits x and y separately - they need to independently give back answer bits a and b - they win if $a + b = xy \bmod 2$ holds, lose otherwise

Alice and Bob prepares the EPR pair state before the game, and share the ($H_A = \mathbb{C}^2 = H_B$):

$$w = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \in H_A \otimes H_B$$

For Alice's part, if $x = 0$, she will measure her part in the computational basis to decide a. Otherwise, she will measure it in the $(|+\rangle, |-\rangle)$-basis, and reply with $a = 0$ in the "+" case, $a = 1$ in the "-" case.

For Bob's part, he is going to work with the orthonormal basis

$$v = \cos\frac{\pi}{8}|0\rangle + \sin\frac{\pi}{8}|1\rangle, \quad v' = \cos\frac{5\pi}{8}|0\rangle + \sin\frac{5\pi}{8}|1\rangle$$

if $y = 0$, and otherwise with the orthonormal basis

$$w = \cos\frac{-\pi}{8}|0\rangle + \sin\frac{-\pi}{8}|1\rangle, \quad w' = \cos\frac{3\pi}{8}|0\rangle + \sin\frac{3\pi}{8}|1\rangle.$$

[2]:
```
zeroket = np.array([1, 0])
oneket = np.array([0, 1])
plusket = np.array([1, 1]) / np.sqrt(2)
minusket = np.array([1, -1]) / np.sqrt(2)
EPR_stt = (np.kron(zeroket, zeroket) + np.kron(oneket, oneket)) / np.
  ↪sqrt(2)
EPR_stt
```

[2]:
```
array([0.70710678, 0.        , 0.        , 0.70710678])
```

[3]:
```
bob_v = np.array([np.cos(np.pi / 8), np.sin(np.pi / 8)])
bob_vp = np.array([np.cos(5 * np.pi / 8), np.sin(5 * np.pi / 8)])
bob_w = np.array([np.cos(-np.pi / 8), np.sin(-np.pi / 8)])
bob_wp = np.array([np.cos(3 * np.pi / 8), np.sin(3 * np.pi / 8)])
```

[4]:
```
def col_vec_inn_prod(v1, v2):
    """Hilbert space inner product of column vectors."""
    return np.conjugate(v1).dot(v2)


def measure_part_in_twoqubit(v, loc, basis_choice):
    """Measure one qubit of a two-qubit state in a specified basis.

    The function performs a projective measurement on one subsystem of a
    two-qubit state and returns both the measurement outcome and the
    post-measurement state.

    Parameters
    ----------
    v : numpy.ndarray
```

```
    One-dimensional complex array of length 4 representing a two-qubit
    state in the computational basis.
loc : {"A", "B"}
    Which qubit to measure.
    ``"A"`` measures the left (first) qubit,
    ``"B"`` measures the right (second) qubit.
basis_choice : int
    Choice of measurement basis for the measured qubit:
    ``0`` for the computational basis,
    ``1`` for the |+⟩ / |−⟩ basis;
    ``2`` for Bob's first basis;
    ``3`` for Bob's second basis.

Returns
-------
int
    Measurement outcome, ``0`` or ``1``, in the chosen basis.
numpy.ndarray
    One-dimensional complex array of length 4 representing the
    normalized post-measurement two-qubit state.
"""
# to be on the safe side, do not assume ||v|| = 1
# normalize the input vector
norm_v = np.sqrt(np.real(col_vec_inn_prod(v, v)))
v = v / norm_v
# decide basis
match basis_choice:
    case 0:
        basis_v0 = zeroket
        basis_v1 = oneket
    case 1:
        basis_v0 = plusket
        basis_v1 = minusket
    case 2:
        basis_v0 = bob_v
        basis_v1 = bob_vp
    case _:
        basis_v0 = bob_w
        basis_v1 = bob_wp
# decide PVM
sE0 = np.atleast_2d(basis_v0).T @ np.atleast_2d(basis_v0)
sE1 = np.atleast_2d(basis_v1).T @ np.atleast_2d(basis_v1)
if loc == "A":
    E0 = np.kron(sE0, np.eye(2))
    E1 = np.kron(sE1, np.eye(2))
else:
    E0 = np.kron(np.eye(2), sE0)
    E1 = np.kron(np.eye(2), sE1)
```

```
    # the case 0 happens with probability ⟨v, E0(v)⟩
    threshold = col_vec_inn_prod(v, E0 @ v)
    rand_num = rng.random()
    if rand_num < threshold:
        E0v = E0 @ v
        norm_E0v = np.sqrt(np.real(col_vec_inn_prod(E0v, E0v)))
        new_state = E0v / norm_E0v
        return (0, new_state)
    else:
        E1v = E1 @ v
        norm_E1v = np.sqrt(np.real(col_vec_inn_prod(E1v, E1v)))
        new_state = E1v / norm_E1v
        return (1, new_state)
```

[5]:
```
measure_part_in_twoqubit(EPR_stt, "A", 1)
```

[5]:
```
(1, array([ 0.5, -0.5, -0.5,  0.5]))
```

[6]:
```
measure_part_in_twoqubit(EPR_stt, "B", 2)
```

[6]:
```
(0, array([0.85355339, 0.35355339, 0.35355339, 0.14644661]))
```

[7]:
```
def play_CHSH():
    init_vec = EPR_stt
    # Charlie's question bits
    x, y = rng.integers(2), rng.integers(2)
    # Alice computes her answer first
    a, new_state = measure_part_in_twoqubit(init_vec, "A", x)
    # Bob computes his answer next
    b, _ = measure_part_in_twoqubit(new_state, "B", y + 2)
    # they win if a + b = x y mod 2
    if (a + b) % 2 == (x * y) % 2:
        return "win"
    else:
        return "lose"
```

[8]:
```
[play_CHSH() for _ in range(100)].count("win")
```

[8]:
```
83
```

In Section 8.1 we looked at Deutsch's algorithm to decide $f(0) = f(1)$ or not for a given Boolean function $f(x)$, with inputs $x = 0, 1$ and possible outputs $0, 1$. We can look at similar problems for Boolean functions $f(x_1 \cdots x_k)$ that takes k-bits as inputs (and still $0, 1$ as possible outputs). This increases the input possibility to 2^k cases, so classical deterministic algorithm can take exponentially longer time if we increase k. Can we do better with quantum algorithms? It turned out we can find ways to achieve "quantum parallelism" when f has particular forms.

13.1. **The Deutsch–Jozsa problem.** Reference: [NC00, Section 1.4.4; Aar18, Section 17.4]

This problem is proposed in a paper by Deutsch and Jozsa in 1992[20], as a straightforward generalization of Deutsch's setting. To be precise, we start with a partial knowledge about the function f:

> either f is constant so that for some i and for all $x = x_1 \cdots x_k$ we have $f(x) = i$, or else (the *balanced case*) $f(x) = 0$ for half of the cases among different choices of inputs, and $f(x) = 1$ for the rest.

We then want to determine if f is constant or balanced.

Classically, we need to choose different choices for the input x and compare the function values $f(x)$. If we see different values among them, we can conclude that f is balanced. In order to be 100% sure, we might have to take $2^{k-1} + 1$ samples of x if we are unlucky, which means that a deterministic classical program have to take exponentially long in terms of the parameter k. The quantum algorithm proposed by Deutsch and Jozsa settles this problem with just one "run", offering an exponential speedup. (To be more honest, in the balanced case, the expected number of samples to see the difference is 3. After seeing the same value k times, by concluding that f is constant the error probability is about 2^{-k}, so probabilistic algorithms without quantum mechanics can be much faster the above estimate.)

The algorithm works with the following $(k+1)$-qubit quantum gate U_f (called the *XOR query gate*):

$$U_f(|x_1 \cdots x_{k+1}\rangle) = |x_1 \cdots x_k x'_{k+1}\rangle\,, \quad x'_{k+1} = \mathrm{XOR}(x_{k+1}, f(x_1 \cdots x_k))$$

defined on the computational basis $|x_1 \cdots x_{k+1}\rangle$ for $x_i = 0, 1$. (Recall that XOR is given by $\mathrm{XOR}(0,1) = 1 = \mathrm{XOR}(1,0)$ and $\mathrm{XOR}(0,0) = 0 = \mathrm{XOR}(1,1)$.)

We will write $\{0,1\}^n$ for the set (collection) of n-bit sequences. Given two string of bits $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_n$ in this set, let us write $x \cdot y = \sum_{i=1}^n x_i y_i$.

Proposition 13.1. *For each y in $\{0,1\}^n$, the vector*

$$v_y = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (-1)^{x \cdot y} |x\rangle$$

is a unit vector of the n-qubit space $\mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2$.

For example, if $n = 2$ and $y = 00$, then we have the state $\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = |{+}{+}\rangle$.

Proposition 13.2. *Take $n = k + 1$ and $y = 1 \cdots 1 \in \{0,1\}^{k+1}$. If $f(x)$ is constant, then $U_f(v_y)$ is either v_y or $-v_y$. If $f(x)$ is balanced, then $U_f(v_y)$ is orthogonal to v_y. (It's enough to have $y_{k+1} = 1$ for this.)*

This means that we can measure the state $U_f(v_y)$ with respect to the PVM

$$E_c = v_y v_y^\dagger\,, \quad E_b = I - E_c$$

to decide if f is constant or not.

Proof for Proposition 13.1. We want to check $\langle v_y, v_y\rangle = 1$. Expanding the definitions, we get

$$\langle v_y, v_y\rangle = \frac{1}{2^n} \sum_{x,z \in \{0,1\}^n} (-1)^{x \cdot y} (-1)^{z \cdot y} \langle x|z\rangle\,.$$

[20] D. Deutsch and R. Jozsa, "Rapid solution of problems by quantum computation", *Proc. R. Soc. Lond. A* **439** (1907), 553–558 (1992).

Since the computational basis vectors are orthogonal, we have $\langle x|z\rangle = 0$ unless $x = z$, and in that case we have $\langle x|z\rangle = 1$. In this case we also have $(-1)^{x\cdot y}(-1)^{z\cdot y} = 1$. The number of choices for x is 2^n, so the above sum is equal to 2^n, and we get the result. □

Proof of Proposition 13.2. First suppose f is constant. If $f(x) = 0$, then $x'_k = \mathrm{XOR}(x_k, f(x))$ is equal to x_k, and U_f does nothing. This gives $U_f(v_y) = v_y$.

If $f(x) = 1$, then x'_k is the flip of x_k. This means U_f exchanges $|x_1 \cdots x_k 0\rangle$ with $|x_1 \cdots x_k 1\rangle$. In the definition of v_y, they appear with opposite signs because y is the sequence $1 \cdots 1$, so $(x_1 \cdots x_k 0) \cdot y$ and $(x_1 \cdots x_k 1) \cdot y$ differ by 1. This gives $U_f(v_y) = -v_y$.

Next suppose f is balanced. This means we can divide $\{0,1\}^k$ into two subsets I_0 and I_1, each containing 2^{k-1} sequences, such that $f(x) = j$ if x is in I_j. By the previous discussion, if $x_1 \cdots x_k$ is in I_0 then we have $U_f(|x_1 \cdots x_{k+1}\rangle) = |x_1 \cdots x_{k+1}\rangle$. If $x_1 \cdots x_k$ is in I_1, then U_f exchanges $|x_1 \cdots x_k 0\rangle$ with $|x_1 \cdots x_k 1\rangle$, and

$$U_f(w) = -w, \quad w = (-1)^{(x_1 \cdots x_k 0)\cdot y} |x_1 \cdots x_k 0\rangle + (-1)^{(x_1 \cdots x_k 1)\cdot y} |x_1 \cdots x_k 1\rangle .$$

If we expand $\langle v_y, U_f(v_y)\rangle$ in terms of the computational basis, besides the normalization factor $\frac{1}{2^{k+1}}$, there will be 2^{k-1} times of 1 and the same number of -1 in the sum. This implies $\langle v_y, U_f(v_y)\rangle = 0$, which is what we wanted. □

13.2. **The Bernstein–Vazirani problem.** Reference: [Aar18, Section 18]

Now, suppose that the function f is given as $f(x) = s\cdot x \bmod 2$ for some sequence $s = s_1 \cdots s_k \in \{0,1\}^k$. We think of s as a "secret" string, and f as something like the hash function. Classically, we determine s_1 by $s_1 = f(10\cdots 0)$, then $s_2 = f(010\cdots 0)$, so on, and we need k computations of $f(x)$ to know s. Bernstein and Vazirani[21] proposed a quantum algorithm to decide s with "one run" of quantum circuit.

This time we consider the following gate on k qubits (called the *phase query gate*):

$$V_f(|x_1 \cdots x_k\rangle) = (-1)^{f(x)} |x_1 \cdots x_k\rangle .$$

We will use the Hadamard gate H, which satisfied $H|0\rangle = |+\rangle$ and $H|1\rangle = |-\rangle$.

Proposition 13.3. *Consider v_y for $y = 0\cdots 0 \in \{0,1\}^k$ as in Proposition 13.1. We then have*

$$V_f(v_y) = (H \otimes \cdots \otimes H)\,|s\rangle .$$

Using

$$(H \otimes \cdots \otimes H)(H \otimes \cdots \otimes H) = (H^2 \otimes \cdots \otimes H^2) = I_2 \otimes \cdots \otimes I_2 = I_{2^k},$$

we see that $|s\rangle$ can be recovered as $(H \otimes \cdots \otimes H)V_f(v_y)$.

Proof of Proposition 13.3. Since $y = 0\cdots 0$, we have

$$v_y = \frac{1}{\sqrt{2^k}} \sum_{x\in\{0,1\}^k} |x\rangle$$

and $U_f(v_y)$ becomes

$$\frac{1}{\sqrt{2^k}} \sum_{x\in\{0,1\}^k} (-1)^{f(x)} |x\rangle .$$

Let us next expand

$$(H \otimes \cdots \otimes H)\,|s\rangle = (H|s_1\rangle) \otimes \cdots \otimes (H|s_k\rangle).$$

By $H|j\rangle = \frac{1}{\sqrt{2}}(|0\rangle + (-1)^j |1\rangle)$ for $j = 0, 1$, we have

$$(H|s_1\rangle) \otimes \cdots \otimes (H|s_k\rangle) = \frac{1}{\sqrt{2^k}}(|0\rangle + (-1)^{s_1} |1\rangle) \otimes \cdots \otimes (|0\rangle + (-1)^{s_k} |1\rangle).$$

[21] E. Bernstein and U. Vazirani, "Quantum complexity theory", *SIAM J. Comput.* **26** (5), 1411–1473 (1997).

If we expand $(|0\rangle+(-1)^{s_1}|1\rangle)\otimes\cdots\otimes(|0\rangle+(-1)^{s_k}|1\rangle)$, we get all computational basis vectors $|x\rangle$. Moreover, the coefficient will be $(-1)^{x\cdot s}$, since $(-1)^{s_j}$ will appear exactly when $x_j = 1$. Then we get

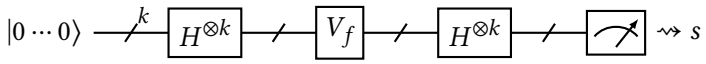
$$\frac{1}{\sqrt{2^k}} \sum_{x\in\{0,1\}^k} (-1)^{x\cdot s} |x\rangle ,$$

and we get the result by $f(x) = x \cdot s$. □

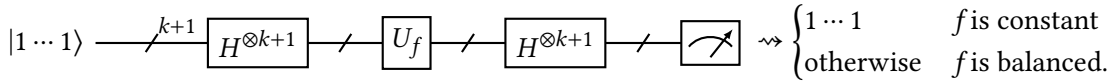
13.3. **Quantum circuit implementation.** If we can only prepare and measure in the computational basis, we have to add a bit more steps. As we saw in the last step of the proof of Proposition 13.3, we actually have

$$(H \otimes \cdots \otimes H)|y\rangle = v_y.$$

So the Bernstein–Vazirani algorithm can be represented by



with the "bundled wire" notation. Similarly, the Deutsch–Jozsa algorithm can be represented as



If we start the circuit with $X \otimes \cdots \otimes X$, we can use the initial state $|0 \cdots 0\rangle$.

13.4. **Computational note for Lecture 13.**

[1]:
```
import sympy as sy
sy.init_printing(use_unicode=True)
from sympy.physics.quantum import TensorProduct
from functools import reduce
import itertools as it
```

[2]:
```
zeroket = sy.Matrix([1, 0])
oneket = sy.Matrix([0, 1])
```

[3]:
```
def comp_basis(x):
    """Return the computational basis vector |x⟩.

    Parameters
    ----------
    x : iterable of int
        Iterable of bits (0 or 1) specifying the computational basis
        index.

    Returns
    -------
    sympy.Matrix
        One-dimensional SymPy column matrix representing the vector |x⟩.
    """
    vec_seq = (zeroket if xi == 0 else oneket for xi in x)
    return reduce(TensorProduct, vec_seq)
```

[4]:
```
# |01⟩ and |10⟩
comp_basis([0, 1]), comp_basis([1, 0])
```

[4]:
$$\left(\begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}\right)$$

[5]:
```
def dot_pr(x, y):
    """Return the dot product of (integer) sequences."""
    return sum(xi * yi for xi, yi in it.zip_longest(x, y))
```

[6]:
```
dot_pr([0, 1], [1, 1])
```

[6]:
1

[7]:
```
def v_y(y):
    """Construct the state vector from Proposition 13.1.

    Parameters
    ----------
    y : iterable of int
        Iterable of bits (0 or 1).

    Returns
```

```
    -------
    sympy.Matrix
        One-dimensional SymPy column matrix representing the normalized␣
↪sum of
        (-1)^(x · y) |x) over all bit strings x of the same length as y.
    """
    n = len(y)
    ind_list = it.product(range(2), repeat=n)
    res = sy.Matrix([0] * 2**n)
    for x in ind_list:
        res += comp_basis(x) * (1/sy.sqrt(2**n)) * (-1)**dot_pr(x, y)
    return res
```

[8]:
```
v_y([0, 0]), v_y([1, 1])
```

[8]:
$$\left(\begin{bmatrix}\frac{1}{2}\\ \frac{1}{2}\\ \frac{1}{2}\\ \frac{1}{2}\end{bmatrix}, \begin{bmatrix}\frac{1}{2}\\ -\frac{1}{2}\\ -\frac{1}{2}\\ \frac{1}{2}\end{bmatrix}\right)$$

[9]:
```
def tens_pow_H(k):
    """Return the tensor product of copies of the Hadamard matrix."""
    H = sy.Matrix([[1, 1],
                   [1, -1]]) * (1/sy.sqrt(2))
    return reduce(TensorProduct, (H,) * k)
```

[10]:
```
tens_pow_H(2)
```

[10]:
$$\begin{bmatrix}\frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2}\\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2}\\ \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2}\\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2}\end{bmatrix}$$

13.4.1. *Deutsch-Josza problem.*

[11]:
```
def U(k, f):
    """Return the U_f gate for the Deutsch-Josza problem."""
    ind_list = it.product(range(2), repeat=k+1)
    res = sy.zeros(2**(k+1))
    for in_x in ind_list:
        out_x = in_x[:-1] + (f(in_x[:-1]) ^ in_x[-1],)
        res += comp_basis(out_x) * comp_basis(in_x).T
    return res
```

[12]:
```
U(1, lambda x: 1)
```

[12]:
$$\begin{bmatrix}0 & 1 & 0 & 0\\ 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}$$

the vector v_y for $y = 1 \cdots 1$

[13]:
```
v_y([1] * 3)
```

[13]:

$$\begin{bmatrix} \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \end{bmatrix}$$

the constant case gives $\pm v_y$ for $y = 1 \cdots 1$

[14]:
```
U(2, lambda x: 1) * v_y([1] * 3)
```

[14]:

$$\begin{bmatrix} -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \end{bmatrix}$$

the balanced case gives a vector orthogonal to the above

[15]:
```
U(2, lambda x: x[0]) * v_y([1] * 3)
```

[15]:

$$\begin{bmatrix} \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ -\frac{\sqrt{2}}{4} \\ \frac{\sqrt{2}}{4} \end{bmatrix}$$

[16]:
```
# computational basis presentation
Hpow = tens_pow_H(3)
Hpow * U(2, lambda x: 0) * Hpow * comp_basis([1] * 3), \
    Hpow * U(2, lambda x: 1) * Hpow * comp_basis([1] * 3), \
    Hpow * U(2, lambda x: x[0]) * Hpow * comp_basis([1] * 3), \
    Hpow * U(2, lambda x: (x[0] + x[1]) % 2) * Hpow * comp_basis([1] * 3)
```

[16]:

$$\left(\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\1\end{bmatrix}, \begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\-1\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\\0\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\\0\\0\\0\\0\end{bmatrix}\right)$$

13.4.2. *Bernstein-Vazirani problem.*

[17]:
```
def V(k, f):
    """Return V_f gate for the Bernstein-Vazirani problem."""
    ind_list = it.product(range(2), repeat=k)
    res = sy.zeros(2**k)
    for x in ind_list:
        res += comp_basis(x) * comp_basis(x).T * (-1)**f(x)
    return res
```

[18]:
```
V(2, lambda x: dot_pr(x, [0, 1])), V(2, lambda x: dot_pr(x, [1, 1]))
```

[18]:
$$\left(\begin{bmatrix}1&0&0&0\\0&-1&0&0\\0&0&1&0\\0&0&0&-1\end{bmatrix}, \begin{bmatrix}1&0&0&0\\0&-1&0&0\\0&0&-1&0\\0&0&0&1\end{bmatrix}\right)$$

[19]:
```
# check Proposition 13.3
s = [0, 1]
V(2, lambda x: dot_pr(x, s)) * v_y([0, 0]), v_y(s)
```

[19]:
$$\left(\begin{bmatrix}\frac{1}{2}\\-\frac{1}{2}\\\frac{1}{2}\\-\frac{1}{2}\end{bmatrix}, \begin{bmatrix}\frac{1}{2}\\-\frac{1}{2}\\\frac{1}{2}\\-\frac{1}{2}\end{bmatrix}\right)$$

[20]:
```
s = [0, 1]
tens_pow_H(2) * V(2, lambda x: dot_pr(x, s)) * v_y([0, 0]), comp_basis(s)
```

[20]:
$$\left(\begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}\right)$$

14. Query complexity

The two problems in the last lecture showed some form of "speedup" by quantum algorithms over classical algorithms. How do we make this more precise? One way is to think about the *query complexity* of algorithms.

In broad terms, we want to know some property of a Boolean function f (constant vs. balanced, or which s defines it) and classical algorithms decides it by asking for the value of $f(x)$ for different inputs x (what is $f(00)$?, what is $f(01)$?, etc.). As we discussed, a deterministic classical algorithm to decide the Deutsch–Jozsa problem for a function with k-bit input would take $2^{k-1}+1$ steps at worst (and probabilistic algorithms would take about 3 steps). Similarly, a standard classical algorithm for the Bernstein–Vazirani problem would take k steps.

The quantum algorithms had quantum gates U_f (XOR query) and V_f (phase query), that we want to treat as analogue of the above queries in classical computation. Since those algorithms used only one copies of these, we can say that they have quantum complexity 1 regardless of the length k for the input bits.

Remark 14.1. We can take the $(k+1)$-qubit analogue of phase query gate defined by

$$U'_f |x_1 \cdots x_k y\rangle = (-1)^{f(x)y} |x_1 \cdots x_k y\rangle$$

for $x_i, y = 0, 1$, so that $V_f|x\rangle$ corresponds to $U'_f|x1\rangle$. This modified phase query gate satisfies $U_f = (I_{2^k} \otimes H)U'_f(I_{2^k} \otimes H)$. In particular, the quantum query complexities in terms of the XOR query and the phase query are essentially the same.

There is a more down-to-earth way to think about complexity called the *circuit complexity*, where we count the number of basic operations that make up the algorithm. In the classical computation, that would be counting the number of logical gates like $\mathrm{AND}(x, y) = xy$, $\mathrm{NOT}(x) = 1 - x$, etc., in a "circuit". Roughly speaking, this is like counting the number of CPU instructions when we run a program.

For the quantum computation, we would be counting the number of basic quantum gates like rotation of single qubit (Example 2.9) or the Hadamard gate H, and their controlled versions that make up the quantum circuit implementing the algorithm. However, estimating this complexity is difficult in practice (but we will do it for Shor's algorithm later!).

14.1. Simon's problem.

Reference: [Aar18, Section 18]

Simon's problem and his quantum algorithm[22] gave a first example of exponential speedup in terms of query complexity even against probabilistic classical algorithms.

This time, the "secret" string $s = s_1 \cdots s_k \in \{0,1\}^k$ is encoded in the following way: we are given a function $f\colon \{0,1\}^k \to \{0,1\}^k$, and we have $f(x) = f(y)$ if and only if either $x = y$ or y is the bitwise-XOR of x with s, i.e., $y_i = \mathrm{XOR}(x_i, s_i)$. If s is not the constant sequence $0 \cdots 0$ (which we will assume), then for any x there is exactly one more y such that $f(x) = f(y)$ holds.

Classically, in order to find s, we have to find a (distinct) pair x, y in the k-bit space $\{0,1\}^k$ satisfying $f(x) = f(y)$, and compute s by $s_i = \mathrm{XOR}(x_i, y_i)$. A worst-case scenario for a deterministic would take $2^{k-1}+1$ steps like the Deutsch–Jozsa problem. We can find a probabilistic algorithm based on the birthday paradox: even though there are 365 days in one year, it's more likely that there is a pair with the same birthday in your class (Wikipedia article). This gives about $\sqrt{2^k}$ steps of random choice in $\{0,1\}^k$ to find a pair x, y as above with high probability. In other words, the query complexity would be $\sqrt{2^k}$.

Simon's quantum algorithm finds (with high probability) enough information to get s after about k runs of a quantum circuit that has copies of the Hadamard gate, one XOR query gate for f and measurements of qubits. This means that the query complexity of the algorithm is k, hence an exponential speedup over the classical one.

Given k-bit strings $x, y \in \{0,1\}^k$, let us write their bitwise XOR as $x \oplus y$. (This is the sum of vectors in $\mathbb{F}_2^k$, the k-dimensional vector space over the field of two elements $\mathbb{F}_2$.) Then the XOR query gate for

[22] D. R. Simon, "On the power of quantum computation", *SIAM J. Comput.* **26** (5), 1474–1483 (1997).

a function $f\colon \{0,1\}^k \to \{0,1\}^k$ is the $2k$-qubit gate U_f defined by

$$U_f(|xy\rangle) = |xz\rangle, z = f(x) \oplus y \quad (x, y \in \{0,1\}^k).$$

Proposition 14.2. *Let v be the $2k$-qubit state given by*

$$v = (H^{\otimes k} \otimes I_{2^k}) U_f (H^{\otimes k} \otimes I_{2^k})(|0 \cdots 0\rangle).$$

When we measure the left k-qubits of v in the computational basis, the result z always satisfies $\sum_i s_i z_i = 0 \bmod 2$.

If we do the above measurement k times, we would get strings $z^1, \dots, z^k$ satisfying

$$z_1^1 s_1 + \cdots + z_k^1 s_k = 0, \quad z_1^2 s_1 + \cdots + z_k^2 s_k = 0, \quad \dots \quad z_1^k s_1 + \cdots + z_k^k s_k = 0 \mod 2.$$

Formally speaking, this is a system of linear equations whose coefficient system is the *field of 2 elements*, namely we consider 0 and 1 as the only possible "numbers". The collection $\{0,1\}^k$ can be considered as the k-dimensional vector space over this number system, and the above system imposes k constraints. Unless there is still too much linear relations among the z^i making it an under-determined system (that's the unlucky case), we can determine s as the unique nonzero solution.

Proof of Proposition 14.2. We chase what happens to the state vector at each step of transform.

Step 1: $(H^{\otimes k} \otimes I_{2^k})(|0 \cdots 0\rangle)$ is given by the vector

$$\frac{1}{\sqrt{2^k}} \sum_{x \in \{0,1\}^k} |x0 \cdots 0\rangle.$$

This follows from $H|0\rangle = |+\rangle$, and

$$|\underset{k \text{ times}}{+ \cdots +}\rangle = \frac{1}{\sqrt{2^k}} \sum_{x \in \{0,1\}^k} |x\rangle$$

which we get from $|{+}{+}\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$, etc.

Step 2: $U_f(H^{\otimes k} \otimes I_{2^k})(|0 \cdots 0\rangle)$ is given by the vector

$$\frac{1}{\sqrt{2^k}} \sum_{w \in \{0,1\}^k} (|xw\rangle + |yw\rangle),$$

where x and y are the two solutions to $f(x) = w = f(y)$. We discard the w without such solutions. This follows from $f(x) \oplus (0 \cdots 0) = f(x)$, and by grouping the terms which have the same "second half". Since half of $\{0,1\}^k$ is hit by f, there are overall $2^{k-1} \times 2 = 2^k$ terms.

Step 3: $(H^{\otimes k} \otimes I_{2^k}) U_f (H^{\otimes k} \otimes I_{2^k})(|0 \cdots 0\rangle)$ is given by the vector

$$\frac{1}{2^k} \sum_{z,w \in \{0,1\}^k} ((-1)^{x \cdot z} + (-1)^{y \cdot z}) |zw\rangle,$$

where x and y are determined by w as above. Note that we have

$$H^{\otimes k} H^{\otimes k} = (H \cdot H) \otimes \cdots \otimes (H \cdot H) = I_2 \otimes \cdots \otimes I_2 = I_{2^k}.$$

Then checking the above claim can be reduced to

$$\frac{1}{2^k}(-1)^{x \cdot z} \sum_{z,w \in \{0,1\}^k} (H^{\otimes k} \otimes I_2^{\otimes k}) |zw\rangle = \frac{1}{\sqrt{2^k}} \sum_{w \in \{0,1\}^k} |xw\rangle.$$

The second half is hit by $I_2^{\otimes k}$, so it does not play any role. We then need to check

$$\frac{1}{\sqrt{2^k}} \sum_{z \in \{0,1\}^k} (-1)^{x \cdot z} H^{\otimes k} |z\rangle = |x\rangle$$

for $x = x_1 \cdots x_k$. When $k = 1$, we have two cases to check:

$$\frac{1}{\sqrt{2}}(H|0\rangle + H|1\rangle) = |0\rangle$$

for $x = 0$, and

$$\frac{1}{\sqrt{2}}(H|0\rangle - H|1\rangle) = |1\rangle$$

for $x = 1$. Using $H|0\rangle = |+\rangle$ and $H|1\rangle = |-\rangle$, we indeed have these equalities. The case of general k is a tensor product of these computations.

For z to be a possible value of the measurement, we must have $(-1)^{x\cdot z} + (-1)^{y\cdot z}) \neq 0$ for x and y determined by some w. But any such pair satisfies $x = y \oplus s$ for the secret string s. Finally, observe that $(-1)^{x\cdot z} + (-1)^{y\cdot z} \neq 0$ happens if and only if $x \cdot z = y \cdot z \bmod 2$. Combining the above two, we get $s \cdot z = x \cdot z - y \cdot z = 0 \bmod 2$. □

14.2. **Computational note for Lecture 14.** Simon's problem: we work with Boolean functions $f\colon \{0,1\}^k \to \{0,1\}^k$ and a secret string $s \in \{0,1\}^k$ that contains 1. We know that $f(x) = f(y)$ if and only if either $x = y$ or y is the bitwise-XOR, and want to work out s from f.

[1]:
```
import numpy as np
np.set_printoptions(suppress=True, legacy='1.25')
rng = np.random.default_rng()
import itertools as it
from functools import reduce
```

[2]:
```
# set secret string
k = 4  # 5 is the sensible maximum of naive implementation?
s = tuple([rng.integers(2) for _ in range(k)])
if 1 not in s:
    # try again
    s = tuple([rng.integers(2) for _ in range(k)])
    if 1 not in s:
        print("we could not generate a secret string.")
```

[3]:
```
# function satisfying the above condition
def f_from_s(x):
    """Return x or x XOR s according to a fixed selection rule.

    Parameters
    ----------
    x : tuple of int
        Tuple of bits (0 or 1) of length ``k`` representing the input.

    Returns
    -------
    tuple of int
        A tuple equal to either ``x`` (if ``s`` is the sequence of 0's)
        or ``x XOR s``.
    """
    ind_s_1 = [i for i in range(k) if s[i] == 1][0]

    if x[ind_s_1] == 0:
        return x
    y_l = [x[i] ^ s[i] for i in range(k)]
    return tuple(y_l)
```

[4]:
```
s, f_from_s(s), f_from_s((0,) * k), f_from_s((1,) * k)
```

[4]:
```
((0, 0, 1, 0), (0, 0, 0, 0), (0, 0, 0, 0), (1, 1, 0, 1))
```

14.2.1. *classical algorithm.* Th classical "birthday attack" algorithm works by picking input strings $x^1, x^2, \ldots$ and querying $f(x^1), f(x^2), \ldots$, recording the results. It stops when it sees a pair x^i, x^j such that $f(x^i) = f(x^j)$. Then the secret is $s = x^i \oplus x^j$, the bitwise XOR.

[5]:
```
res_rec = {}  # dictionary of f(x) as key and x as value
ind_list = it.product(range(2), repeat=k)
```

```
for x in ind_list:
    f_x = f_from_s(x)
    if f_x in res_rec.keys():
        prev_x = res_rec[f_x]
        found_secret = tuple(x[i] ^ prev_x[i] for i in range(k))
        print(f"found secret {found_secret} after {1 + len(res_rec)}␣
 ↪queries")
        break
    # if f(x) is not found, record the results and continue with next x
    res_rec[f_x] = x
```

```
found secret (0, 0, 1, 0) after 3 queries
```

the expected run time is $\sqrt{2^k}$, where k is the length of secret

[6]:
```
np.sqrt(2**k)
```

[6]:
```
4.0
```

14.2.2. *Simon's quantum algorithm.* the quantum algorithm looks at the state

$$v = (H^{\otimes k} \otimes I_{2^k}) U_f (H^{\otimes k} \otimes I_{2^k}) \, |0 \cdots 0\rangle$$

in the $2k$-qubit space $\mathbb{C}^{2^{2k}} = \mathbb{C}^{2^k} \otimes \mathbb{C}^{2^k}$.

[7]:
```
zeroket = np.array([1, 0])
oneket = np.array([0, 1])
```

[8]:
```
def comp_basis(x):
    # computational basis vector
    # input
    #   x: array of int (0 or 1)
    # output
    #   numpy.array, 1-dimensional array representing |x⟩
    vec_seq = (zeroket if xi == 0 else oneket for xi in x)
    return reduce(np.kron, vec_seq)
```

[9]:
```
def tens_pow_H(k):
    # tensor product of copies of the Hadamard matrix
    H = np.array([[1, 1],
                  [1, -1]]) * (1/np.sqrt(2))
    return reduce(np.kron, [H] * k)
```

[10]:
```
def termwise_XOR(x, y):
    # termwise XOR of tuples
    return tuple([xi ^ yi for xi, yi in it.zip_longest(x, y)])


def U(k, f):
    # U_f gate for the Simon algorithm
    ind_list = it.product(range(2), repeat=2*k)
    res = np.zeros((2**(2*k), 2**(2*k)), dtype=np.int8)
```

```
    for in_x in ind_list:
        out_x = in_x[0:k] + termwise_XOR(f(in_x[0:k]), in_x[k:2*k])
        in_as_row = np.atleast_2d(comp_basis(in_x))
        out_as_column = np.atleast_2d(comp_basis(out_x)).T
        res += out_as_column @ in_as_row
    return res
```

[11]:
```
U(k, f_from_s).shape
```

[11]:
```
(256, 256)
```

[12]:
```
# first 20 x 20 portion of U_f
print(U(k, f_from_s)[0:20, 0:20])
```

```
[[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]]
```

[13]:
```
# bottom right 20x20 portion
print(U(k, f_from_s)[-20:, -20:])
```

```
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 0 0 0 0 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 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]]
```

[14]:
```
H_pow_ampl = np.kron(tens_pow_H(k), np.identity(2**k))
simon_gate = H_pow_ampl @ U(k, f_from_s) @ H_pow_ampl
```

[15]:
```
v = simon_gate @ comp_basis([0] * (2 * k))
```

[16]:
```
def proj(x):
    """Return the projection for the computational basis."""
    basis_vec = np.atleast_2d(comp_basis(x))
    return basis_vec.T @ basis_vec


def meas_prob(x):
    """Return the probability of observing x in the first half of v."""
    return np.dot(v, np.kron(proj(x), np.identity(2**k)) @ v)
```

[17]:
```
intervals = {}
t = 0.0
for x in it.product(range(2), repeat=k):
    prob_for_x = meas_prob(x)
    intervals[x] = (t, t + prob_for_x)
    t += prob_for_x
    print(f"{x} with probability {prob_for_x}")
```

```
(0, 0, 0, 0) with probability 0.12499999999999983
(0, 0, 0, 1) with probability 0.12499999999999983
(0, 0, 1, 0) with probability 3.8457049370327185e-64
(0, 0, 1, 1) with probability 3.8457049370327185e-64
(0, 1, 0, 0) with probability 0.12499999999999983
(0, 1, 0, 1) with probability 0.12499999999999983
(0, 1, 1, 0) with probability 3.8457049370327185e-64
(0, 1, 1, 1) with probability 3.8457049370327185e-64
(1, 0, 0, 0) with probability 0.12499999999999983
(1, 0, 0, 1) with probability 0.12499999999999983
(1, 0, 1, 0) with probability 3.8457049370327185e-64
(1, 0, 1, 1) with probability 3.8457049370327185e-64
(1, 1, 0, 0) with probability 0.12499999999999983
(1, 1, 0, 1) with probability 0.12499999999999983
(1, 1, 1, 0) with probability 3.8457049370327185e-64
(1, 1, 1, 1) with probability 3.8457049370327185e-64
```

[18]:
```
x_samples = []
for _ in range(k):  # when k is small, it's better to repeat a bit more
    rand_num = rng.random()
    for x in it.product(range(2), repeat=k):
        if intervals[x][0] < rand_num and rand_num < intervals[x][1]:
            x_samples.append(x)
            break
```

[19]:
```
x_samples
```

[19]:
```
[(1, 1, 0, 0), (1, 0, 0, 1), (1, 0, 0, 1), (1, 0, 0, 1)]
```

[20]:
```
# find s by solving x•s = 0 mod 2 for the x in the above samples
def inn_prod(x, s):
    return sum(xi * si for xi, si in it.zip_longest(x, s))


def predicate(s):
    if s == (0,) * k:
        return False
    inn_prod_mod_2_list = [inn_prod(x, s) % 2 for x in x_samples]
    if 1 in inn_prod_mod_2_list:
        return False
    return True


for s_cand in it.product(range(2), repeat=k):
    if predicate(s_cand):
        print(f"{s_cand} is a candidate for the secret")
```

```
(0, 0, 1, 0) is a candidate for the secret
(1, 1, 0, 1) is a candidate for the secret
(1, 1, 1, 1) is a candidate for the secret
```

Variation: every time we get a new candidate for $x^i = x^i_1 \cdots x^i_k$, we set the syste of linear equations

$$x^i_1 s_1 + x^i_2 s_2 + \cdots + x^i_k s_k = 0 \bmod 2 \quad (i = 1, 2, \ldots)$$

and continue until there is only one non-zero solution for s.

[21]:
```
import sympy as sy
# prepare variables s_1, ..., s_k
s_var = [sy.Symbol(f"s_{j+1}") for j in range(k)]

x_samples = []
l_poly = []
for trial_cnt in range(2 * k):
    rand_num = rng.random()
    for x in it.product(range(2), repeat=k):
        if intervals[x][0] < rand_num and rand_num < intervals[x][1]:
            new_x = x
            break
```

```
    # ignore x^i = (0,..,0), otherwise set linear polynomial with
    # coefficients x^i_j
    if new_x == (0,) * k:
        continue
    else:
        f2 = sy.FF(2)
        i_th_lin_poly = sy.Poly(sum(f2(new_x[j]) * s_var[j] for j in␣
 ↪range(k)))
        l_poly.append(i_th_lin_poly)
    # find solutions
    ans_set = sy.solve(l_poly)
    print(ans_set)
    if len(ans_set) == k - 1:
        print(f"got a unique solution after {trial_cnt + 1} trials")
        break
```

```
{s_4: 0}
{s_1: 0, s_4: 0}
{s_1: 0, s_4: 0}
{s_1: 0, s_4: 0}
{s_1: 0, s_2: 0, s_4: 0}
got a unique solution after 6 trials
```

[22]:
```
# find free variables
free_vars = set()
for key in ans_set.keys():
    free_vars |= ans_set[key].free_symbols
# get the nonzero solution as a string
if len(free_vars) == 0:
    subs_dict = {}
else:
    subs_dict = {list(free_vars)[0]: 1}

ans_seq = []
for the_var in s_var:
    if the_var in ans_set.keys():
        ans_seq.append(ans_set[the_var].subs(subs_dict) % 2)
    else:
        ans_seq.append(1)

ans_seq
```

[22]:
```
[0, 0, 1, 0]
```

optimization 1: sparse matrices. try bigger k with sparse matrix algorithms from `scipy` library

[23]:
```
# set secret string
k = 7
s = tuple([rng.integers(2) for _ in range(k)])
if 1 not in s:
```

```
    # try again
    s = tuple([rng.integers(2) for _ in range(k)])
    if 1 not in s:
        print("we could not generate a secret string.")
```

[24]:
```
s
```

[24]:
```
(1, 1, 0, 0, 0, 0, 0)
```

[25]:
```
# sparse matrix for large k
import scipy.sparse as spa


def ind_from_seq(x):
    """Convert the binary sequence of length k to matrix index 0...2**k-1.
    """
    return sum(2**(len(x)-j-1) * x[j] for j in range(len(x)))


def comp_basis_sp(x):
    """Return 2-d column vector representing the x-th basis."""
    mat_size = 2**len(x)
    res = spa.dok_matrix((mat_size, 1))
    res[ind_from_seq(x), 0] = 1
    return res.tocsr()


def U_sp(k, f):
    """Return U_f gate for Simon's algorithm."""
    mat_size = 2**(2*k)
    res = spa.dok_matrix((mat_size, mat_size))
    ind_list = it.product(range(2), repeat=2*k)
    for in_x in ind_list:
        out_x = in_x[0:k] + termwise_XOR(f(in_x[0:k]), in_x[k:2*k])
        res[ind_from_seq(out_x), ind_from_seq(in_x)] = 1
    return res.tocsr()
```

[26]:
```
H_pow_ampl_sp = spa.kron(tens_pow_H(k), np.identity(2**k))
```

[27]:
```
simon_gate_sp = H_pow_ampl_sp @ U_sp(k, f_from_s) @ H_pow_ampl_sp
```

[28]:
```
v = simon_gate_sp.dot(comp_basis_sp([0] * (2 * k)))
```

[29]:
```
# sparse version of helper commands
def proj(x):
    """Return projection for the computational basis."""
    basis_vec = comp_basis_sp(x)
    return basis_vec @ basis_vec.T
```

```
def meas_prob(x):
    """Return the probability of observing x in the first half of v."""
    return (v.T @ spa.kron(proj(x), np.identity(2**k)) @ v)[0, 0]
```

[30]:
```
intervals = {}
t = 0.0
for x in it.product(range(2), repeat=k):
    prob_for_x = meas_prob(x)
    intervals[x] = (t, t + prob_for_x)
    t += prob_for_x
```

[31]:
```
x_samples = []
for _ in range(k):  # when k is small, it's better to repeat a bit more
    rand_num = rng.random()
    for x in it.product(range(2), repeat=k):
        if intervals[x][0] < rand_num and rand_num < intervals[x][1]:
            x_samples.append(x)
            break
```

[32]:
```
x_samples
```

[32]:
```
[(0, 0, 1, 1, 1, 1, 0),
 (1, 1, 0, 1, 1, 0, 0),
 (1, 1, 1, 0, 0, 1, 1),
 (0, 0, 1, 1, 0, 0, 1),
 (1, 1, 0, 0, 1, 0, 0),
 (0, 0, 1, 0, 0, 1, 1),
 (0, 0, 1, 0, 0, 1, 1)]
```

[33]:
```
for s_cand in it.product(range(2), repeat=k):
    if predicate(s_cand):
        print(f"{s_cand} is a candidate for the secret")
```

```
(1, 1, 0, 0, 0, 0, 0) is a candidate for the secret
```

optimization 2: fast Walsh-Hadamard transform. we avoid creating matrices and do in-place manipulation as much as possible

[34]:
```
# set secret string
k = 10
s = tuple([rng.integers(2) for _ in range(k)])
if 1 not in s:
    # try again
    s = tuple([rng.integers(2) for _ in range(k)])
    if 1 not in s:
        print("we could not generate a secret string.")

s
```

[34]:
```
(1, 0, 0, 0, 0, 0, 0, 0, 1, 1)
```

[35]:
```python
def fwht(a):
    """Perform the in-place Fast Walsh–Hadamard Transform of array a."""
    h = 1
    n = len(a)
    while h < n:
        # perform FWHT
        for i in range(0, n, h * 2):
            for j in range(i, i + h):
                x = a[j]
                y = a[j + h]
                a[j] = x + y
                a[j + h] = x - y
        # normalize and increment h
        a *= 1/np.sqrt(2)
        h *= 2


def H_pow_ampl_fwht(x, k):
    """Apply (H^{⊗k} ⊗ I_{2^k}) to x using FWHT.

    Parameters
    ----------
    x : numpy.ndarray
        One-dimensional array of length ``2**(2*k)`` representing a state
        on $2k$ qubits.
    k : int
        Number of qubits on which H^{⊗k} acts (the first subsystem).

    Returns
    -------
    numpy.ndarray
        One-dimensional array representing (H^{⊗k} ⊗ I_{2^k}) x.
    """
    x = x.reshape(2**k, 2**k)  # shape (2^k, 2^k)

    # Apply FWHT along axis 0
    for col in range(2**k):
        fwht(x[:, col])

    return x.reshape(2**(2*k))
```

[36]:
```python
def apply_index_permutation(x, perm):
    """Apply a basis index permutation to a state vector.

    The permutation ``perm`` is regarded as the action on computational
    basis indices, and the function returns the permuted state
    y = P x, where P|i⟩ = |perm[i]⟩.

    Parameters
```

```
    ----------
    x : numpy.ndarray
        One-dimensional array of length ``N``, representing a vector in an
        ``N``-dimensional vector space.
    perm : array_like of int
        One-dimensional array-like of length ``N`` representing a␣
 ↪permutation
        of ``range(N)``.

    Returns
    -------
    numpy.ndarray
        One-dimensional array ``y`` of the same shape and dtype as ``x``,
        satisfying ``y[perm[i]] = x[i]`` for all ``i``.
    """
    perm = np.asarray(perm)
    y = np.empty_like(x)
    y[perm] = x  # y[perm[i]] = x[i]
    return y
```

[37]:
```
perm_rep_Uf = [0] * 2**(2*k)
ind_list = it.product(range(2), repeat=2*k)
for in_x in ind_list:
    out_x = in_x[0:k] + termwise_XOR(f_from_s(in_x[0:k]), in_x[k:2*k])
    perm_rep_Uf[ind_from_seq(in_x)] = ind_from_seq(out_x)
```

[38]:
```
def comp_basis_big(x):
    """Return 2-d column vector representing the x-th basis."""
    res = np.array([0] * 2**len(x), dtype='float64')
    res[ind_from_seq(x)] = 1
    return res
```

[39]:
```
start_vec = comp_basis_big([0] * (2 * k))
first_H_eff_vec = H_pow_ampl_fwht(start_vec, k)
permed_vec = apply_index_permutation(first_H_eff_vec, perm_rep_Uf)
second_H_eff_vec = H_pow_ampl_fwht(permed_vec, k)
```

[40]:
```
# adaptation of helper commands
def pair_first_half(v, w):
    """Contract a 2k-qubit state v with a k-qubit state w over the first k
    qubits. (Components are assumed to be real.)

    The input state ``v`` is interpreted as a bipartite state on
    H_A ⊗ H_B, where each subsystem has k qubits. The function
    contracts against a state ``w`` on H_A to produce a state on H_B.

    Parameters
    ----------
    v : numpy.ndarray
```

```
        One-dimensional array of length ``2**(2*k)`` representing a state
        on $2k$ qubits.
    w : numpy.ndarray
        One-dimensional array of length ``2**k`` representing a state on
        the first $k$ qubits.

    Returns
    -------
    numpy.ndarray
        One-dimensional array of length ``2**k`` representing the␣
 ↪resulting
        state on the last ``k`` qubits.
    """
    # Reshape v as a bipartite state: (first k qubits, last k qubits)
    v_reshaped = v.reshape(2**k, 2**k)

    # Contract over the first subsystem: u = (<w| ⊗ I) |v>
    # This is u = w.conj().T @ v_reshaped
    return w.T @ v_reshaped


def meas_prob(x, v):
    """Return the probability of observing x in the first half of v."""
    w = comp_basis_big(x)
    u = pair_first_half(v, w)
    return np.dot(u, u)
```

[41]:
```
intervals = {}
t = 0.0
for x in it.product(range(2), repeat=k):
    prob_for_x = meas_prob(x, second_H_eff_vec)
    intervals[x] = (t, t + prob_for_x)
    t += prob_for_x
```

perform the measurement to get strings $x^1, x^2, ...$, then solve the linear equations to get the secret string

[42]:
```
import sympy as sy

f2 = sy.FF(2)

# prepare variables s_1, ..., s_k
s_var = [sy.Symbol(f"s_{j+1}") for j in range(k)]

x_samples = []
l_poly = []
coeff_list = []
for trial_cnt in range(2 * k):
    rand_num = rng.random()
    for x in it.product(range(2), repeat=k):
```

```
        if intervals[x][0] < rand_num and rand_num < intervals[x][1]:
            new_x = x
            break
    # ignore x^i = (0,..,0), otherwise set linear polynomial with
    # coefficients x^i_j
    if new_x == (0,) * k:
        continue
    else:
        i_th_lin_poly = sy.Poly(sum(new_x[j] * s_var[j] for j in␣
 ↪range(k)), domain=f2)
        l_poly.append(i_th_lin_poly)
        coeff_list.append(list(new_x))
    # find solutions
    ans_set = sy.solve(l_poly, s_var)
    print(ans_set)
    if len(ans_set) == k - 1:
        print(f"got a unique solution after {trial_cnt + 1} trials")
        break
```

```
{s_2: -s_7 - s_8}
{s_1: -s_10 - s_4 + s_7, s_2: -s_7 - s_8}
{s_1: -s_10 - s_4 + s_7, s_2: -s_7 - s_8, s_3: -s_10 - s_5 - s_6 + s_7 +␣
 ↪s_8 -
s_9}
{s_1: s_10 + s_5 + s_6, s_2: -s_7 - s_8, s_3: -s_10 - s_5 - s_6 + s_7 + s_8␣
 ↪-
s_9, s_4: -2*s_10 - s_5 - s_6 + s_7}
{s_1: -s_10 - s_6 + s_7, s_2: -s_7 - s_8, s_3: s_10 + s_6 + s_8 - s_9, s_4:␣
 ↪s_6,
s_5: -2*s_10 - 2*s_6 + s_7}
{s_1: s_10 - s_6, s_2: -2*s_10 - s_8, s_3: s_10 + s_6 + s_8 - s_9, s_4: s_6,
s_5: -2*s_6, s_7: 2*s_10}
{s_1: 4*s_10 + 2*s_8 - s_9, s_2: -2*s_10 - s_8, s_3: -2*s_10 - s_8, s_4: -
3*s_10
- 2*s_8 + s_9, s_5: 6*s_10 + 4*s_8 - 2*s_9, s_6: -3*s_10 - 2*s_8 + s_9, s_7:
2*s_10}
{s_1: -6*s_10 + s_9, s_2: 3*s_10 - s_9, s_3: 3*s_10 - s_9, s_4: 7*s_10 -␣
 ↪s_9,
s_5: -14*s_10 + 2*s_9, s_6: 7*s_10 - s_9, s_7: 2*s_10, s_8: -5*s_10 + s_9}
{s_1: -3*s_10, s_2: 0, s_3: 0, s_4: 4*s_10, s_5: -8*s_10, s_6: 4*s_10, s_7:
2*s_10, s_8: -2*s_10, s_9: 3*s_10}
got a unique solution after 9 trials
```

[43]:
```
# find free variables
free_vars = set()
for key in ans_set.keys():
    free_vars |= ans_set[key].free_symbols
# get the nonzero solution as a string
if len(free_vars) == 0:
```

```
    free_vars = set(s_var).difference(ans_set.keys())

subs_dict = {list(free_vars)[0]: 1}

ans_seq = []
for the_var in s_var:
    if the_var in ans_set.keys():
        ans_seq.append(ans_set[the_var].subs(subs_dict))
    else:
        ans_seq.append(1)

ans_seq
```

[43]:
```
[-3, 0, 0, 4, -8, 4, 2, -2, 3, 1]
```

`sympy.solve` does not handle polynomial over finite fields correctly, and we get bad guess when there are fractions with even numbers in the denominator. We try our own Gaussian elimination algorithm over the binary field GF(2). (If this happens, most likely we have not found enough conditions, and will get an incorrect guess anyway. The more sensible thing is to integrate this in the above search part.)

The Gaussian elimination goes as follows. * input: matrix A * output: the row echelon form B of A, and the list of pivot column indexes

1. start with copying A as B, setting m = number of rows, n = number of columns
2. set cur_row (this will be where we place the pivot) to 0
3. start with col index from 0 to n - 1, and transform B according the following:
    1. find a pivot row at or below the current row, in this column: start with r = cur_row to m - 1, set pivot = 0 if B[r][col] == 1
    2. if there is no pivot in this column, skip to next column.
    3. if we have found the pivot, swap the pivot row and the current row (if needed).
    4. record the pivot column index
    5. eliminate all 1's below the pivot in this column using XOR: start with r = cur_row + 1 to m - 1, and set B[r] to B[r] XOR B[cur_row] if B[r][col] == 1.
    6. if we've used all rows, stop.
    7. otherwise, increase cur_row by 1, and look for the next pivot.

[44]:
```
def gauss_elim_f2(A):
    """
    Perform Gaussian elimination over F2 (mod 2) using XOR.

    Parameters
    ----------
        A : list
            2-D list representing a binary matrix of shape (m, n)

    Returns
    -------
        B : list
            Row echelon form of A over F2
        pivots : list
            List of pivot column indices
    """
```

```
    B = A
    m, n = len(B), len(B[0])
    pivots = []
    row = 0

    for col in range(n):
        pivot = None
        for r in range(row, m):
            if B[r][col] == 1:
                pivot = r
                break

        if pivot is None:
            continue

        if pivot != row:
            B_piv = B[pivot]
            B[pivot] = B[row]
            B[row] = B_piv

        pivots.append(col)

        for r in range(row + 1, m):
            if B[r][col] == 1:
                B_r = B[r]
                B[r] = [B_r[j] ^ B[row][j] for j in range(n)]

        row += 1
        if row == m:
            break

    return B, pivots
```

[45]:
```
A, pivots = gauss_elim_f2(coeff_list)
A, pivots
```

[45]:
```
([[1, 1, 0, 1, 0, 0, 0, 1, 0, 1],
  [0, 1, 0, 0, 0, 0, 1, 1, 0, 0],
  [0, 0, 1, 0, 1, 1, 1, 1, 1, 1],
  [0, 0, 0, 1, 1, 1, 1, 0, 0, 0],
  [0, 0, 0, 0, 1, 0, 1, 0, 0, 0],
  [0, 0, 0, 0, 0, 1, 1, 0, 1, 1],
  [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
  [0, 0, 0, 0, 0, 0, 0, 0, 1, 1]],
 [0, 1, 2, 3, 4, 5, 6, 7, 8])
```

`pivot` gives the list of indexes for the dependent variables, while the missing one is free variable index

[46]:
```
free_var_index_s = set(range(k)).difference(set(pivots))
free_var_index_l = list(free_var_index_s)
```

[47]:
```
m = len(A)  # row count for A
n = len(free_var_index_l)
print(f"There are {n} free variables.")

for val_cand in it.product(range(2), repeat=n):
    if val_cand == (0,) * n:
        continue
    # initialize the free variables
    x = [0] * k
    for j in range(n):
        x[free_var_index_l[j]] = val_cand[j]

    for i in range(m - 1, -1, -1):
        Ai = A[i]
        if 1 not in Ai:
            continue
        first_nonzero_ent_ind = Ai.index(1)
        Ai_rest = Ai.copy()
        Ai_rest[first_nonzero_ent_ind] = 0
        new_val = sum(x[j] for j in range(k) if Ai_rest[j] == 1) % 2
        x[first_nonzero_ent_ind] = new_val
    print(x)
```

```
There are 1 free variables.
[1, 0, 0, 0, 0, 0, 0, 0, 1, 1]
```

How about the classical algorithm?

[48]:
```
# query count estimate
np.sqrt(2**k)
```

[48]:
```
32.0
```

[49]:
```
res_rec = {}  # dictionary of f(x) as key and x as value
for _ in range(2**k):
    x = tuple(rng.integers(2) for _ in range(k))
    if x in res_rec.values():
        continue
    f_x = f_from_s(x)
    if f_x in res_rec.keys():
        prev_x = res_rec[f_x]
        found_secret = tuple(x[i] ^ prev_x[i] for i in range(k))
        print(f"found secret {found_secret} after {1 + len(res_rec)}␣
 ↪queries")
        break
    # if f(x) is not found, record the results and continue with next x
    res_rec[f_x] = x
```

```
found secret (1, 0, 0, 0, 0, 0, 0, 0, 1, 1) after 28 queries
```

Reference: [Aar18, Sections 19.1, 19.2.1]

The RSA cryptosystem proposed by Rivest, Shamir, and Adleman[23] was the first widely used public-key cryptography system that guarantees secure data transmission over untrusted communication channels such as internet. It is asymmetric, in the sense that it allows one party to receive securely secret message from other parties even when the transmitted signals can be eavesdropped.

Let us describe the design of the protocol. We have the following players:

- Alice, the recipient of messages;
- Bob, Charlie, ..., who want to send messages to Alice;
- Eve, the eavesdropper.

The protocol works in the following way, starting with the preparation part:

(1) Alice chooses (big) prime numbers p and q;
(2) she chooses another integer e that does not have any prime factor p' that can divide both $p-1$ and $q-1$ (in other words, e is coprime to the least common multiple $\operatorname{lcm}(p-1, q-1)$);
(3) she finds d satisfying $ed \equiv 1 \bmod \operatorname{lcm}(p-1, q-1)$ (this is the *private key*);
(4) she advertises $N = pq$ and e as the *public key*.

Now, suppose that Bob wants to send a message represented by an integer x to Alice. The rest of the protocol goes as follows:

(5) Bob checks that x does not share prime factors with N (if it does have one of them as a prime factor, he asks for a new public key);
(6) he computes $y = x^e$;
(7) he sends y to Alice;
(8) Alice recovers x as $x \equiv y^d \bmod N$.

Why does this protocol provide a protection from Eve? The main point is only Alice knows p and q, which makes it easy to compute d, while just from knowing N and e, it takes very long time to work out d. Eve can capture N, e, and y from steps 4 and 7, but she then needs to work out d in order to know the message content $x = y^d$.

More precisely, the "extended Euclidean division algorithm" can compute d from p and q in a polynomial time in $\log N$ (see below), while finding p and q from N takes about $2^{C\sqrt{\log N \log\log N}}$-time for some constant $C > 1$ (which gives a bit better than N^C overall) in any known (classical) algorithms. Bob's check at step 5 can also be done with the Euclidean division algorithm.

15.1. **Extended Euclidean division algorithm.** Consider the following problem:

> Given integers a and b, find integers x and y satisfying $ax + by = \gcd(a, b)$.

Here, $\gcd(a, b)$ is the greatest common divisor of a and b. Regardless of the value of x and y, the $\gcd(a, b)$ always divides $ax+by$. So the problem is to find an *optimal* choice that makes $ax+by$ as small as possible besides the trivial answer $a \cdot 0 + b \cdot 0 = 0$.

Suppose a and b does not have common prime factors. Then we have $\gcd(a, b) = 1$, and the above solution x satisfies $ax \equiv 1 \bmod x$ from

$$ax = 1 - by \equiv 1 \bmod b.$$

For the RSA protocol, Alice would use $a = e$ and $b = \operatorname{lcm}(p-1, q-1)$.

Computing $\operatorname{lcm}(p-1, q-1)$ from p and q can be done in a similar way. This time she uses $a = p-1$ and $b = q-1$, and finds x and y that gives $ax + by = \gcd(a, b)$. Then $\operatorname{lcm}(a, b)$ can be computed as $ab/\gcd(a, b)$.

Now, let us describe the extended Euclidean division algorithm itself. We construct three sequences r_i, s_i, and t_i in the following way:

- $i = 0$: $r_0 = a$, $s_0 = 1$, $t_0 = 0$;
- $i = 1$: $r_1 = b$, $s_1 = 0$, $t_1 = 1$;

[23] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Commun. ACM* **21** (2), 120–126 (1978).

- the $(i+1)$-th step:
  - find q_i such that $r_{i+1} = r_{i-1} - q_i r_i$ holds for $0 \le r_{i+1} < r_i$;
  - if $r_{i+1} = 0$, then terminate the algorithm with output $x = s_i$, $y = t_i$, and $\gcd(a,b) = r_i$;
  - otherwise set $s_{i+1} = s_{i-1} - q_i s_i$ and $t_{i+1} = t_{i-1} - q_i t_i$ and proceed to the next step.

The running time is of order $(\log \min(a,b))^2$ if we take into account of the complexity of division problem at initial steps with large r_i[24].

Why does this work? The algorithm is set up to keep the condition

$$as_i + bt_i = r_i \quad (i = 0, 1, \ldots)$$

while making r_i smaller at each step. Indeed, this equality is obvious for $i = 0, 1$, and the induction step is compatible with it because we have

$$as_{i+1} + bt_{i+1} = a(s_{i-1} - q_i s_i) + b(t_{i-1} - q_i t_i) = r_{i-1} - q_i r_i = r_{i+1}.$$

We are guaranteed to get the correct answer because $\gcd(a,b)$ is the smallest positive integer solution k for $ax + by = k$ with (possibly negative) integers x and y.

15.2. **Why decoding works.** In the above description of the protocol, Bob encodes his message as $y = x^e$ using the public key e, and Alice decodes it as $x \equiv y^d \bmod N$ using her secret key d. The consistency condition between e and d was $e \cdot d \equiv 1 \bmod \operatorname{lcm}(p-1, q-1)$. Why does this work?

Example 15.1. As an example take $N = 15$, so that $p = 3$ and $q = 5$. We then have $\operatorname{lcm}(p-1, q-1) = \operatorname{lcm}(2,4) = 4$. One possible choice is $e = 3$ and $d = 3$, because we would have

$$e \cdot d = 9 = 2 \times 4 + 1 \equiv 1 \bmod 4.$$

Suppose Bob's message was $x = 2$. Then he computes $y = x^3 = 8$ and sends it to Alice. Alice computes $y^3 = 8^3 = 512 = 34 \cdot 15 + 2$, and she knows that the message was $x = 2$.

Let us set $c = \operatorname{lcm}(p-1, q-1)$. The key mathematical fact behind decoding scheme is the following.

Proposition 15.2. *For any integer x coprime to N, we have $x^{kc} \equiv 1 \bmod N$.*

The condition $e \cdot d \equiv 1 \bmod c$ means we have $e \cdot d = kc + 1$ for some integer k. We want to check $x \equiv y^d \bmod N$ from $y = x^e$, or in other words, we want $x \equiv x^{kc+1} \bmod N$. The above proposition gives $x^{kc} = k'N + 1$ for some integer k'. We then have

$$x^{kc+1} = x^{kc} \cdot x = xk'N + x \equiv x \bmod N,$$

as we wanted.

Proof of Proposition 15.2. Step 1: for any integer z, we have $z \equiv 1 \bmod N$ if and only if $z \equiv 1 \bmod p$ and $z \equiv 1 \bmod q$.

The allows us to reduce $x^{kc} \equiv 1 \bmod N$ to $x^{kc} \equiv 1 \bmod p$ and $x^{kc} \equiv 1 \bmod q$. To be precise, we want the "if" implication (which is the nontrivial one). Let us suppose $z \equiv 1 \bmod p$ and $z \equiv 1 \bmod q$. This means $z = ap + 1 = bq + 1$ for some integers a and b. We then have $ap = bq$, and since p and q are different prime numbers, q must divide a. Then we get $a = a'q$a and

$$z = a'pq + 1 = a'N + 1 \equiv 1 \bmod N.$$

Since p and q play the same role, it is enough to check $x^{kc} \equiv 1 \bmod p$.

Step 2: for any integer z coprime to p, we have $z^{p-1} \equiv 1 \bmod p$. This is (one form of) *Fermat's little theorem* which has many proofs, see for example the Wikipedia article.

Step 3: for any integer z coprime to p, we have $z^{kc} \equiv 1 \bmod p$. To see this, observe that $p-1$ divides $c = \operatorname{lcm}(p-1, q-1)$, and then it divides kc. If z_1, z_2 satisfy $z_i \equiv 1 \bmod p$, we have $z_1 z_2 \equiv 1 \bmod p$ from $(ap+1)(bp+1) = (ab + a + b)p + 1 \equiv 1 \bmod p$. Using this repeatedly, we get $z^{kc} \equiv 1 \bmod p$ from step 2. □

[24] D. E. Knuth, "The Art of Computer Programming", Volume 2: Seminumerical Algorithms (3rd ed.). Addison–Wesley. (1997), p. 373.

15.3. **What do we have to do to attack the RSA protocol?** If we can know the prime factors p and q from just knowing N then we can compute $c = \mathrm{lcm}(p-1, q-1)$ by Euclidean division algorithm and $\mathrm{lcm}(a,b) = ab/\gcd(a,b)$. Then the private key d (satisfying $ed \equiv 1 \bmod c$) can be worked out again with the extended Euclidean division as a solution for $ed + cy = 1$ (recall that Alice chose e as an integer coprime to c, hence $\gcd(e,c) = 1$ holds). For Eve, she can know N as part of public key, so this would be enough to decode Bob's message x from intercepting his transmission y to Alice.

Now, let us fix an integer z coprime to N, and consider the function $f(r) = z^r$. We want to know a small integer s that satisfies $f(r) = f(r+s) \bmod N$, in other words, a solution to $z^s = 1 \bmod N$. This is called the *period finding problem* for the function f.

If we find such s, which also happens to be an even number, then we can write

$$0 = z^s - 1 = (z^{\frac{s}{2}} - 1)(z^{\frac{s}{2}} + 1) \bmod N.$$

If moreover N does not divide neither of $z^{\frac{s}{2}} + 1$ and $z^{\frac{s}{2}} - 1$, then they must have some common prime factor with N, which should be p or q. Then we can know this common factor as $\gcd(z^{\frac{s}{2}} - 1, N)$ again using Euclidean division. If s is not even, or if N divides $z^{\frac{s}{2}} + 1$ or $z^{\frac{s}{2}} - 1$, then we choose a different z and repeat this argument again.

15.4. **Computational note for Lecture 15.**

[1]:
```
import numpy as np
np.set_printoptions(suppress=True, legacy='1.25')
```

15.4.1. *Extended Euclidean division algorithm.* find x and y satisfying

$$ax + by = \gcd(a, b)$$

construct three sequences r_i, s_i, and t_i in the following way: - $i = 0$: $r_0 = a$, $s_0 = 1$, $t_0 = 0$; - $i = 1$: $r_1 = b$, $s_1 = 0$, $t_1 = 1$; - the $(i + 1)$-th step: + find q_i such that $r_{i+1} = r_{i-1} - q_i r_i$ holds for $0 \leq r_{i+1} < r_i$; + if $r_{i+1} = 0$, then terminate the algorithm with output $x = s_i$, $y = t_i$, and $\gcd(a, b) = r_i$; + otherwise set $s_{i+1} = s_{i-1} - q_i s_i$ and $t_{i+1} = t_{i-1} - q_i t_i$ and proceed to the next step.

The algorithm is set up to keep the condition

$$as_i + bt_i = r_i \quad (i = 0, 1, \ldots)$$

while making r_i smaller at each step. At each step, r_{i+1} is the remainder of r_{i-1} divided by r_i, and q_i is the quotient We get the correct answer because $\gcd(a, b)$ is the smallest positive integer solution k for $ax + by = k$ with (possively negative) integers x and y.

[2]:
```
def ext_Euclid_div(a, b, report=False):
    """Returns (gcd(a,b), x, y) such that a * x + b * y = gcd(a, b)."""
    # i = 0 and i = 1
    r = [a, b]
    s = [1, 0]
    t = [0, 1]
    for i in range(2, max(a, b)**2):
        q_i, rem = divmod(r[-2], r[-1])
        if rem == 0:
            if report:
                print(f"found {r[-1]} = gcd({a}, {b}) after {i} steps")
            return (r[-1], s[-1], t[-1])
        # add next terms, then continue
        r.append(rem)
        s.append(s[-2] - q_i * s[-1])
        t.append(t[-2] - q_i * t[-1])
        if report:
            print(f"we now have r: {r[-1]}, s: {s[-1]}, t: {t[-1]}")
```

[3]:
```
a, b = 40902, 24140  # from Knuth's book
gcd, x, y = ext_Euclid_div(a, b, report=True)
```

```
we now have r: 16762, s: 1, t: -1
we now have r: 7378, s: -1, t: 2
we now have r: 2006, s: 3, t: -5
we now have r: 1360, s: -10, t: 17
we now have r: 646, s: 13, t: -22
we now have r: 68, s: -36, t: 61
we now have r: 34, s: 337, t: -571
found 34 = gcd(40902, 24140) after 9 steps
```

[4]:
```
a * x + b * y, gcd == a * x + b * y
```

[4]: (34, True)

15.4.2. *RSA cryptsystem.* Preparation: first Alice chooses (big) prime numbers p and q

[5]:
```
# helper function to compute exponentioal modulo k for big exponent
from functools import reduce


def pow_mod(base, expo, k):
    """Returns base**expo mod k, allowing numpy integer inputs."""
    if expo < 0:
        return None
    return reduce(lambda s, t: (s * t) % k, [base] * expo)
```

We take lange integers at random and check if it is prime or not by the Miller-Rabin primality test. The code is taken from Rosetta Code and lightly modified.

[6]:
```
import numpy as np
rng = np.random.default_rng()


def is_prime(n: np.int64):
    """
    Miller-Rabin primality test.

    A return value of False means n is certainly not prime. A return␣
 ↪value of
    True means n is very likely a prime.
    """
    # special cases for small n
    if n in [2, 3, 5, 7]:
        return True
    elif n < 10:
        return False

    s = 0
    d = n-1
    while d % 2 == 0:
        d >>= 1
        s += 1
    assert (2**s * d == n-1)

    def trial_composite(a):
        if pow_mod(a, d, n) == 1:
            return False
        for i in range(s):
            if pow_mod(a, 2**i * d, n) == n-1:
                return False
        return True
```

```
    for i in range(8):  # number of trials
        a = rng.integers(2, n)
        if trial_composite(a):
            return False

    return True


def getLargePrime(keysize=1024, report=False):
    """Generate a random prime number with specified keysize"""
    for try_count in range(keysize ** 2):
        num = rng.integers(2**(keysize-1), 2**(keysize))
        if is_prime(num):
            if report:
                print(f"found a prime number {num} after {try_count}␣
 ↪trials.")
            return num
```

[7]:
```
p = getLargePrime(10, report=True)
q = getLargePrime(10, report=True)
p, q
```

```
found a prime number 863 after 6 trials.
found a prime number 547 after 1 trials.
```

[7]:
```
(863, 547)
```

she chooses another integer e that is coprime to the least common multiple $\text{lcm}(p-1, q-1)$

[8]:
```
gcd, _, _ = ext_Euclid_div(p-1, q-1)
c = int((p - 1) * (q - 1) / gcd)  # lcm
e_cands = [3, 5, 7, 11, 13, 17, 19]
for e_cand in e_cands:
    gcd_e_c, _, _ = ext_Euclid_div(e_cand, c)
    if gcd_e_c == 1:
        e = e_cand
        break
    e = None
c, e
```

[8]:
```
(235326, 5)
```

she finds d satisfying $ed \equiv 1 \bmod \text{lcm}(p-1, q-1)$ (this is the *private key*)

[9]:
```
_, d, _ = ext_Euclid_div(e, c)
if d < 0:
    d = d % c
d, (e * d) % c
```

[9]:
```
(188261, 1)
```

she advertises $N = pq$ and e as the *public key*.

[10]:
```
N = p * q
print(f"Public key: N={N}, e={e}")
```

```
Public key: N=472061, e=5
```

Now, suppose that Bob wants to send a message represented by an integer x to Alice. He first checks that x does not share prime factors with N (if it does have one of them as a prime factor, he asks for a new public key)

[11]:
```
x = 42
```

[12]:
```
gcd, _, _ = ext_Euclid_div(x, N)
gcd == 1  # should be true
```

[12]:
```
True
```

he computes $y = x^e$, and sends it to Alice

[13]:
```
y = pow_mod(x, e, N)
y
```

[13]:
```
402396
```

Alice recoves x as $x \equiv y^d \bmod N$.

[14]:
```
pow_mod(y, d, N)
```

[14]:
```
42
```

Reference: [Aar18, Section 19.2.2]

Now, let us explain Shor's quantum algorithm[25] to find s satisfying $z^s = 1 \bmod N$, that would solve the RSA cryptosystem as we discussed above.

We set $Q = 2^k$, and consider indexes $0 \le x < Q$. This will be about N^2, and eventually we will need about $2Q$ qubits, i.e., $4 \log_2 N$ qubits.

Since the k-qubit space $(\mathbb{C}^2)^{\otimes k}$ has dimension Q, we can use such x as index for basis, and write $\mathbb{C}^Q$ for the Hilbert space. Concretely, given a k-bit string $x_0 \cdots x_{k-1}$, we take the integer

$$x = x_0 + 2x_1 + \cdots + 2^{k-1}x_{k-1}$$

represented by the binary digit expansion $x_0 \cdots x_{k-1}$. (To match our convention for the Kronecker product, it's also sensible to say that $x_0 \cdots x_{k-1}$ corresponds to $2^{k-1}x_0 + \cdots + x_{k-1}$, or equivalently, $x_{k-1} \cdots x_0$ corresponds to $x_0 + 2x_1 + \cdots + 2^{k-1}x_{k-1}$.)

16.1. **Part 1: quantum parallelism.** Given a function $f(x)$ defined on $0 \le x < Q$, consider the XOR query gate on $\mathbb{C}^Q \otimes \mathbb{C}^Q$

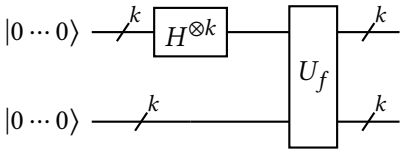
$$U_f|xy\rangle = |xz\rangle\,, \quad z = y \oplus f(x), \quad (0 \le x, y < Q),$$

where $z = y \oplus f(x)$ is the integer represented by bit-wise XOR of the k-bit strings for y and $f(x)$. We consider this for the function

$$f(x) = z^x \bmod N$$

for a fixed z coprime to N.

If we look at the effect of the quantum circuit on $2k$ qubits given by



we have the state vector

$$v = \frac{1}{\sqrt{Q}} \sum_{0 \le x < Q} |xf(x)\rangle\,.$$

from

$$H^{\otimes k}|0 \cdots 0\rangle = \frac{1}{\sqrt{2^k}} \sum_{x \in \{0,1\}^k} |x\rangle$$

and $0 \oplus f(x) = f(x)$, as in Step 2 of the proof for Proposition 14.2 for Simon's algorithm.

Suppose s is the period of z modulo N, i.e., the smallest positive integer such that $z^s = 1 \bmod N$ holds. (This is what we want to know.) We then have

$$f(x+s) = z^{x+s} = z^x z^s = z^x \bmod N,$$

and similarly $f(x) = f(x+ks)$ holds for all integers k. For each $0 \le r < S$, there are about $\frac{Q}{s}$ different input integers

$$r, \quad r+s, \quad r+2s, \quad \ldots$$

between 0 and Q with the same output $w = f(r)$, so that v contains terms of the form

$$|rw\rangle\,, \quad |(r+s)w\rangle\,, \quad |(r+2s)w\rangle\,, \quad \ldots\,.$$

Summarizing the above, there are s different possibilities $f(0), f(1), \ldots, f(s-1)$ for the possible values of w, and for each such $w = f(r)$ we have

$$|rw\rangle + |(r+s)w\rangle + \cdots + |(r+(L-1)s)w\rangle \tag{16.1}$$

for an integer L about Q/s. (But we don't know the value of s!)

[25] Shor, P.W. (1994). "Algorithms for quantum computation: Discrete logarithms and factoring". Proceedings 35th Annual Symposium on Foundations of Computer Science. pp. 124–134. doi:10.1109/sfcs.1994.365700.

16.2. **Part 2: Fourier transform.** Reference: [NC00, Section 5.3.1; Aar18, Section 20.1]

Now, suppose that s divides Q, so that $L = Q/s$ is an integer. We want to know the value of s from measurement of the first half of the state v. Observe that (16.1) is like a wave of the *wavelength* s as in Figure 16.1.

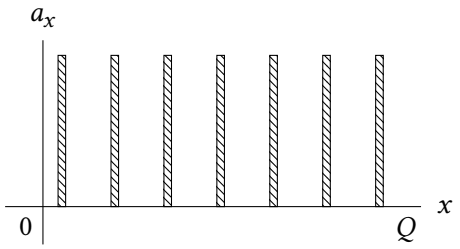


FIGURE 16.1.

The *Fourier transform* is a perfect tool for extracting wavelength / frequency out of a series of data. Suppose we have a sequence of (complex) numbers

$$a_0, a_1, \dots, a_{Q-1}$$

with Q terms. (For us, it will be the coefficients of state vectors in $\mathbb{C}^Q$.) Then the Fourier transform F_Q gives another sequence $b = F_Q(a)$ defined by

$$b_j = \frac{1}{\sqrt{Q}} \sum_{l=0}^{Q-1} \exp\left(\frac{2\pi i}{Q} jl\right) a_l.$$

(Recall $\exp(i\theta) = \cos\theta + i\sin\theta$.) In other words, F_Q is a $Q \times Q$-matrix whose components are

$$[F_Q]_{jl} = \frac{1}{\sqrt{Q}} \exp\left(\frac{2\pi i}{Q} jl\right) \quad (j, l = 0, \dots, Q-1).$$

Proposition 16.1. *Suppose we have*

$$a_r = a_{r+s} = \cdots = a_{r+(L-1)s} = \frac{1}{\sqrt{L}}$$

and $a_j = 0$ for j not of the form $r + ns$. Then $b = F_Q(a)$ is given by

$$b_0 = \frac{1}{\sqrt{s}}, b_L = \frac{1}{\sqrt{s}} \exp\left(\frac{2\pi i}{Q} r\right), b_{2L} = \frac{1}{\sqrt{s}} \exp\left(\frac{2\pi i}{Q} 2r\right), \dots, b_{(s-1)L} = \frac{1}{\sqrt{s}} \exp\left(\frac{2\pi i}{Q}(s-1)r\right)$$

and $b_j = 0$ for j not of the form cL.

Proposition 16.2. *The transform F_Q is a unitary transform on $\mathbb{C}^Q$.*

Now, consider the state

$$(F_Q \otimes I_{2^k})v = (F_Q \otimes I_{2^k}) U_f (H^{\otimes k} \otimes I_{2^k}) |0 \cdots 0\rangle.$$

When we look at the first half, Proposition 16.1 implies that the coefficients are concentrated on the computational basis vectors $|x\rangle$ for $x = 0, L, 2L, \dots (s-1)L$. By making measurement of this part several times, we get integers of the from $c_1 L$, $c_2 L$, ...for different c_j. We can then take the greatest common divisor (using the Euclidean division algorithm) to know L, and hence $s = Q/L$.

Proof of Proposition 16.1. We are adding up the k-th columns of F_Q for $k = r, r+s, r+2s, \dots, r+(L-1)s$. Looking at the j-th component, we have

$$\frac{1}{\sqrt{QL}} \left(\omega^{jr} + \omega^{j(r+s)} + \cdots + \omega^{j(r+(L-1)s)}\right), \quad \omega = \exp\left(\frac{2\pi i}{Q}\right).$$

The denominator is $\sqrt{QL} = \sqrt{sLL} = L\sqrt{s}$, and each term in the sum has the common factor ω^{jr}. The claim is equivalent to

$$1 + \omega^{js} + \omega^{2js} + \cdots + \omega^{(L-1)js} = \begin{cases} L & (j = cL) \\ 0 & \text{otherwise.} \end{cases}$$

When j is of the from cL, we have $\omega^{kjs} = \omega^{kcLs} = (\omega^Q)^{kc} = 1$ from $\omega^Q = e^{2\pi i} = 1$. Otherwise we have a sum of evenly distributed points on the unit circle of the complex plane, hence we get 0 (the center of the unit circle), see Figure 16.2. □

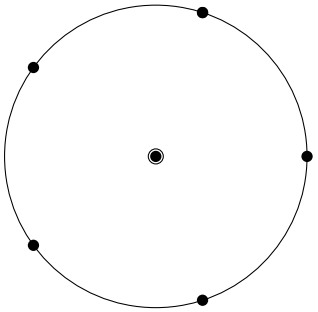


FIGURE 16.2. Center is the average of evenly distributed points on the circle

Proof of Proposition 16.2. The adjoint matrix for F_Q is $[F_Q^\dagger]_{jl} = \frac{1}{\sqrt{Q}}\omega^{-jl}$. Computing the (j,l)-component of the matrix product $F_Q F_Q^\dagger$, we get

$$[F_Q F_Q^\dagger]_{j,l} = \sum_{l=0}^{Q-1} [F_Q]_{j,l}[F_Q^\dagger]_{l,l} = \frac{1}{Q}\sum_l \omega^{l(j-l)}.$$

When $j = l$, each term in the sum is 1 and we get $[F_Q F_Q^\dagger]_{j,j} = 1$. Otherwise we get $[F_Q F_Q^\dagger]_{j,l} = 0$ from the same argument as in the last part of the proof for Proposition 16.1. □

16.3. **Part 3: continued fraction approximation.** Reference: [NC00, Section A4.4; Aar18, Section 21.1]

If s does not divide Q, the above estimate still holds approximately, and measurement of $(F_Q \otimes I_{2^k})v$ in the computational basis $(|x\rangle)_{x=0}^{Q-1}$ gives the outcome case x for

$$x = c\frac{Q}{s} \pm \epsilon$$

with some $0 \le c < s$ and $0 \le \epsilon < 1$. In other words, we get

$$\frac{x}{Q} = \frac{c}{s} \pm \epsilon'$$

with $\epsilon' < \frac{1}{Q}$. If we know that s is small compared to Q, this puts some restriction on s. For our convention, s is not bigger than N while we can choose Q to be about N^2, then the above equality can happen for limited number of c and s.

To make this idea precise, we use the *continued fraction*. This is a way to represent (rational) numbers in the form

$$a = c_0 + \cfrac{1}{c_1 + \cfrac{1}{c_2 + \cfrac{1}{\ddots + \cfrac{1}{c_M}}}}$$

using integers $c_0, \ldots, c_M$. We write $[c_0; c_1, \ldots, c_M]$ for this formula. Any positive rational number has a continued fraction representation with some *positive* integers $c_0, \ldots, c_M$.

Remark 16.3. If a is a rational number satisfying $0 < a < 1$, then we would have $c_0 = 0$ in the representation $a = [c_0; c_1, \dots, c_M]$. The other terms c_i are nonzero. (Although we can allow $[c_0; 0, 0, c_1] = [c_0; c_1]$, etc.) If the last term c_M is equal to 1, we can also reduce the length by 1 using $[c_0; c_1, \dots, c_{M-1}, 1] = [c_0; c_1, \dots, c_{M-2}, c_{M-1} + 1]$.

Proposition 16.4. *Let a be a rational number. If another rational number $b = \frac{n}{d}$ satisfies the estimate*

$$|b - a| \leq \frac{1}{2d^2},$$

then b can be computed from the continued fraction presentation of a: if we have

$$a = [c_0; c_1, \cdots, c_M] = c_0 + \cfrac{1}{c_1 + \cfrac{1}{c_2 + \cfrac{1}{\ddots + \cfrac{1}{c_M}}}}$$

for positive integers $c_0, \dots, c_M$, then we have

$$b = [c_0; c_1, \cdots, c_m] = c_0 + \cfrac{1}{c_1 + \cfrac{1}{c_2 + \cfrac{1}{\ddots + \cfrac{1}{c_m}}}}$$

for some integer $m \leq M$.

We can use this for $a = x/Q$, with the measurement outcome x, to estimate $b = c/s$. We first find the coefficient integers $c_0, \cdots, c_M$ satisfying $a = [c_0; c_1, \cdots, c_M]$. (Since $x < Q$, we actually have $c_0 = 0$.)

Proof of Proposition 16.4. Let us choose (positive) integers $c_0, \dots, c_m$ satisfying $b = [c_0; c_1, \dots, c_m]$. We want to find $c_{m+1}, \dots, c_M$ such that $a = [c_0; c_1, \dots, c_M]$ holds. Writing

$$a = b + \frac{\delta}{2d_m^2}$$

for $|\delta| \leq 1$, we adjust m (see Remark 16.3) so that δ has the sign $(-1)^m$.

Consider the integer sequences n_i and d_i defined by

$$n_0 = c_0, \quad d_0 = 1, \quad n_1 = 1 + c_0 c_1, \quad d_1 = c_1, \quad n_i = c_i n_{i-1} + n_{i-2}, \quad d_i = c_i d_{i-1} + d_{i-2}.$$

We then have

$$\frac{n_i}{d_i} = [c_0; c_1, \dots, c_i]$$

for each $0 \leq i \leq m$. This can be checked by induction, based on

$$[c_0; c_1, \dots, c_{i+1}] = \left[c_0; c_1, \dots, c_{i-1}, c_i + \frac{1}{c_{i+1}}\right].$$

Now, let us find a rational number λ that satisfies $a = [c_0, \dots, c_m, \lambda]$. Again by the above recursive relation for the n_i and the d_i, this is equivalent to

$$a = \frac{\lambda n_m + n_{m-1}}{\lambda d_m + d_{m-1}}.$$

We see that

$$\lambda = \frac{n_{m-1} - a d_{m-1}}{a d_m - n_m} = 2\frac{d_m n_{m-1} - n_m d_{m-1}}{\delta} - \frac{d_{m-1}}{d_m}$$

works.

Recall that δ has the sign $(-1)^m$. As for $d_m n_{m-1} - n_m d_{m-1}$, we have $d_m n_{m-1} - n_m d_{m-1} = (-1)^m$. This can be seen by induction using

$$d_m n_{m-1} - n_m d_{m-1} = (c_m d_{m-1} + d_{m-2}) n_{m-1} - (c_m n_{m-1} + n_{m-2}) d_{m-1} = d_{m-2} n_{m-1} - n_{m-2} d_{m-1}.$$

Hence these signs match. Then λ is bigger than 1, since $(d_m n_{m-1} - n_m d_{m-1})/\delta$ is bigger than 1 while d_{m-1}/d_m is smaller than 1, as d_m is bigger than d_{m-1}.

Now we can write the continued fraction

$$\lambda = [c_0'; \dots, c_k']$$

with positive integers $c'_0, \dots, c'_k$. Setting $c_{m+1} = c'_0$, $c_{m+2} = c'_1$, and so on, we get the claim. □

16.3.1. *Digression: infinite continued fractions.* Similarly, we can represent irrational numbers r by *infinite continued fraction*

$$r = c_0 + \cfrac{1}{c_1 + \cfrac{1}{c_2 + \cdots}},$$

which gives a much better approximation than the usual digit expansion. For example, we have

$$\sqrt{2} = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cdots}} = [1; 2, 2, \cdots]$$

with $c_0 = 1$, $c_1 = c_2 = \cdots = 2$. If we stop this fraction at $c_M = 2$, the rational number

$$a_M = 1 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{\ddots + \frac{1}{2}}}} \quad (2 \text{ appears } M \text{ times})$$

with denominator Q_M and numerator P_M satisfies

$$\left|\sqrt{2} - a_M\right| = \frac{C}{Q_M^2}$$

for some $C > 0$. (Moreover, we have $Q_M \sim (1 + \sqrt{2})^M$). In contrast, if we use the digit expansion $\sqrt{2} = 1.d_1 d_2 \cdots$, the M-th cut-off $b_M = 1.d_1 \cdots d_M$ satisfies

$$\left|\sqrt{2} - a_M\right| \sim \frac{1}{10^M},$$

and 10^M is the denominator of b_M.

16.4. **Computational note for Lecture 16.**

[1]:
```
import numpy as np
np.set_printoptions(suppress=True, legacy='1.25')
rng = np.random.default_rng()
import itertools as it
from functools import reduce
```

[2]:
```
N = 15
k = 4
Q = 2**k
# this should be bigger than N^2 to allow s not dividing Q
# for N=15, we know s=2 (z=4,11,14) or s=4 (z=2,7,8,13) will divide Q,
# so Q > N is enough
```

[3]:
```
zeroket = np.array([1, 0])
oneket = np.array([0, 1])
```

[4]:
```
def bin_seq_for_int(x: int):
    """Returns binary sequence representation of the integer x.

    Parameters
    ==========
        x: int

    Returns
    =======
        tuple of int
          (x0, x1, ..., x(k-1)) with xi = 0, 1 and
          x = x0 + 2 * x1 + ... + 2**(k-1) * x(k-1)
    """
    # for i=0,..., k-1, do bit shift by i and bitwise AND with 1
    return tuple((x >> i) & 1 for i in range(k))


def int_from_bin_seq(x_seq):
    """Returns integer for the binary sequence of length k."""
    return sum(2**n * x_seq[n] for n in range(k))
```

[5]:
```
bin_seq_for_int(1), bin_seq_for_int(3)
```

[5]:
```
((1, 0, 0, 0), (1, 1, 0, 0))
```

16.4.1. *Fourier transform.*

[6]:
```
def Fourier(n, j, l):
    """Returns the coefficient of Fourier transform matrix of size n."""
    return np.exp(2 * 1j * np.pi * j * l / n) / np.sqrt(n)
```

Me need to take care of different conventions for indexing: On one hand, we say bit strings $x_0x_1\cdots x_{k-1}$ represents the integer

$$x = x_0 + 2x_1 + \cdots 2^{k-1}x_{k-1}.$$

On the other, the Kronecker product convention says that this string corresponds to the j-th row / column of the matrix, where

$$j = 2^{k-1}x_0 + 2^{k-2}x_1 + \cdots + x_{k-1}.$$

[7]:
```
bin_seq_for_int(1), bin_seq_for_int(1)[::-1]
```

[7]:
```
((1, 0, 0, 0), (0, 0, 0, 1))
```

[8]:
```
def converted_index(x):
    """Returns j = 2^{k-1} x_0 + 2^{k-2} x_1 + ... + x_{k-1}."""
    x_seq = bin_seq_for_int(x)
    j = sum(2**(k-1-n) * x_seq[n] for n in range(k))
    return j


# x = 1 gives the sequence (1, 0, 0, 0), hence j = 8, and vice versa
# x = 3 gives the sequence (1, 1, 0, 0), hence j = 12, and vice versa
converted_index(1), converted_index(3)
```

[8]:
```
(8, 12)
```

[9]:
```
Fourier_Q_mat = np.array([[Fourier(Q, converted_index(j),␣
 ↪converted_index(l))
                           for l in range(Q)] for j in range(Q)])
```

16.4.2. *Shor's algorithm for period finding.*

[10]:
```
def tens_pow_H(k):
    # tensor product of copies of the Hadamard matrix
    H = np.array([[1, 1],
                  [1, -1]]) * (1/np.sqrt(2))
    return reduce(np.kron, [H] * k)
```

[11]:
```
def termwise_XOR(x, y):
    # termwise XOR of tuples
    return tuple(xi ^ yi for xi, yi in it.zip_longest(x, y))


def comp_basis(x_seq):
    """Returns computational basis vector for binary sequence.

    Parameters
    ==========
        x_seq: list of int

    Returns
    =======
        numpy.array
          1-dimensional array representing |x⟩
    """
    vec_seq = (zeroket if xi == 0 else oneket for xi in x_seq)
```

```
    return reduce(np.kron, vec_seq)


def U(k, f):
    # U_f gate for the Shor's algorithm
    res = np.zeros((2**(2*k), 2**(2*k)), dtype=np.int8)
    for in_x in range(Q):
        in_x_seq = bin_seq_for_int(in_x)
        f_x_seq = bin_seq_for_int(f(in_x))
        out_x_seq = in_x_seq + f_x_seq
        in_as_row = np.atleast_2d(comp_basis(in_x_seq + (0,) * k))
        out_as_column = np.atleast_2d(comp_basis(out_x_seq)).T
        res += out_as_column @ in_as_row
    return res
```

[12]:
```
z = 2  # coprime to N=15, period 4


def f_from_z(x):
    return z**x % N
```

[13]:
```
H_pow_ampl = np.kron(tens_pow_H(k), np.identity(2**k))
F_Q_ampl = np.kron(Fourier_Q_mat, np.identity(2**k))
Shor_gate = F_Q_ampl @ U(k, f_from_z) @ H_pow_ampl
```

[14]:
```
v = Shor_gate @ comp_basis((0,) * (2 * k))
```

[15]:
```
def proj(x):
    """Return the projection for the computational basis."""
    basis_vec = np.atleast_2d(comp_basis(x))
    return basis_vec.T @ basis_vec


def meas_prob(x):
    """Return the probability of observing x in the first half of v."""
    return np.real(np.dot(np.conjugate(v), np.kron(proj(x), np.
  ↪identity(2**k))
                          @ v))
```

[16]:
```
intervals = {}
t = 0.0
for x in it.product(range(2), repeat=k):
    prob_for_x = meas_prob(x)
    intervals[x] = (t, t + prob_for_x)
    t += prob_for_x
    print(f"{x} with probability {prob_for_x}")
```

```
(0, 0, 0, 0) with probability 0.24999999999999983
(0, 0, 0, 1) with probability 0.24999999999999983
(0, 0, 1, 0) with probability 0.24999999999999983
```

```
(0, 0, 1, 1) with probability 0.24999999999999983
(0, 1, 0, 0) with probability 2.4470051813928706e-33
(0, 1, 0, 1) with probability 4.073856156508302e-31
(0, 1, 1, 0) with probability 4.498781880176817e-32
(0, 1, 1, 1) with probability 2.189959002944489e-31
(1, 0, 0, 0) with probability 3.186062412001537e-33
(1, 0, 0, 1) with probability 1.979696497851188e-31
(1, 0, 1, 0) with probability 5.554620943171135e-32
(1, 0, 1, 1) with probability 5.289982201335954e-31
(1, 1, 0, 0) with probability 1.159853971866433e-32
(1, 1, 0, 1) with probability 7.433780893255674e-31
(1, 1, 1, 0) with probability 1.2971604543390791e-31
(1, 1, 1, 1) with probability 1.1455302955436829e-30
```

[17]:
```python
x_samples = []
for _ in range(k):  # when k is small, it's better to repeat a bit more
    rand_num = rng.random()
    for x in it.product(range(2), repeat=k):
        if intervals[x][0] < rand_num and rand_num < intervals[x][1]:
            x_samples.append(x)
            break
```

[18]:
```python
x_samples  # binary sequnces replesenting c*L for L = Q/s
```

[18]:
```
[(0, 0, 0, 0), (0, 0, 1, 1), (0, 0, 1, 1), (0, 0, 1, 1)]
```

[19]:
```python
# get L as the greatest common divisor of integers c*L
from math import gcd
L = reduce(gcd, (int_from_bin_seq(x_seq) for x_seq in x_samples))
s = Q // L
L, s
```

[19]:
```
(12, 1)
```

16.4.3. *continued fraction.* code sample from sympy documentation

[20]:
```python
import sympy as sy
```

[21]:
```python
def list_to_frac(coeff_list):
    """Returns continued fraction with the given coefficients."""
    expr = sy.Integer(0)
    for i in reversed(coeff_list[1:]):
        expr += i
        expr = 1/expr
    return coeff_list[0] + expr
```

continued fraction approximation of $\sqrt{2} = [1; 2, 2, \cdots]$

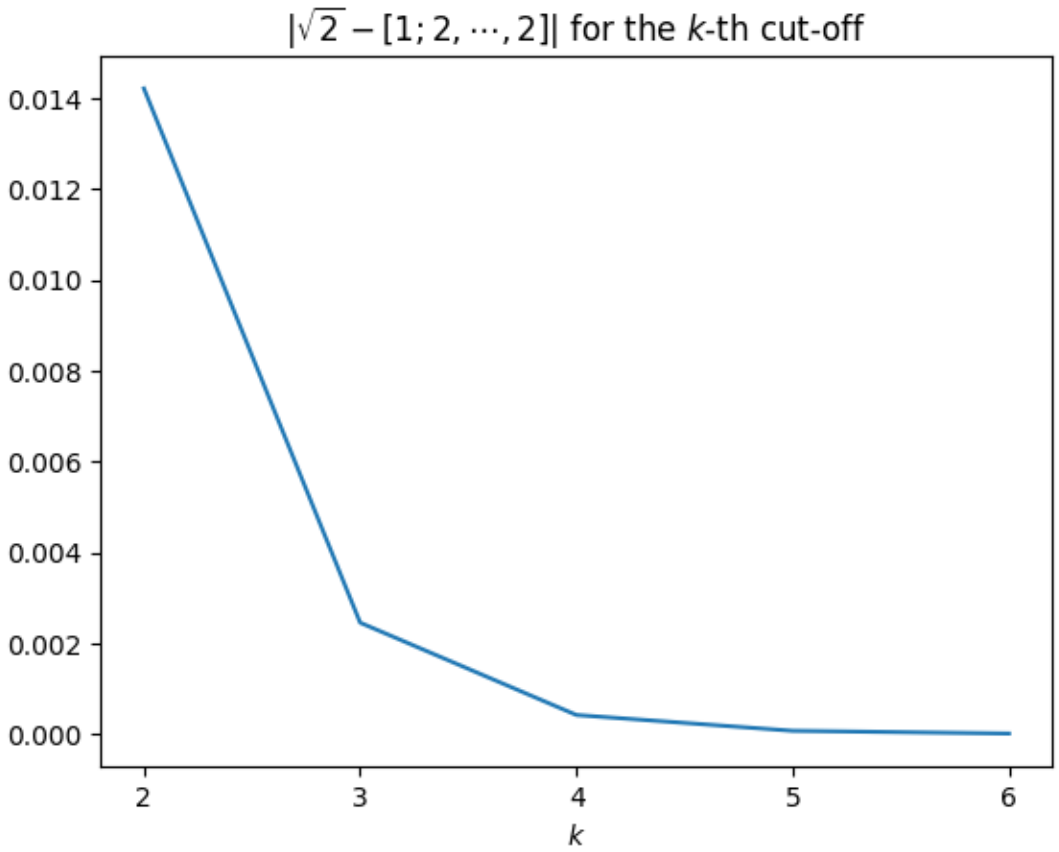
[22]:
```python
list_to_frac([1, 2]), list_to_frac([1, 2, 2]), \
  list_to_frac([1, 2, 2, 2])
```

[22]:
```
(3/2, 7/5, 17/12)
```

[23]:
```
def sqrt2_approx_error(k):
    """Approximation error of continued fracton of sqrt(2)."""
    frac_at_k = list_to_frac([1] + [2] * k)
    error = sy.N(sy.sqrt(2) - frac_at_k)
    err_abs = error if error > 0 else -error
    return err_abs
```

[24]:
```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
k_seq = [k for k in range(2, 7)]
y_seq = [sqrt2_approx_error(k) for k in k_seq]
ax.set_title("$|\\sqrt{2}-[1;2,\\cdots,2]|$ for the $k$-th cut-off")
ax.set_xticks(k_seq)
ax.set_xlabel("$k$")
ax.plot(k_seq, y_seq)
```

[24]:
```
[<matplotlib.lines.Line2D at 0x1233a7a10>]
```



error estimate with signs

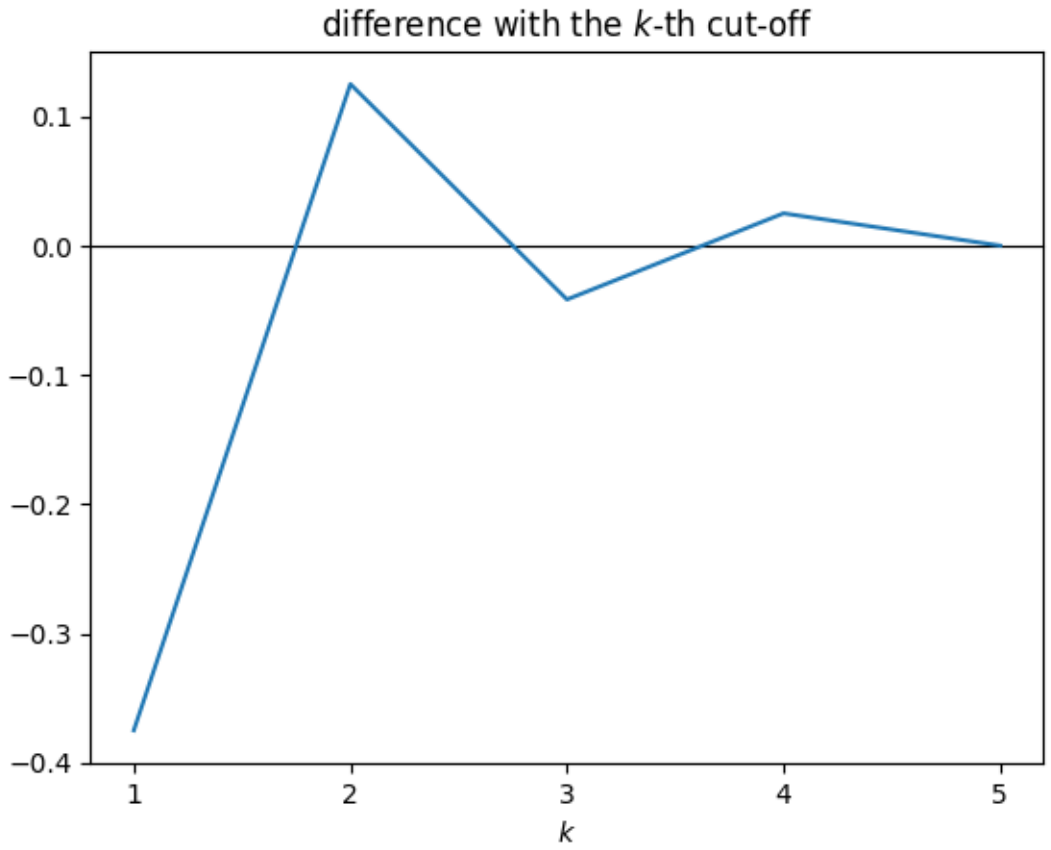
[25]:
```
c_seq = [1, 1, 1, 1, 1, 1]
cont_frac = list_to_frac(c_seq)
cont_frac
```

[25]: $\frac{13}{8}$

[26]:
```
def error_from_cutoff(k):
    """Difference between the cut-off at the k-th term (inclusive)."""
    cut_off = list_to_frac(c_seq[0:k+1])
    return cont_frac - cut_off
```

[27]:
```
fig2, ax2 = plt.subplots()
k_seq = [k for k in range(1, len(c_seq))]
y_seq = [error_from_cutoff(k) for k in k_seq]
ax2.set_title("difference with the $k$-th cut-off")
ax2.set_xticks(k_seq)
ax2.set_xlabel("$k$")
sec_xax = ax2.secondary_xaxis(0, transform=ax2.transData)
sec_xax.set_xticks([])
ax2.plot(k_seq, y_seq)
```

[27]: [<matplotlib.lines.Line2D at 0x1237b02f0>]



[28]:
```
def coeff_for_cont_frac(rat: sy.Rational):
    """Returns the coefficients for continued fraction.

    Parameters
    ----------
```

```
    rat : sympy.Rational

    Returns
    -------
    list
        list of integers c0, c1, ..., cM satisfying
        rat = [c0; c1, ..., cM].
    """
    rat = sy.simplify(rat)
    num, denom = rat.as_numer_denom()
    if denom == 1:
        return [num]
    c0 = num // denom
    next_rat = 1 / (rat - c0)
    return [c0] + coeff_for_cont_frac(next_rat)
```

[29]:
```
coeff_for_cont_frac(sy.Rational(5, 2))
```

[29]:
```
[2, 2]
```

In Shor's algorithm, we want to estimate $b = \frac{c}{s}$ from $a = \frac{x}{Q}$ using $s < N$, $N^2 < Q$, and $|a-b| \leq \frac{1}{Q} < \frac{1}{N^2}$

[30]:
```
N = 21  # 3 * 7
Q = 2**9  # 512 > 21 * 21 = 441
```

if we choose $z = 2$ (coprime to $N = 21$), we actually know that the period is $s = 6$ from $2^6 = 64 = 3 \times 21 + 1$.

[31]:
```
[2**a % N for a in range(1, N)]
```

[31]:
```
[2, 4, 8, 16, 11, 1, 2, 4, 8, 16, 11, 1, 2, 4, 8, 16, 11, 1, 2, 4]
```

we can get $x = 85$, that makes a closer to $\frac{1}{6}$

[32]:
```
x = 85
a = sy.Rational(x, Q)
sy.N(a - sy.Rational(1, 6)), sy.N(sy.Rational(1, N**2))
```

[32]:
```
(-0.000651041666666667, 0.00226757369614512)
```

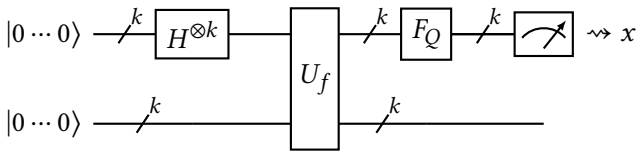
To work out s from a, we get the continued fraction coefficients of a, and find a candidate of b that has small denominator ($s < N$) but close enough to a

[33]:
```
a_coeff = coeff_for_cont_frac(a)


def cutoff_denom(k):
    cutoff_rat = sy.simplify(list_to_frac(a_coeff[:k+1]))
    _, denom = cutoff_rat.as_numer_denom()
    return denom


for k in range(len(a_coeff)):
```

```
    b_cand = sy.simplify(list_to_frac(a_coeff[:k+1]))
    b_cand_denom = cutoff_denom(k)
    print(f"cutoff at k={k} gives b={b_cand} with␣
  ↪denominator={b_cand_denom}")
    error = sy.N(a - b_cand)
    if b_cand_denom > N:
        print(f"👎 the denominator is too big against N={N}.")
    elif error > sy.N(1/N**2):
        print(f"👎 the error {error} is too big against 1/N**2 = {sy.N(1/
  ↪N**2)}")
    else:
        print("👍 this is a good candidate!")
```

```
cutoff at k=0 gives b=0 with denominator=1
👎 the error 0.166015625000000 is too big against 1/N**2 = 0.
  ↪00226757369614512
cutoff at k=1 gives b=1/6 with denominator=6
👍 this is a good candidate!
cutoff at k=2 gives b=42/253 with denominator=253
👎 the denominator is too big against N=21.
cutoff at k=3 gives b=85/512 with denominator=512
👎 the denominator is too big against N=21.
```

Let us analyze how many basic operation it takes to implement Shor's algorithm from previous section. Overall, we have the quantum circuit



and use its measurement outcome x to get a rational number $a = \frac{x}{Q}$, then use the continued fraction approximation to estimate $b = \frac{c}{s}$ with $s < N$ and $|a - b| < \frac{1}{N^2}$. (We need to be careful about the indexing for F_Q, because bit-representation of integers $x = x_0 + 2x_1 + \cdots 2^{k-1}x_{k-1}$ is not compatible with the convention of the Kronecker product.)

We see that there are k copies of the Hadamard gate H, but we need to estimate how many basic operations we need to implement the XOR query gate U_f and the Fourier transform F_Q.

17.1. **XOR query gate.** The function $f(x) = z^x$ can be computed using *repeated squaring* method. The basic building block is the square function $\mathrm{sq}(y) = y^2$. If $e = 2^j$, we have

$$z^e = \left((z^2)^2 \cdots\right)^2 \quad (2 \text{ appears } j \text{ times}).$$

In other words, we have $z^e = \mathrm{sq}(\cdots \mathrm{sq}(z) \cdots)$, where sq is applied j times. If $x = x_0 + 2x_1 + \cdots + 2^{k-1}x_k$ with $x_j = 0, 1$,we can compute

$$z^x = z^{x_0 + 2x_1 + \cdots + 2^{k-1}x_k}$$

by computing z^{2^j} for each j such that $x_j = 1$ we can compute this, and multiply the results.

Overall, there would be k times of condition checking $x_j = 1$, at worst $1+2+\cdots+(k-1) = \frac{(k-1)k}{2}$ times of repeated squares, and $(k-1)$ times of product operation to collect the result. Of course, we need to translate these operations for bit strings to quantum gates that does the same for the computational basis. See [RP11, Sections 6.3 and 6.4] for how to achieve this.

17.2. **Fourier transform.** Reference: [Aar18, Section 20.1.1]

When $Q = 2$ (hence $k = 1$), the Fourier transform F_2 on $\mathbb{C}^2$ agrees with H:

$$F_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} \exp(\pi i 0 \cdot 0) & \exp(\pi i 0 \cdot 1) \\ \exp(\pi i 1 \cdot 0) & \exp(\pi i 1 \cdot 1) \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

We are going to establish a formula relating F_Q with $F_{Q/2}$. This will give a recursive definition formula for F_Q, of step $\log_2 Q = k$. If each step involves Ck^2 steps of basic operations, we can conclude that Shor's algorithm has circuit complexity of order k^3. Since $Q = 2^k$ is of order N^2, overall the circuit complexity will be of order $(\log N)^3$.

To have a cleaner formula for recursion, instead of the usual Fourier transform matrix

$$\left[\frac{1}{\sqrt{Q}} \exp\left(\frac{2\pi i}{Q} jl \right) \right]_{j,l=0}^{Q-1},$$

we consider the modified ones

$$[F_Q]_{j,l} = \frac{1}{\sqrt{Q}} \exp\left(\frac{2\pi i}{Q} j\tilde{l} \right), \tag{17.1}$$

where we modify the input index $0 \le l < Q$ to another integer $0 \le \tilde{l} < Q$ by the rule $l = 2^{k-1}x_0 + \cdots + x_{k-1}$ if and only if $\tilde{l} = x_0 + 2x_1 + \cdots + 2^{k-1}x_{k-1}$. (This means that the input binary sequence $x_0 \cdots x_{k-1}$ in the quantum gate F_Q should be interpreted as the integer $\tilde{l}$.)

Let us also consider the diagonal matrix of size $Q/2 = 2^{k-1}$ of the following form:

$$B_{Q/2} = \begin{bmatrix} 1 & 0 & \cdots & \\ 0 & \exp\left(\frac{2\pi i}{Q}\right) & 0 & \cdots \\ \vdots & & \ddots & \\ 0 & \cdots & 0 & \exp\left((2^{k-1}-1)\frac{2\pi i}{Q}\right) \end{bmatrix}.$$

Proposition 17.1. *We have*

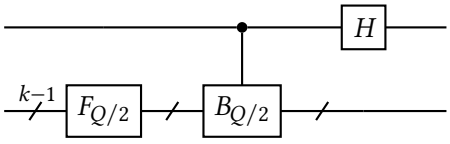
$$F_Q = \frac{1}{\sqrt{2}} \begin{bmatrix} F_{Q/2} & B_{Q/2}F_{Q/2} \\ F_{Q/2} & -B_{Q/2}F_{Q/2} \end{bmatrix}.$$

We can use this to estimate the number of basic gates that make up an implementation of F_Q. For the block matrix part, observe

$$\frac{1}{\sqrt{2}} \begin{bmatrix} F_{Q/2} & B_{Q/2}F_{Q/2} \\ F_{Q/2} & -B_{Q/2}F_{Q/2} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} I_{Q/2} & I_{Q/2} \\ I_{Q/2} & -I_{Q/2} \end{bmatrix} \begin{bmatrix} I_{Q/2} & 0 \\ 0 & B_{Q/2} \end{bmatrix} \begin{bmatrix} F_{Q/2} & 0 \\ 0 & F_{Q/2} \end{bmatrix}.$$

On the left hand side, the block matrix of $I_{Q/2}$, together with the factor $\frac{1}{\sqrt{2}}$ is $H \otimes I_{Q/2}$. Similarly, the part involving $F_{Q/2}$ is $I_2 \otimes F_{Q/2}$, and the middle part can be considered controlled $B_{Q/2}$, where control bit is the first one (corresponding to x_0).

The claim of Proposition 17.1 means F_Q can be presented as



(17.2)

If we look at the effect of this circuit on $|0x_1 \cdots x_{k-1}\rangle$, first $F_{Q/2}$ acts on $|x_1 \cdots x_{k-1}\rangle$ to give $v = F_{Q/2}|x_1 \cdots x_{k-1}\rangle$, where v is the l-th column of $F_{Q/2}$ for $l = x_1 + 2x_2 + \cdots + 2^{k-2}x_{k-1}$. Then the top / 0-th qubit being $|0\rangle$, then we apply the Hadamard gate on the top / 0-th qubit to get

$$|+\rangle \otimes v = \frac{1}{\sqrt{2}}(|0\rangle \otimes v + |1\rangle \otimes v)$$

The output index of F_Q is compatible with the Kronecker product, so the column vectors

$$\begin{bmatrix} v \\ 0 \end{bmatrix}, \quad \begin{bmatrix} 0 \\ v \end{bmatrix}$$

represent $|0\rangle \otimes v$ and $|1\rangle \otimes v$. Then we have

$$\frac{1}{\sqrt{2}} \begin{bmatrix} v \\ v \end{bmatrix} = |+\rangle \otimes v,$$

and we see that the effect of the above quantum circuit on $|0x_1 \cdots x_{k-1}\rangle$ represents the l-th column of the right hand side of Proposition 17.1. The proposition then says this vector is the l-th column of $F_Q S_k$. But this is exactly the l'-th column of F_Q, where $l' = 2l = 0 + 2x_1 + \cdots + 2^{k-1}x_{k-1}$. On the other hand, $F_Q|0x_1 \cdots x_{k-1}\rangle$ is also the l'-th column of F_Q.

We can compare the effect of the above circuit on $|1x_1 \cdots x_{k-1}\rangle$, and see that it agrees with the effect of F_Q on $|1x_1 \cdots x_{k-1}\rangle$.

Proof of Proposition 17.1. We compare the effect of both sides on the computational basis vector $|l\rangle$ for $0 \le l < Q$. (The corresponding bit string $x_0 \cdots x_{k-1}$ satisfies $l = 2^{k-1}x_0 + \cdots x_{k-1}$.) This means picking up the l-th column of these matrices. We need to analyze two different cases, depending on the input index l being small $0 \le l < Q/2$ or large $Q/2 \le l < Q$.

When l is small, we have $x_0 = 0$. This means $\tilde{l} = x_0 + \cdots + 2^{k-1}x_{k-1}$ is an even integer, and the component (17.1) can appear in $F_{Q/2}$. More precisely, $a = \tilde{l}/2$ is given by $a = x_1 + 2x_2 + \cdots + 2^{k-2}x_{k-1}$, and if j is smaller than $Q/2$,

$$[F_Q]_{j,l} = \frac{1}{\sqrt{Q}} \exp\left(\frac{2\pi i}{Q} j\tilde{l}\right) = \frac{1}{\sqrt{Q}} \exp\left(\frac{2\pi i}{Q/2} ja\right)$$

is, up to a factor of $\sqrt{2}$ for correcting $\sqrt{Q/2}$ to $\sqrt{Q}$, equal to $[F_{Q/2}]_{j,l'}$ for some l' such that $\tilde{l}' = a$ (here we should use the conversion rule for $(k-1)$-bit strings). Working out the conversion rule, we get $l' = 2^{k-2}x_1 + 2^{k-3}x_2 + \cdots + x_{k-1}$, which is equal to l by $x_0 = 0$. That is, we get $[F_Q]_{j,l} = [F_{Q/2}]_{j,l}$ if j and l are both small. If j is bigger than $Q/2$, we have

$$\frac{1}{\sqrt{Q}} \exp\left(\frac{2\pi i}{Q/2} ja\right) = \frac{1}{\sqrt{Q}} \exp\left(\frac{2\pi i}{Q/2}(j - Q/2)a\right) = [F_{Q/2}]_{j-Q/2,l},$$

and we see that the left half of F_Q is a block matrix of the form

$$\frac{1}{\sqrt{2}} \begin{bmatrix} F_{Q/2} \\ F_{Q/2} \end{bmatrix}.$$

When l is large, we have $x_0 = 1$. In this case $\tilde{l} = x_0 + 2x_1 + \cdots + 2^{k-1}x_{k-1}$ is an odd integer. Looking at the claim of the proposition, we want to check

$$\exp\left(\frac{2\pi i}{Q} j\tilde{l}\right) = \exp\left(\frac{2\pi i}{Q} j\right) \exp\left(\frac{2\pi i}{Q/2} j\tilde{l}'\right) \tag{17.3}$$

for the converted index $\tilde{l}'$ corresponding to $l' = l - Q/2$ (as $(k-1)$-bit strings). Indeed, if j is smaller than $Q/2$, then the factor $\exp\left(\frac{2\pi i}{Q} j\right)$ is equal to the j-th diagonal entry of $B_{Q/2}$. If we have $Q/2 \le j < Q$, this factor is equal to $-[B_{Q/2}]_{j'j'}$ for $j' = j - Q/2$, since we have $\exp\left(\frac{2\pi i}{Q} Q/2\right) = \exp(\pi i) = -1$.

Using $l = 2^{k-1}x_0 + \cdots + x_{k-1}$, the integer $l' = l - Q/2$ is represented by the $(k-1)$-bit string $x_1 x_2 \cdots x_{k-1}$, and we get $\tilde{l}' = x_1 + 2x_2 + \cdots + 2^{k-2}x_{k-1}$. We then have $\tilde{l} = 1 + 2\tilde{l}'$, and expanding $\tilde{l}$ in this way on the left hand side of (17.3), we get the right hand side. □

17.3. **Estimate of circuit complexity.** We now have enough material to get an estimate of circuit complexity (number of basic gates used to achieve the quantum circuit) for Shor's algorithm. Recall that we have $Q \sim N^2$ for the parameter $N = pq$. We claim that it has circuit complexity (at worst) $O((\log N)^2)$, which gives an exponential speedup over the known classical algorithms which have complexity $O(N^C)$ for some $C > 1$. (To be precise, one can have it about $\exp(C(\log N)^{\frac{1}{3}}(\log\log N)^{\frac{2}{3}})$.)

The quantum parallelism part is achieved by $H^{\otimes k}$, hence of complexity $k = O(\log N)$.

The XOR query gate has Ck^2 basic logical gates for some constant C (that goes into implementing the square function $\mathrm{sq}(y) \bmod N$), hence has complexity $O((\log N)^2)$.

We claim that F_Q has about $C'k^2$ basic gates. To be precise, the basic gates are the Hadamard gate H and controlled-$D(1, e^{iy})$ gate with

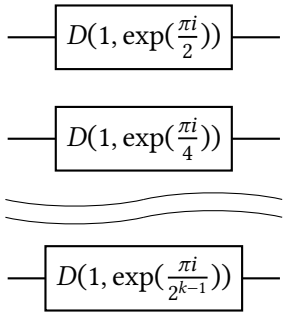
$$D(1, e^{iy}) = \begin{bmatrix} 1 & 0 \\ 0 & e^{iy} \end{bmatrix}.$$

First, using the substitution of $F_{Q/2^a}$ repeatedly using the diagram (17.2), we see that F_Q can be represented by k copies of H and the controlled-$B_{Q/2^a}$-gates.

It remains to check that the gate $B_{Q/2^a} = B_{2^{k-a}}$ can be implemented by $k - a$ times of $D(1, e^{iy})$ gates (then the controlled-$B_{Q/2^a}$ gates would be a product of controlled-$D(1, e^{iy})$ gates). Then it's enough to work out the case of $a = 1$. The gate B_Q transforms the computational basis $|x_1 \cdots x_{k-1}\rangle$ to $\alpha|x_1 \cdots x_{k-1}\rangle$, where $\alpha = \exp(j\frac{2\pi i}{Q})$ with $j = 2^{k-2}x_1 + 2^{k-3}x_2 + \cdots + x_{k-1}$. In other words, α is given by

$$\alpha = \exp\left(2^{k-2}x_1 \frac{2\pi i}{Q}\right) \exp\left(2^{k-3}x_2 \frac{2\pi i}{Q}\right) \cdots \exp\left(x_{k-1} \frac{2\pi i}{Q}\right).$$

As a quantum circuit, $B_{Q/2}$ is represented by the k-qubit circuit



.

17.4. **Computational note for Lecture 17.** Recursive relation for the Fourier transform (Proposition 17.1)

$$F_Q = \frac{1}{\sqrt{2}} \begin{bmatrix} F_{Q/2} & B_{Q/2} F_{Q/2} \\ F_{Q/2} & -B_{Q/2} F_{Q/2} \end{bmatrix},$$

with

$$[F_Q]_{j,l} = \frac{1}{\sqrt{Q}} \exp\left(\frac{2\pi i}{Q} jl\right),$$

where $\tilde{l}$ is the integer obtained by reordering the bit representation of l

[1]:
```
import sympy as sy
sy.init_printing(use_unicode=True)

from sympy.physics.quantum.dagger import Dagger
```

[2]:
```
k = 4
Q = 2**k
Q2 = 2**(k-1)
```

[3]:
```
# from Lecture 16
def bin_seq_for_int(k, x: int):
    """Returns binary sequence representation of the integer x.

    Parameters
    ==========
        x: int

    Returns
    =======
        tuple of int
          (x0, x1, ..., x(k-1)) with xi = 0, 1 and
          x = x0 + 2 * x1 + ... + 2**(k-1) * x(k-1)
    """
    # for i=0,..., k-1, do bit shift by i and bitwise AND with 1
    return tuple((x >> i) & 1 for i in range(k))


def int_from_bin_seq(k, x_seq):
    """Returns integer for the binary sequence of length k."""
    return sum(2**n * x_seq[n] for n in range(k))


# convert k-bit integer to another with reversed bit sequence
def converted_index(k, x):
    """Returns j = 2^{k-1} x_0 + 2^{k-2} x_1 + ... + x_{k-1}."""
    x_seq = bin_seq_for_int(k, x)
    j = sum(2**(k-1-n) * x_seq[n] for n in range(k))
    return j
```

[4]:
```
# from Lecture 16
def Fourier(n, j, l):
```

```
    """Returns the coefficient of Fourier transform matrix of size n."""
    return sy.exp(2 * sy.I * sy.pi * j * l / n) / sy.sqrt(n)
```

[5]:
```
F_Q = sy.Matrix([[Fourier(Q, j, converted_index(k, l))
                  for l in range(Q)] for j in range(Q)])
F_Q2 = sy.Matrix([[Fourier(Q2, j, converted_index(k-1, l))
                   for l in range(Q2)] for j in range(Q2)])
```

[6]:
```
F_Q
```

[6]:
$$\begin{bmatrix}
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\
\frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & -\frac{i}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{8}}}{4} & \frac{e^{-\frac{7i\pi}{8}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} & \frac{e^{-\frac{3i\pi}{8}}}{4} & \frac{e^{\frac{3i\pi}{8}}}{4} & \frac{e^{-\frac{5i\pi}{8}}}{4} & \frac{e^{\frac{7i\pi}{8}}}{4} & \frac{e^{-\frac{i\pi}{8}}}{4} \\
\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & \frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} \\
\frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{8}}}{4} & \frac{e^{-\frac{5i\pi}{8}}}{4} & \frac{e^{\frac{i\pi}{8}}}{4} & \frac{e^{-\frac{7i\pi}{8}}}{4} & \frac{e^{\frac{7i\pi}{8}}}{4} & \frac{e^{-\frac{i\pi}{8}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} & \frac{e^{-\frac{3i\pi}{8}}}{4} \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & \frac{i}{4} & \frac{i}{4} & \frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} \\
\frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & -\frac{i}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} & \frac{e^{-\frac{3i\pi}{8}}}{4} & \frac{e^{\frac{i\pi}{8}}}{4} & \frac{e^{-\frac{i\pi}{8}}}{4} & \frac{e^{\frac{7i\pi}{8}}}{4} & \frac{e^{-\frac{3i\pi}{8}}}{4} & \frac{e^{\frac{3i\pi}{8}}}{4} & \frac{e^{-\frac{5i\pi}{8}}}{4} \\
\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{i}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} \\
\frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{\frac{7i\pi}{8}}}{4} & \frac{e^{-\frac{i\pi}{8}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} & \frac{e^{-\frac{3i\pi}{8}}}{4} & \frac{e^{\frac{3i\pi}{8}}}{4} & \frac{e^{-\frac{5i\pi}{8}}}{4} & \frac{e^{\frac{i\pi}{8}}}{4} & \frac{e^{-\frac{7i\pi}{8}}}{4} \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\
\frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & -\frac{i}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{7i\pi}{8}}}{4} & \frac{e^{\frac{i\pi}{8}}}{4} & \frac{e^{-\frac{3i\pi}{8}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} & \frac{e^{-\frac{5i\pi}{8}}}{4} & \frac{e^{\frac{3i\pi}{8}}}{4} & \frac{e^{-\frac{i\pi}{8}}}{4} & \frac{e^{\frac{7i\pi}{8}}}{4} \\
\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & \frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} \\
\frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{5i\pi}{8}}}{4} & \frac{e^{\frac{3i\pi}{8}}}{4} & \frac{e^{-\frac{i\pi}{8}}}{4} & \frac{e^{\frac{7i\pi}{8}}}{4} & \frac{e^{-\frac{7i\pi}{8}}}{4} & \frac{e^{\frac{i\pi}{8}}}{4} & \frac{e^{-\frac{3i\pi}{8}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{i}{4} & \frac{i}{4} & \frac{i}{4} \\
\frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & -\frac{i}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{8}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} & \frac{e^{\frac{i\pi}{8}}}{4} & \frac{e^{-\frac{7i\pi}{8}}}{4} & \frac{e^{\frac{7i\pi}{8}}}{4} & \frac{e^{-\frac{5i\pi}{8}}}{4} & \frac{e^{\frac{3i\pi}{8}}}{4} & \frac{e^{-\frac{3i\pi}{8}}}{4} \\
\frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{i}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} \\
\frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{8}}}{4} & \frac{e^{\frac{i\pi}{8}}}{4} & \frac{e^{-\frac{7i\pi}{8}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} & \frac{e^{-\frac{3i\pi}{8}}}{4} & \frac{e^{\frac{3i\pi}{8}}}{4} & \frac{e^{-\frac{5i\pi}{8}}}{4} & \frac{e^{\frac{7i\pi}{8}}}{4} & \frac{e^{-\frac{i\pi}{8}}}{4}
\end{bmatrix}$$

[7]:
```
F_Q2
```

[7]:
$$\begin{bmatrix}
\frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} \\
\frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{4} & \frac{\sqrt{2}i}{4} & -\frac{\sqrt{2}i}{4} & \frac{\sqrt{2}e^{\frac{i\pi}{4}}}{4} & \frac{\sqrt{2}e^{-\frac{3i\pi}{4}}}{4} & \frac{\sqrt{2}e^{\frac{3i\pi}{4}}}{4} & \frac{\sqrt{2}e^{-\frac{i\pi}{4}}}{4} \\
\frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{4} & \frac{\sqrt{2}i}{4} & \frac{\sqrt{2}i}{4} & -\frac{\sqrt{2}i}{4} & -\frac{\sqrt{2}i}{4} \\
\frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{4} & -\frac{\sqrt{2}i}{4} & \frac{\sqrt{2}i}{4} & \frac{\sqrt{2}e^{\frac{3i\pi}{4}}}{4} & \frac{\sqrt{2}e^{-\frac{i\pi}{4}}}{4} & \frac{\sqrt{2}e^{\frac{i\pi}{4}}}{4} & \frac{\sqrt{2}e^{-\frac{3i\pi}{4}}}{4} \\
\frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{4} \\
\frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{4} & \frac{\sqrt{2}i}{4} & -\frac{\sqrt{2}i}{4} & \frac{\sqrt{2}e^{-\frac{3i\pi}{4}}}{4} & \frac{\sqrt{2}e^{\frac{i\pi}{4}}}{4} & \frac{\sqrt{2}e^{-\frac{i\pi}{4}}}{4} & \frac{\sqrt{2}e^{\frac{3i\pi}{4}}}{4} \\
\frac{\sqrt{2}}{4} & \frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{4} & -\frac{\sqrt{2}i}{4} & -\frac{\sqrt{2}i}{4} & \frac{\sqrt{2}i}{4} & \frac{\sqrt{2}i}{4} \\
\frac{\sqrt{2}}{4} & -\frac{\sqrt{2}}{4} & -\frac{\sqrt{2}i}{4} & \frac{\sqrt{2}i}{4} & \frac{\sqrt{2}e^{-\frac{i\pi}{4}}}{4} & \frac{\sqrt{2}e^{\frac{3i\pi}{4}}}{4} & \frac{\sqrt{2}e^{-\frac{3i\pi}{4}}}{4} & \frac{\sqrt{2}e^{\frac{i\pi}{4}}}{4}
\end{bmatrix}$$

$B_{Q/2}$ is a diagonal matrix of size $2^{k-1} = Q/2$, with components $1, \exp(\frac{2\pi i}{Q}), \exp(2\frac{2\pi i}{Q}), \cdots, \exp((2^{k-1}-1)\frac{2\pi i}{Q})$

[8]:
```
diag_list = [sy.exp(j * 2 * sy.pi * sy.I / Q) for j in range(2**(k-1))]
B_Q2 = sy.diag(*diag_list)
```

[9]:
```
B_Q2
```

[9]:
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & e^{\frac{i\pi}{8}} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & e^{\frac{i\pi}{4}} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & e^{\frac{3i\pi}{8}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & i & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & e^{\frac{5i\pi}{8}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & e^{\frac{3i\pi}{4}} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & e^{\frac{7i\pi}{8}} \end{bmatrix}$$

[10]:
```
A = sy.Matrix(sy.BlockMatrix([[F_Q2, B_Q2 * F_Q2],
                              [F_Q2, -B_Q2 * F_Q2]])) * (1/sy.sqrt(2))
```

[11]:
```
A
```

[11]:
$$\begin{bmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & -\frac{i}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{8}}}{4} & -\frac{e^{\frac{i\pi}{8}}}{4} & \frac{ie^{\frac{i\pi}{8}}}{4} & -\frac{ie^{\frac{i\pi}{8}}}{4} & \frac{e^{\frac{3i\pi}{8}}}{4} & \frac{e^{-\frac{5i\pi}{8}}}{4} & \frac{e^{\frac{7i\pi}{8}}}{4} & \frac{e^{-\frac{i\pi}{8}}}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & -\frac{e^{\frac{i\pi}{4}}}{4} & -\frac{e^{\frac{i\pi}{4}}}{4} & \frac{ie^{\frac{i\pi}{4}}}{4} & \frac{ie^{\frac{i\pi}{4}}}{4} & -\frac{ie^{\frac{i\pi}{4}}}{4} & -\frac{ie^{\frac{i\pi}{4}}}{4} & \frac{ie^{\frac{i\pi}{4}}}{4} \\ \frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{8}}}{4} & -\frac{e^{\frac{3i\pi}{8}}}{4} & -\frac{ie^{\frac{3i\pi}{8}}}{4} & \frac{ie^{\frac{3i\pi}{8}}}{4} & \frac{e^{-\frac{7i\pi}{8}}}{4} & \frac{e^{\frac{i\pi}{8}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} & \frac{e^{-\frac{3i\pi}{8}}}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & \frac{i}{4} & \frac{i}{4} & \frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} \\ \frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & -\frac{i}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} & -\frac{e^{\frac{5i\pi}{8}}}{4} & \frac{ie^{\frac{5i\pi}{8}}}{4} & -\frac{ie^{\frac{5i\pi}{8}}}{4} & \frac{e^{-\frac{i\pi}{8}}}{4} & \frac{e^{\frac{3i\pi}{8}}}{4} & \frac{e^{-\frac{5i\pi}{8}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{i}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & -\frac{e^{\frac{3i\pi}{4}}}{4} & -\frac{e^{\frac{3i\pi}{4}}}{4} & -\frac{ie^{\frac{3i\pi}{4}}}{4} & -\frac{ie^{\frac{3i\pi}{4}}}{4} & \frac{ie^{\frac{3i\pi}{4}}}{4} & \frac{ie^{\frac{3i\pi}{4}}}{4} \\ \frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{7i\pi}{8}}}{4} & -\frac{e^{\frac{7i\pi}{8}}}{4} & -\frac{ie^{\frac{7i\pi}{8}}}{4} & \frac{ie^{\frac{7i\pi}{8}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} & \frac{e^{-\frac{3i\pi}{8}}}{4} & \frac{e^{\frac{i\pi}{8}}}{4} & \frac{e^{-\frac{7i\pi}{8}}}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & -\frac{i}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & -\frac{e^{\frac{i\pi}{8}}}{4} & \frac{e^{\frac{i\pi}{8}}}{4} & -\frac{ie^{\frac{i\pi}{8}}}{4} & \frac{ie^{\frac{i\pi}{8}}}{4} & -\frac{e^{\frac{3i\pi}{8}}}{4} & -\frac{e^{-\frac{5i\pi}{8}}}{4} & -\frac{e^{\frac{7i\pi}{8}}}{4} & -\frac{e^{-\frac{i\pi}{8}}}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & \frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} & -\frac{e^{\frac{i\pi}{4}}}{4} & -\frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & -\frac{ie^{\frac{i\pi}{4}}}{4} & -\frac{ie^{\frac{i\pi}{4}}}{4} & \frac{ie^{\frac{i\pi}{4}}}{4} & \frac{ie^{\frac{i\pi}{4}}}{4} \\ \frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & -\frac{e^{\frac{3i\pi}{8}}}{4} & \frac{e^{\frac{3i\pi}{8}}}{4} & \frac{ie^{\frac{3i\pi}{8}}}{4} & -\frac{ie^{\frac{3i\pi}{8}}}{4} & -\frac{e^{\frac{7i\pi}{8}}}{4} & -\frac{e^{\frac{i\pi}{8}}}{4} & -\frac{e^{\frac{5i\pi}{8}}}{4} & -\frac{e^{-\frac{3i\pi}{8}}}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{i}{4} & \frac{i}{4} & \frac{i}{4} \\ \frac{1}{4} & -\frac{1}{4} & \frac{i}{4} & -\frac{i}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & -\frac{e^{\frac{5i\pi}{8}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} & -\frac{ie^{\frac{5i\pi}{8}}}{4} & \frac{ie^{\frac{5i\pi}{8}}}{4} & -\frac{e^{\frac{i\pi}{8}}}{4} & -\frac{e^{\frac{7i\pi}{8}}}{4} & -\frac{e^{\frac{3i\pi}{8}}}{4} & -\frac{e^{\frac{5i\pi}{8}}}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{i}{4} & -\frac{e^{\frac{3i\pi}{4}}}{4} & -\frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & \frac{ie^{\frac{3i\pi}{4}}}{4} & \frac{ie^{\frac{3i\pi}{4}}}{4} & -\frac{ie^{\frac{3i\pi}{4}}}{4} & -\frac{ie^{\frac{3i\pi}{4}}}{4} \\ \frac{1}{4} & -\frac{1}{4} & -\frac{i}{4} & \frac{i}{4} & \frac{e^{-\frac{i\pi}{4}}}{4} & \frac{e^{-\frac{3i\pi}{4}}}{4} & \frac{e^{\frac{i\pi}{4}}}{4} & \frac{e^{\frac{3i\pi}{4}}}{4} & -\frac{e^{\frac{7i\pi}{8}}}{4} & -\frac{e^{\frac{7i\pi}{8}}}{4} & \frac{ie^{\frac{7i\pi}{8}}}{4} & -\frac{ie^{\frac{7i\pi}{8}}}{4} & \frac{e^{\frac{5i\pi}{8}}}{4} & \frac{e^{-\frac{3i\pi}{8}}}{4} & -\frac{e^{\frac{i\pi}{8}}}{4} & -\frac{e^{-\frac{7i\pi}{8}}}{4} \end{bmatrix}$$

[12]:
```
sy.simplify(A) == sy.simplify(F_Q)
```

[12]:
```
True
```

Reference: [Aar18, Section 16.2]

18.1. **Universality in classical computation.** Classical computing deals with functions on strings of bits (Boolean functions $f(x)$ with inputs $x = x_1x_2\cdots x_k$ and output $0, 1$). There are many logic gates (basic operations on bits), for example

$$\mathrm{AND}(x,y) = \begin{cases} 1 & (x = y = 1), \\ 0 & \text{otherwise}, \end{cases} \qquad \mathrm{OR}(x,y) = \begin{cases} 0 & (x = y = 0), \\ 1 & \text{otherwise}, \end{cases} \qquad \mathrm{NOT}(x) = \begin{cases} 0 & (x = 1), \\ 1 & (x = 0). \end{cases}$$

Definition 18.1. We say that a collection $\{f_1, f_2, \dots, f_m\}$ of logic gates (each f_i would take certain number of inputs, like $f_1(x)$, $f_2(x,y,z)$, etc.) is *universal* if we can write *any* logic gate can be expressed as a composite of the f_i. We are allowed to *copy* input variables, like $f_2(x,x,y)$.

The most economic choice of universal collection is {NAND} for

$$\mathrm{NAND}(x,y) = \mathrm{NOT}(\mathrm{AND}(x,y)) = \begin{cases} 0 & (x = y = 1), \\ 1 & \text{otherwise}. \end{cases}$$

For example, we have $\mathrm{NOT}(x) = \mathrm{NAND}(x,x)$. Similarly, the constant function $\mathrm{FALSE}(x) = 0$ can be expressed as

$$\mathrm{FALSE}(x) = \mathrm{NAND}(x, \mathrm{NOT}(x)) = \mathrm{NAND}(x, \mathrm{NAND}(x,x)),$$

and OR can be expressed as

$$\mathrm{OR}(x,y) = \mathrm{NAND}(\mathrm{NOT}(x), \mathrm{NOT}(y)) = \mathrm{NAND}(\mathrm{NAND}(x,x), \mathrm{NAND}(y,y)),$$

and XOR as

$$\mathrm{XOR}(x,y) = \mathrm{AND}(\mathrm{OR}(x,y), \mathrm{NAND}(x,y))$$

with AND = NOT(NAND) and OR expressed in terms of NAND as above.

Remark 18.2. Arithmetic of single bits can be done with AND and XOR: $\mathrm{AND}(x,y)$ represents the product xy and $\mathrm{XOR}(x,y)$ represents the sum $x + y \mod 2$. Arithmetic of multi-bits becomes more complicated, but it is still possible to write down a combination of products and sums (modulo $N = 2^k$) for integers represented by k-bit strings ($x_0\cdots x_{k-1}$ representing $x = x_0 + 2x_1 + \cdots + 2^{k-1}x_{k-1}$, etc.). For example, see the Wikipedia article for the two-bit multiplication circuit.

As a variation, we can consider k-bit-to-k-bit *reversible* operations to make the setting closer to quantum computation. We would then consider transforms of sequences $f\colon \{0,1\}^n \to \{0,1\}^n$ that is reversible in the sense that any $y = y_1\cdots y_n$ is a value of f, i.e., there is (unique) $x \in \{0,1\}^n$ such that $f(x) = y$. Again a collection of such transforms $\{f^1, f^2, \cdots, f^k\}$ (each f_j defined on some n_j-bit string space) is universal if any reversible function $f\colon \{0,1\}^k \mapsto \{0,1\}^k$ can be written as some composition of the f^i. We can also encode non-reversible functions as reversible functions by adding extra bits. For example, if $f(x,y) = 0, 1$ is a Boolean function on two bits, we get an reversible three-bit transform f' by

$$f'(x,y,z) = (x, y, \mathrm{XOR}(z, f(x,y))),$$

the XOR query gate.

The *Toffoli* gate is one such gate. It can be considered as the controlled-controlled-NOT gate:

$$f_{\mathrm{Toffoli}}(x,y,z) = (x,y,w) \quad w = \mathrm{NOT}(z) \text{ if } x = y = 1,$$

and equivalently it is the XOR query gate for AND:

$$f_{\mathrm{Toffoli}}(x,y,z) = (x,y,\mathrm{XOR}(z, \mathrm{AND}(x,y))).$$

The NAND gate can be expressed as a component of the Toffoli gate:

$$(x, y, \mathrm{NAND}(x,y)) = f_{\mathrm{Toffoli}}(x,y,1),$$

hence $\{f_{\mathrm{Toffoli}}\}$ is again a universal collection of reversible gates.

Using similar ideas, it is possible (but takes some time) to implement arithmetic operations like $x \mapsto \mathrm{sq}(x) \mod N$ used in Shor's algorithm as reversible gates. See [RP11, Section 6.4] for the details.

18.2. **Universality in quantum computation.** We can consider a similar condition for collections of quantum gates in quantum computation, as follows.

Definition 18.3. A collection $\{U_1, U_2, \dots, U_m\}$ of quantum gates (each U_i is a unitary transform on some d_i-dimensional Hilbert space $\mathbb{C}^{d_i}$) is *universal* if any quantum gate on any finite-dimensional Hilbert space $\mathbb{C}^N$ can be *approximated* by products of the U_i up to taking tensor product with identity transform.

That is, we allow $I_a \otimes U_i \otimes I_b$ for various a and b satisfying $N = abd_i$ to get transforms on the N-dimensional space, and take their products. Here we don't ask for exact equality, but only approximation $V_k \to W\,(k \to \infty)$ for a given W on $\mathbb{C}^N$ in terms of the products V_k of the transforms of the form $I_a \otimes U_i \otimes I_b$. The reason is we expect *continuously many* quantum gates, like the rotation transforms R_θ on $\mathbb{C}^2$ for all possible angles θ. If we start from a finite collection of quantum gates, realizing all of them exactly by iterated products is a hopeless task.

There are several possible obstructions for a collection of quantum gates to be be universal.

- can it create superposition states like $|+\rangle$? If we stick to (reversible) logical gates, their effect on the computational basis vectors are always computational basis vectors. They cannot approximate the Hadamard gate H that does $H|0\rangle = |+\rangle$.
- can it create entangled states like the EPR pair state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$? The Hadamard gate composed with the CNOT gate transforms the separable state $|00\rangle$ to the EPR pair state. If the candidate collection for universality only contains single-qubit gates, separable states are mapped to separable states.
- can it create complex amplitudes? Matrix presentations of H and CNOT have real entries, so their compositions cannot approximate the ones with complex entries like the Pauli Y-gate.
- is your collection contained in the "stabilizer set" of some group of unitary transforms? This touches on the idea of *linear representations of groups* that goes beyond the scope of this lecture, but here is a concrete example (the Gottesman–Knill theorem[26]). Consider the collection of CNOT, H, and

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}.$$

These gates stabilize the collection of the Pauli gates up to sign in the following sense:

$$HXH^\dagger = Z, \quad HYH^\dagger = -Y, \quad HZH^\dagger = X, \quad SXS^\dagger = Y, \quad SYS^\dagger = -X, \quad SZS^\dagger = Z,$$

and

$$\begin{aligned} \mathrm{CNOT}(X \otimes I_2)\mathrm{CNOT} &= X \otimes X, & \mathrm{CNOT}(I_2 \otimes X)\mathrm{CNOT} &= I_2 \otimes X, \\ \mathrm{CNOT}(Y \otimes I_2)\mathrm{CNOT} &= Y \otimes X, & \mathrm{CNOT}(I_2 \otimes Y)\mathrm{CNOT} &= Z \otimes Y, \\ \mathrm{CNOT}(Z \otimes I_2)\mathrm{CNOT} &= Z \otimes I_2, & \mathrm{CNOT}(I_2 \otimes Z)\mathrm{CNOT} &= Z \otimes Z. \end{aligned}$$

One example of a universal collection of quantum gates is[27]

$$\left\{\mathrm{CNOT}, H, D(1, e^{i\frac{\pi}{8}}) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{8}} \end{bmatrix}\right\}.$$

Another choice is[28]

$$\{U_{\text{Toffoli}}, H, S\}$$

where U_{Toffoli} is the 3-qubit transform that acts as f_{Toffoli} on the computational basis.

---

[26] D. Gottesman, "The Heisenberg Representation of Quantum Computers". arXiv:quant-ph/9807006 (1998).

[27] P. O. Boykin, T. Mor, M. Pulver, V. Roychowdhury, and F. Vatan, "A new universal and fault-tolerant quantum basis", *Information Processing Letters* **75** (3), 101–107 (2000).

[28] Y. Shi, "Both Toffoli and controlled-NOT need little help to do universal quantum computation", *Quantum Information & Computation* **3** (1), 84–92 (2003).

Reference: [NC00, Section 4.5]

The starting point for searching a universal collection of quantum gates is to factorize a general unitary matrix into a product of simple ones,

$$U = U_1 \cdots U_d \quad (U \text{ is an } n\text{-by-}n \text{ unitary matrix}).$$

We can then try to approximate each U_i by composition of basic quantum gates like U_{Toffoli}, H, S, or CNOT as we discussed before.

19.1. **Factorization by two-level quantum gates.**

Definition 19.1. A *two-level quantum gate* is a unitary matrix V concentrated in two rows and columns except for the diagonal

$$V = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & a & & & & b & \\ & & & 1 & & & & \\ & & & & \ddots & & & \\ & & & & & 1 & & \\ & & c & & & & d & \\ & & & & & & & 1 \\ & & & & & & & & \ddots \end{bmatrix}.$$

The two-level quantum gates form a universal set[29], as follows.

Proposition 19.2. *Let U be a unitary matrix of size n. We can find unitary matrices $V^1, \ldots, V^k$ such that:*

- *each V_i is a two-level quantum gate; and*
- $U = V^1 \cdots V^k$.

As a variation, we can also have each V^i concentrated in two consecutive rows / columns, so that they look like

$$V^i = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & a & b & & \\ & & c & d & & \\ & & & & 1 & \\ & & & & & \ddots \end{bmatrix}.$$

Proof of Proposition 19.2. Taking the inverses of V^i, the claim is equivalent to finding two-level quantum gates $U^1, U^2, \ldots, U^k$ such that $U^k \cdots U^1 U = I_n$.

Our goal is to find matrices $U^1, U^2, \ldots$, so that $U^j \cdots U^1 U$ has first column of the form

$$\begin{bmatrix} * \\ 0 \\ \vdots \\ 0 \\ * \\ \vdots \end{bmatrix} \quad (0 \text{ appears } j \text{ times}).$$

After finding U^{n-1}, the first column would have all 0 except for the first entry. Since every row and column of a unitary matrix is a unit vector, this entry α should have absolute value 1, and then the first row would be all 0 except for this. By multiplying the diagonal matrix with entries α^*, 1, ..., 1 (which is a two-level matrix), we can assume $\alpha = 1$.

[29] M. Reck, A. Zeilinger, H. J. Bernstein, and P. Bertani, "Experimental realization of any discrete unitary operator", *Phys. Rev. Lett.* **73** (1), 58–61 (1994).

Then we have

$$U^{n-1}\cdots U^1 U = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & & & \\ \vdots & & U' & \\ 0 & & & \end{bmatrix}$$

for some $(n-1)$-by-$(n-1)$ unitary matrix U', and we can use the induction on n to get the claim.

The first matrix U^1 can be taken as follows. If the first column of U starts with a_0, c_0, then we can take

$$U_1 = \begin{bmatrix} a_0^*/r_0 & c_0^*/r_0 & 0 & \cdots & \\ c_0/r_0 & -a_0/r_0 & 0\cdots & & \\ 0 & 0 & 1 & 0 & \cdots \\ \vdots & \vdots & 0 & \ddots & \end{bmatrix}$$

where $r_0 = \sqrt{|a_0|^2 + |c_0|^2}$. Then, setting $U' = U_1 U$, we can take

$$U_2 = \begin{bmatrix} a_1^*/r_1 & 0 & c_1^*/r_1 & 0 & \cdots \\ 0 & 1 & 0 & 0 & \cdots \\ c_1/r_1 & 0 & -a_1/r_1 & 0 & \cdots \\ 0 & 0 & 0 & 1 & \cdots \end{bmatrix}$$

with $a_1 = u'_{11}$, $c_1 = u'_{31}$, and $r_1 = \sqrt{|a_1|^2 + |c_1|^2}$. Continuing this way, we get the desired matrices U^i. □

19.2. **Factorization by single-qubit gates and CNOT.** Building on the previous section, we can prove that the collection of CNOT and all of the single qubit gates is universal[30].

Theorem 19.3. *Let U be an N-qubit unitary gate, i.e., a unitary transform on $(\mathbb{C}^2)^{\otimes N} \simeq \mathbb{C}^{2^N}$. We can then find unitary gates $V^1, \dots, V^k$, satisfying $U = V^1 \cdots V^k$, and each V^i is either:*

- *single qubit gate acting on some qubit: $V = I_{2^a} \otimes W \otimes I_{2^{N-a-1}}$ for some 2-by-2-unitary matrix W; or*
- *CNOT on two qubits.*

Let us sketch the proof.

19.2.1. *Permutation of computational basis.* By Proposition 19.2, we can assume that U is a two-level quantum gate. In other words, U only affects two basis vectors $|s_0 \cdots s_{N-1}\rangle$ and $|t_0 \cdots t_{N-1}\rangle$ among the computational basis vectors $|x_0 \cdots x_{N-1}\rangle$, $x \in \{0,1\}^N$.

First, find a sequence of N-bit strings $g^0, \dots, g^m$ such that:

- $g^0 = s$ and $g^m = t$; while
- g^i and g^{i+1} differ at one position.

This allows us to construct an algorithm that factorizes U in terms of:

- controlled flip of bits; and
- controlled-W gate for some single qubit gate W.

The first kind, a controlled flip of bits (for indexes of computational basis) can exchange two N-bit strings g and g' which are different at one bit while fixing the other strings. For example, if $N = 3$ and the controlled bits are x_0 and x_2, with control conditions $x_0 = 0$ and $x_2 = 1$, then the corresponding controlled flip exchanges

$$g = (0,0,1), \quad g' = (0,1,1)$$

while fixing the other strings.

Repeating this for each step between g^i and g^{i+1}, we get a permutation P of computational basis that sends $|s\rangle = |g^0\rangle$ to $|g^{m-1}\rangle$ and each g^i to g^{i-1} for $0 < i < m$, and fixing the basis vectors $|x\rangle$ that do not appear among the $|g^i\rangle$. Let's say g^{m-1} and $g^m = t$ are different at the j-th bit. Then, PUP^{-1} is a special kind of two-level quantum gate: it acts on the j-th qubit (in the same way U acts on $|s\rangle$ and $|t\rangle$) whenever the other qubits are like those of t, and otherwise do nothing. In other words, this is a controlled-W gate, i.e., the second kind in the above list.

[30] D. P. DiVincenzo, "Two-bit gates are universal for quantum computation", *Phys. Rev. A* **51** (2), 1015–1022 (1995).

19.2.2. *Single-bit control.* Reference: [NC00, Section 4.3]

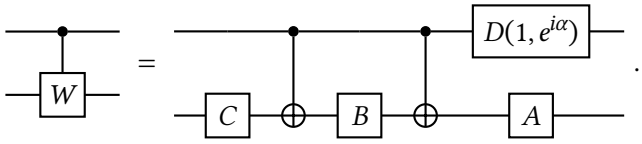
Now, suppose that we have a single-qubit gate W. We want to express the controlled-W gate by a composition of single-qubit gates and the CNOT gate. We are going to use the following factorization.

Proposition 19.4 ([NC00, Corollary 4.2])**.** *Given a 2-by-2 unitary matrix W, we can find three 2-by-2 unitary matrices A, B, C satisfying $ABC = I_2$ and*

$$W = e^{i\alpha} AXBXC,$$

for some real number α and the Pauli-X gate (flipping the computational basis $|0\rangle$ and $|1\rangle$).

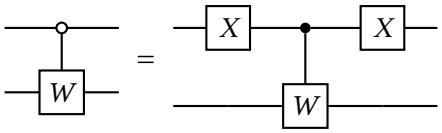
We can then check



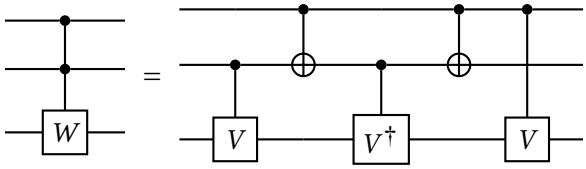
for the diagonal matrix (phase-shift matrix)

$$D(1, e^{i\alpha}) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{bmatrix}.$$

To change the control condition (apply W when the control bit is 0), we can change the control bit by X:

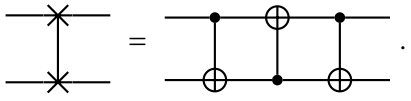


19.2.3. *Multi-bit control.* To implement control with multiple bits, we combine the controlled-W gate with CNOT gates. For example, for control by two bits (both conditioned to be 1), we have



for a unitary matrix V satisfying $V^2 = W$.

We can change the control condition by applying the X gate. We can also change where the control bits are by swapping the qubits, which can be implemented by composition of CNOT gates:



19.3. **Computational note for Lecture 19.**

[1]:
```
import sympy as sy
sy.init_printing(use_unicode=True)

from sympy.physics.quantum.dagger import Dagger
```

We want to decompose a given unitary matrix U as a product of two-level unitary matrices. We start by finding a unitary matrix U_1 such that U_1U has the first column of the form

$$\begin{bmatrix} u'_{11} \\ 0 \\ * \end{bmatrix}$$

If the first column of U starts with a, c, then we can take

$$U_1 = \begin{bmatrix} a^*/r & c^*/r & 0 & \cdots & \\ c/r & -a/r & 0 \cdots & & \\ 0 & 0 & 1 & 0 & \cdots \\ 0 & 0 & 0 & \ddots & \end{bmatrix}$$

where $r = \sqrt{|a|^2 + |c|^2}$.

[2]:
```
# Fourier transform matrix of size 3
U = sy.Matrix([[1, 1, 1],
               [1, sy.exp(2 * sy.I * sy.pi/3), sy.exp(-2 * sy.I * sy.pi/
↪3)],
               [1, sy.exp(-2 * sy.I * sy.pi/3), sy.exp(2 * sy.I * sy.pi/
↪3)]]) \
  * (1/sy.sqrt(3))
```

[3]:
```
a = U[0, 0]  # matrix index is 0-based in sympy
c = U[1, 0]
r = sy.sqrt(a * sy.conjugate(a) + c * sy.conjugate(c))
a, c, r
```

[3]:
$$\left(\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{6}}{3}\right)$$

[4]:
```
U1 = sy.Matrix([[sy.conjugate(a) / r, sy.conjugate(c) / r, 0],
                [c / r, -a / r, 0],
                [0, 0, 1]])
```

[5]:
```
U1 * U
```

[5]:
$$\begin{bmatrix} \frac{\sqrt{6}}{3} & \frac{\sqrt{6}}{6} + \frac{\sqrt{6}e^{\frac{2i\pi}{3}}}{6} & \frac{\sqrt{6}}{6} + \frac{\sqrt{6}e^{-\frac{2i\pi}{3}}}{6} \\ 0 & \frac{\sqrt{6}}{6} - \frac{\sqrt{6}e^{\frac{2i\pi}{3}}}{6} & \frac{\sqrt{6}}{6} - \frac{\sqrt{6}e^{-\frac{2i\pi}{3}}}{6} \\ \frac{\sqrt{3}}{3} & \frac{\sqrt{3}e^{-\frac{2i\pi}{3}}}{3} & \frac{\sqrt{3}e^{\frac{2i\pi}{3}}}{3} \end{bmatrix}$$

Then find U_2 such that U_2U_1U has the first column of the form

$$\begin{bmatrix} u''_{11} \\ 0 \\ 0 \\ * \end{bmatrix}$$

We can take

$$U_2 = \begin{bmatrix} a^*/r & 0 & c^*/r & 0 & \cdots \\ 0 & 1 & 0 & 0 & \cdots \\ c/r & 0 & -a/r & 0 & \cdots \\ 0 & 0 & 0 & 1 & \cdots \end{bmatrix}$$

with $a = u'_{11}$, $c = u'_{31}$, and $r = \sqrt{|a|^2 + |c|^2}$.

[6]:
```
Up = U1 * U
a = Up[0, 0]
c = Up[2, 0]
r = sy.sqrt(a * sy.conjugate(a) + c * sy.conjugate(c))
a, c, r
```

[6]: $\left(\frac{\sqrt{6}}{3}, \frac{\sqrt{3}}{3}, 1\right)$

[7]:
```
U2 = sy.Matrix([[sy.conjugate(a) / r, 0, sy.conjugate(c) / r],
                [0, 1, 0],
                [c / r, 0, -a / r]])
```

[8]:
```
U2 * U1 * U
```

[8]: $$\begin{bmatrix} 1 & \frac{1}{3} + \frac{e^{-\frac{2i\pi}{3}}}{3} + \frac{e^{\frac{2i\pi}{3}}}{3} & \frac{1}{3} + \frac{e^{-\frac{2i\pi}{3}}}{3} + \frac{e^{\frac{2i\pi}{3}}}{3} \\ 0 & \frac{\sqrt{6}}{6} - \frac{\sqrt{6}e^{\frac{2i\pi}{3}}}{6} & \frac{\sqrt{6}}{6} - \frac{\sqrt{6}e^{-\frac{2i\pi}{3}}}{6} \\ 0 & \frac{\sqrt{2}}{6} + \frac{\sqrt{2}e^{\frac{2i\pi}{3}}}{6} - \frac{\sqrt{2}e^{-\frac{2i\pi}{3}}}{3} & \frac{\sqrt{2}}{6} - \frac{\sqrt{2}e^{\frac{2i\pi}{3}}}{3} + \frac{\sqrt{2}e^{-\frac{2i\pi}{3}}}{6} \end{bmatrix}$$

[9]:
```
A = (U2 * U1 * U)
sy.Matrix([[A[j, k].rewrite(sy.sin).simplify()
            for k in range(3)] for j in range(3)])
```

[9]: $$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{\sqrt{6}}{4} - \frac{\sqrt{2}i}{4} & \frac{\sqrt{6}}{4} + \frac{\sqrt{2}i}{4} \\ 0 & \frac{\sqrt{2}}{4} + \frac{\sqrt{6}i}{4} & \frac{\sqrt{2}}{4} - \frac{\sqrt{6}i}{4} \end{bmatrix}$$

Reference: [NC00, Section A3]

If we want to understand some estimate for approximation of quantum gates involved in the universality problem, we need to have a way to quantify the distance of two matrices (or linear transforms). Like the distance between vectors, we first consider the *norm* (length) $\|A\|$ of such objects, which would allow us to define the distance by

$$d(A,B) = \|A - B\|.$$

There are many ways to make sense of the norm $\|A\|$. For example, we can treat an $m \times n$-matrix $A = [a_{ij}]_{i,j}$ as a vector of length mn, and use the Hilbert space norm (called the Hilbert–Schmidt norm of A)

$$\|A\|_{\mathrm{HS}} = \sqrt{\sum_{i,j} |a_{ij}|^2},$$

or other norms computed in terms of the components a_{ij}. Instead, we consider the *operator norm* of linear transform, defined as

$$\|T\| = \max_{v \in \mathbb{C}^d, \|v\|=1} \|Tv\|.$$

This satisfies some useful equalities like

$$\|T^\dagger T\| = \|T\|^2, \quad \|UTV\| = \|T\|$$

for arbitrary transforms T and unitary transforms U, V.

As the space of quantum gates, we will work with the *special unitary groups*, denoted by

$$\mathrm{SU}(d) = \{U \mid d \times d\text{-unitary matrix satisfying } \det U = 1.\}$$

The determinant $\det U$ is the product of all eigenvalues of U (counted with multiplicity). In general, if V is a $d \times d$-unitary matrix, it has d eigenvalues $\lambda_1, \dots, \lambda_d$ which satisfy $|\lambda_i| = 1$. We can then find a scalar μ satisfying $\mu^d = \prod_i \lambda_i$, and set $U = \mu^* V$ to get a unitary matrix in $\mathrm{SU}(d)$.

The Solovay–Kitaev theorem, independently proved by Solovay (unpublished) and Kitaev[31], gives a concrete estimate of approximation of general quantum gates in terms of product of basic ones.

Theorem 20.1 (The Solovay–Kitaev theorem)**.** *There are some positive constants C_1 and C_2 such that, for each $d = 2, 3, \dots$ we can find a positive constant $\epsilon = \epsilon_d$ with the following property: let S is some collection of matrices in $\mathrm{SU}(d)$ satisfying*

- *$V \in S \Rightarrow V^\dagger \in S$;*
- *for any $U \in \mathrm{SU}(d)$, there is some $V \in S$ such that $\|U - V\| < \epsilon$. (We say that S is ϵ-dense in $\mathrm{SU}(d)$)*

Then, for any $\delta > 0$ and $U \in \mathrm{SU}(d)$, there are some $k \leq C_1 |\log \delta|^{C_2}$ and $V_1, \dots, V_k \in S$ such that

$$\|U - V_1 \cdots V_k\| < \delta.$$

The constant C_2 can be taken as $\log 5 / \log(3/2) \sim 4$.

The main idea is to find $V \in S$ close to U, and write $V^\dagger U \sim I_d$ as an *commutator product* $V_1 W_1 V_1^\dagger W_1^\dagger$ for some $V_1, W_1 \in S$, closer to I_d than $V^\dagger U$. This would give an approximation of U by $V V_1 W_1 V_1^\dagger W_1^\dagger$. Repeating this process n times, we get an approximation of U by a product of 5^n-elements from S. We want to control the approximation error along the way to get an estimate of the form above.

There are two key technical facts about approximation of matrices.

Proposition 20.2. *Consider positive numbers $1 > \delta > \Delta > 0$ and $d \times d$-unitary matrices $U, V, \tilde{U}, \tilde{V}$ satisfying*

$$\|U - \tilde{U}\|, \|V - \tilde{V}\| < \Delta, \quad \|I_d - U\|, \|I_d - V\| < \delta.$$

We then have

$$\|UVU^\dagger V^\dagger - \tilde{U}\tilde{V}\tilde{U}^\dagger \tilde{V}^\dagger\| < 42\delta\Delta.$$

[31] A. Y. Kitaev. "Quantum computations: algorithms and error correction". Russ. Math. Surv., **52** (6), 1191–1249 (1997).

Proposition 20.3. *There is a constant $\delta(d)$ with the following property. If $U \in \mathrm{SU}(d)$ satisfies $\|I_d - U\| < \delta$ for some $\delta < \delta(d)$, we can find $V, W \in \mathrm{SU}(d)$ such that*

$$\|U - VWV^\dagger W^\dagger\| < 150\delta^{3/2}$$

holds.

Based on these, we can control the error at each step in the following way.

Proposition 20.4. *Suppose S is an ϵ-dense subset of $\mathrm{SU}(d)$ for ϵ smaller than $\delta(d)$ from Proposition 20.3. Then the subset*

$$S' = \{V_1V_2V_3V_4V_5 \mid V_i \in S\} \subset \mathrm{SU}(d)$$

is $276\epsilon^{3/2}$-dense.

Proof. We first find $V_1 \in S$ satisfying $\|U - V_1\| < \epsilon$. We then have $\|I_d - V_1^\dagger U\| < \epsilon$, so Proposition 20.3 says that we can then find $V, W \in \mathrm{SU}(d)$ such that

$$\|V_1^\dagger U - VWV^\dagger W^\dagger\| < 150\epsilon^{3/2}$$

holds, together with $\|I_d - V\|, \|I_d - W\| < 3\sqrt{\epsilon}$.

We next take $\tilde{V}, \tilde{W} \in S$ that are ϵ-close to V and W respectively. Using $\epsilon < 3\sqrt{\epsilon}$, from Proposition 20.2 we get

$$\|VWV^\dagger W^\dagger - \tilde{V}\tilde{W}\tilde{V}^\dagger\tilde{W}^\dagger\| < 42 \cdot 3\sqrt{\epsilon} \cdot \epsilon = 126\epsilon^{3/2}.$$

We then have

$$\|V_1^\dagger U - \tilde{V}\tilde{W}\tilde{V}^\dagger\tilde{W}^\dagger\| \leq \|V_1^\dagger U - VWV^\dagger W^\dagger\| + \|VWV^\dagger W^\dagger - \tilde{V}\tilde{W}\tilde{V}^\dagger\tilde{W}^\dagger\| < 276\epsilon^{3/2},$$

which implies that U is $276\epsilon^{3/2}$-close to $V_1V_2V_3V_4V_5$ for $V_2 = \tilde{V}$, $V_3 = \tilde{W}$, $V_4 = \tilde{V}^\dagger$, and $V_5 = \tilde{W}^\dagger$. □

Proof of Theorem 20.1. Let U be an arbitrary $d \times d$-unitary matrix. We will start by taking a parameter $\delta(d) > \delta_0 > \epsilon_d$ that will control the final error δ. We will think about the precise choice later, and how small ϵ_d should be also will follow from this condition.

Consider the sequence of subsets $S^{(0)}, S^{(1)}, \cdots$ by $S^{(0)} = S$ and $S^{(n+1)} = S^{(n)\prime}$, so that $S^{(n)}$ is the collection of matrix products of 5^n terms from S.

First, by Proposition 20.4, we know that $S^{(1)}$ is δ_1-dense for $\delta_1 = 276\delta_0^{3/2}$. Let's assume that δ_0 was small enough so that $\delta_1 < \delta_0$ holds. If we apply Proposition 20.4 again to $S^{(1)}$, we now know that $S^{(2)}$ is δ_2-dense for

$$\delta_2 = 276\delta_1^{3/2} = 276^{1+3/2}\delta_0^{(3/2)^2}.$$

The next step gives δ_3-density of $S^{(3)}$ for

$$\delta_3 = 276\delta_2^{3/2} = 276 \cdot (276^{1+3/2})^{3/2}(\delta_0^{(3/2)^2})^{3/2} = 276^{1+3/2+(3/2)^2}\delta_0^{(3/2)^3},$$

and so on, and after n steps we get δ_n-density of $S^{(n)}$ for

$$\delta_n = 276^{1+3/2+\cdots+(3/2)^{n-1}}(\delta_0)^{(3/2)^n}.$$

The exponent of 276 is

$$1 + \frac{3}{2} + \cdots \left(\frac{3}{2}\right)^{n-1} < 2\left(\frac{3}{2}\right)^n,$$

hence we have

$$\delta_n < 276^2(276\delta_0)^{(3/2)^n}.$$

Now, suppose $275\delta_0$ is small enough so that the right hand side is bounded by $\exp(-(3/2)^n)$ (that is, we first ask $276\delta_0 < e^{-1}$, and make δ_0 even smaller so that we can beat the leading term 276^2). Summarizing what we have so far, we have $k \sim 5^n$ terms from S and the error is $\delta \sim \exp(-(3/2)^n)$. In other words, $\log \delta^{-1} = -\log \delta$ is about $(3/2)^n$, hence we have

$$\log\log \delta^{-1} \sim \log\left((3/2)^n\right) = n\log(3/2).$$

Then $k \sim 5^n$ can be estimated as

$$\exp(n \log 5) \sim \exp\left(\log(\log \delta^{-1}) \frac{\log 5}{\log(3/2)}\right) = \exp(\log \log \delta^{-1})^{\frac{\log 5}{\log(3/2)}},$$

hence $k \sim (\log \delta^{-1})^{C_2}$ holds for $C_2 = \log 5 / \log(3/2)$. □

Propositions 20.2 and 20.3 are related to the *Baker–Campbell–Hausdorff formula*

$$\exp(X) \exp(Y) \exp(-X) \exp(-Y) \sim \exp([X, Y]) \quad (X, Y\text{: small } d \times d\text{-matrix})$$

for the *commutator bracket* $[X, Y] = XY - YX$. Here, if X is a small matrix ($\|X\|$ is small), then $\exp(X)$ is close to I_d. The precise estimate of the error depends on the matrix size d.

To get Proposition 20.3, we take a $d \times d$-matrix H satisfying $H^\dagger = -H$ (skew-Hermitian), $\operatorname{tr} H = 0$, and $U = \exp(H)$. Under the first two assumptions, we can find skew-Hermitian matrices X, Y satisfying

$$H = [X, Y], \quad \operatorname{tr}(X) = \operatorname{tr}(Y) = 0, \quad \|X\|, \|Y\| \leq 2\sqrt{\|H\|}.$$

We can then take $U = \exp(X)$ and $W = \exp(Y)$, and the error estimate follows from $\|H\| \sim \|I_d - U\| < \delta$ and a precise form of the BCH formula.

20.1. **Computational note for Lecture 20.**

[ ]:
```
import plotly.offline as pyo
import plotly.graph_objects as go
pyo.init_notebook_mode()

import numpy as np
np.set_printoptions(suppress=True, legacy='1.25')

from scipy.linalg import expm  # matrix exponential

import itertools as it
from functools import reduce
```

As the initial set S of generators, we take

$$V_1 = \exp(iX), \quad V_2 = \exp(iY), \quad V_3 = \exp(iZ)$$

for the Pauli matrices X, Y, Z, and their conjugates, and the identity matrix.

[3]:
```
T1 = np.array([[0, 1j], [1j, 0]])
V1 = expm(T1)
T2 = np.array([[0, -1], [1, 0]])
V2 = expm(T2)
T3 = np.array([[1j, 0], [0, -1j]])
V3 = expm(T3)
gen_list = [np.eye(2), V1, V1.conj().T, V2, V2.conj().T, V3, V3.conj().T]
```

We then set $S_k = \{V_{i_1} V_{i_2} \cdots V_{i_k} \mid V_{i_j} \in S\}$, the collection of product of k-terms from S

[4]:
```
S2 = [reduce(np.matmul, V_i_tup) for V_i_tup in it.product(gen_list,␣
 ↪repeat=2)]
S3 = [reduce(np.matmul, V_i_tup) for V_i_tup in it.product(gen_list,␣
 ↪repeat=3)]
S4 = [reduce(np.matmul, V_i_tup) for V_i_tup in it.product(gen_list,␣
 ↪repeat=4)]
S5 = [reduce(np.matmul, V_i_tup) for V_i_tup in it.product(gen_list,␣
 ↪repeat=5)]
```

We will represent qubit states v on the Bloch sphere. To do that, we compute the density operator $\rho = vv^\dagger$, and work out the coefficients x, y, z for

$$\rho = \frac{1}{2}(I_2 + xX + yY + zZ),$$

in terms of the Pauli matrices X, Y, Z (see Lecture 4). In this parametrization we would have $x^2 + y^2 + z^2 = 1$. We can compute them as

$$x = \rho_{01} + \rho_{10}, \quad y = i(\rho_{01} - \rho_{10}), \quad z = \rho_{00} - \rho_{11}.$$

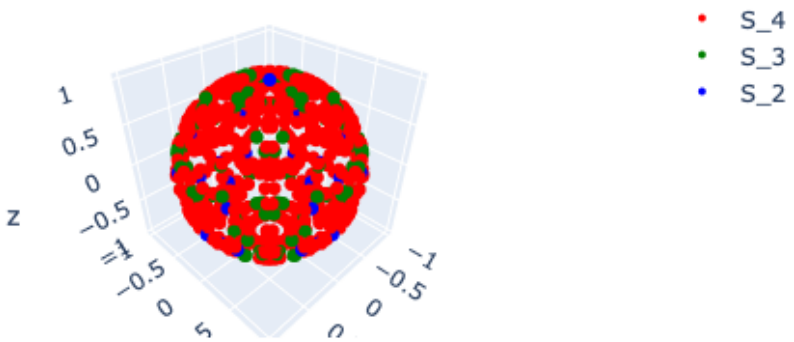
[5]:
```
def coord_from_state(v):
    """Returns 3D coordinates for a qubit state on the Bloch sphere."""
    rho = np.atleast_2d(v).conj().T @ np.atleast_2d(v)  # density operator
    x = np.real(rho[0, 1] + rho[1, 0])
    y = np.real(1j * (rho[0, 1] - rho[1, 0]))
```

```
    z = np.real(rho[0, 0] - rho[1, 1])
    return (x, y, z)
```

We compute the states $V|0\rangle$ for V in the collection S, and plot them on the Bloch sphere.

[6]:
```
zeroket = np.array([1, 0])
states_2 = [V @ zeroket for V in S2]
states_3 = [V @ zeroket for V in S3]
states_4 = [V @ zeroket for V in S4]
```

[7]:
```
# plot the states
mydata = []
coords = np.array([coord_from_state(v) for v in states_4])
mydata.append(go.Scatter3d(
    x=coords[:, 0],
    y=coords[:, 1],
    z=coords[:, 2],
    mode='markers',
    name='S_4',
    marker=dict(size=4,color='red')
))
coords = np.array([coord_from_state(v) for v in states_3])
mydata.append(go.Scatter3d(
    x=coords[:, 0],
    y=coords[:, 1],
    z=coords[:, 2],
    mode='markers',
    name='S_3',
    marker=dict(size=4,color='green')
))
coords = np.array([coord_from_state(v) for v in states_2])
mydata.append(go.Scatter3d(
    x=coords[:, 0],
    y=coords[:, 1],
    z=coords[:, 2],
    mode='markers',
    name='S_2',
    marker=dict(size=4,color='blue')
))

fig = go.Figure(data=mydata)
fig.show()
```



The distance between unitary matrices is given by $d(U,V) = \|U - V\|$, the operator norm of the difference. We have

$$\|U - V\|^2 = \|(U - V)^\dagger (U - V)\| = \|2I_d - U^\dagger V - V^\dagger U\|,$$

so we can compute $I_d - U^\dagger V$ and try to extract something close. For example, $\mathrm{Re}(\mathrm{tr}(I_d - U^\dagger V))$, the real part of the trace, would give the *trace norm* of $(U - V)^\dagger (U - V)$.

[8]:
```
def tr_dist_sq(U, V):
    """Returns the trace distance between 2x2 unitary matrices U and V."""
    return np.sqrt(np.real(np.trace(np.eye(2) - U.conj().T @ V)))
```

some random test unitary matrix, given as exponential of skew-Hermitian matrix

[9]:
```
test_mat = expm(np.array([[0.5j, -0.4 + 0.3j], [0.4 + 0.3j, -0.5j]]))
```

measure the minimal distance between this matrix and the ones from S_k

[10]:
```
def dist_from_set(mat_set):
    return min(tr_dist_sq(test_mat, sample_mat) for sample_mat in mat_set)
```

[11]:
```
dist_from_set(S2), dist_from_set(S3), dist_from_set(S4), dist_from_set(S5)
```

[11]: (0.566416404740649, 0.317731489690042, 0.257689141154173, 0.150354361988005)

Reference: [NC00, Section 6.1; Aar18, Section 22]

Grover's algorithm[32] gives a quadratic speedup for general *search problems*. To be precise, we set up the problem in the following way: Let $f\colon \{0,1\}^k \to \{0,1\}$ be a Boolean function with k-bit input (which we call the *oracle function*). Suppose that there is a unique input value x^* (which we call the *marked item*) such that $f(x^*) = 1$ happens. The problem is to find x^*.

Classically, we need $N = 2^k$ steps (at worst) to solve this problem, by asking for the value of $f(x)$ for each k-bit string x. Grover's algorithm goes in the following way.

(1) start with the initial state $|00\cdots 0\rangle \in (\mathbb{C}^2)^{\otimes k}$;
(2) apply the Hadamard gate on each qubit to get quantum parallelism, call the state v_0;
(3) repeat the following, starting with $j = 0$ and stop at around $j \sim \sqrt{N}\frac{\pi}{4}$:
   (a) apply the *phase query* gate $V_f|x\rangle = (-1)^{f(x)}|x\rangle$ ($x \in \{0,1\}^k$) to v_j, call the result w_j;
   (b) apply the *diffusion operator*

$$D|x\rangle = \left(\frac{1}{2^{k-1}} \sum_{y\in\{0,1\}^k} |y\rangle\right) - |x\rangle$$

   to w_j, call the result v_{j+1};
   (c) move to the next step.
(4) measure the state v_j in the computational basis.

Then the result agrees with x^* with high probability.

First let's check that the diffusion operator is a quantum gate.

Proposition 21.1. *The operator D is a unitary transform on $(\mathbb{C}^2)^{\otimes k}$.*

Proof. The matrix presentation of D is

$$\begin{bmatrix} \frac{2}{N}-1 & \frac{2}{N} & \frac{2}{N} & \cdots \\ \frac{2}{N} & \frac{2}{N}-1 & \frac{2}{N} & \cdots \\ \frac{2}{N} & \frac{2}{N} & \ddots & \\ \vdots & \vdots & & \end{bmatrix},$$

hence D is selfadjoint. We then want to check $D^2 = I_N$.

If we consider the input vector $v = \sum_{x\in\{0,1\}^k} \alpha_x |x\rangle$, the effect of D can be computed as

$$D(v) = 2\frac{\sum_x \alpha_x}{N} \sum_y |y\rangle - \sum_x \alpha_x |x\rangle.$$

This means that the coefficient of $|x\rangle$ is transformed as

$$\alpha_x \rightsquigarrow \beta_x = 2\bar{\alpha} - \alpha_x,$$

where $\bar{\alpha}$ is the average of the α_y for $y \in \{0,1\}^k$. The average of the new coefficients β_x satisfy $\bar{\beta} = \bar{\alpha}$, and we have

$$2\bar{\beta} - \beta_x = 2\bar{\alpha} - (2\bar{\alpha} - \alpha_x) = \alpha_x,$$

which shows $D^2 = I_N$. □

Let's analyze how this algorithm works.

- the state v_0 is the quantum parallelism vector $\frac{1}{\sqrt{N}}\sum_x |x\rangle$;
- applying V_f, we flip the sign of the coefficient of $|x^*\rangle$, while keeping the others. So we have $w_0 = \frac{1}{\sqrt{N}}(\sum_{x\neq x^*} |x\rangle - |x^*\rangle)$;

[32] L. Grover, "A fast quantum mechanical algorithm for database search", Proceedings of the 28th Annual ACM Symposium on Theory of Computing, 212–219 (1996).

- the average coefficient of w_0 is

$$\frac{1}{N}\left((N-1)\times\frac{1}{\sqrt{N}}+1\times\frac{-1}{\sqrt{N}}\right)=\frac{N-2}{N^{3/2}}.$$

  Then the result of D is

$$v_1=\sum_{x\neq x^*}\left(\frac{2(N-2)}{N^{3/2}}-\frac{1}{\sqrt{N}}\right)|x\rangle+\left(\frac{2(N-2)}{N^{3/2}}+\frac{1}{\sqrt{N}}\right)|x^*\rangle\,.$$

  Now $|x^*\rangle$ has a bit bigger coefficient than the others.
- applying V_f again, we flip the sign of this bigger coefficient. Now the average becomes smaller.
- applying D next, the coefficient of x^* becomes $\bar{\alpha}-\alpha_{x^*}$ which is bigger than $\bar{\alpha}$ (by $\alpha_{x^*}<0$) while the other ones will get even smaller coefficient (by $\alpha_x>0$).

This way, we see that the coefficient of $|x^*\rangle$ keeps getting bigger while the other ones α_x keep getting smaller, *as long as* $\alpha_x>0$ *holds*. We need to stop before α_x becomes negative, because the process $\alpha_x\rightsquigarrow 2\bar{\alpha}-\alpha_x$ would start to roll back then.

Next we need to analyze the stopping time $j_{\text{stop}}\sim\sqrt{N}\frac{\pi}{4}$. We note that the intermediate state vectors v_j and w_j are all real linear combinations of $v_0=\frac{1}{\sqrt{N}}\sum_x|x\rangle$ and $|x^*\rangle$. This means we can look at the (real) 2-dimensional span of these vectors. As an orthogonal basis we can take $v'=\frac{1}{\sqrt{N-1}}\sum_{x\neq x^*}|x\rangle$ and $|x^*\rangle$.

The important viewpoint is to recognize the operators V_f and D as *reflections*:

- V_f is the reflection along the span of v'; while
- D is the reflection along the span of v_0;

Then DV_f is a rotation by angle 2θ, where θ satisfies

$$v_0=\cos\theta v'+\sin\theta|x^*\rangle\,,$$

see Figure 21.1. When $2j\theta\sim\frac{\pi}{2}$, we have $(DV_f)^jv_0\sim|x^*\rangle$, hence v_j is close to the desired state $|x^*\rangle$. Since v_0 contains $|x^*\rangle$ with coefficient $\frac{1}{\sqrt{N}}$, we have $\sin\theta=\frac{1}{\sqrt{N}}$. Moreover, for small θ we have $\theta\sim\sin\theta$, so the stop step j_{stop} should satisfy $2j_{\text{stop}}\frac{1}{\sqrt{N}}\sim\frac{\pi}{2}$, and we get the desired condition.

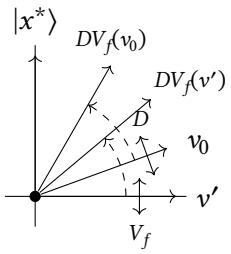
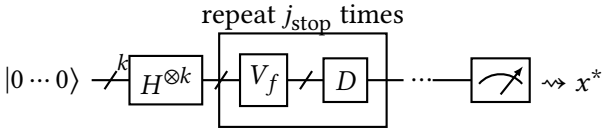


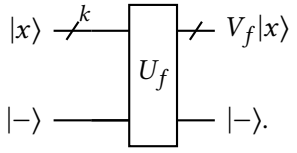
FIGURE 21.1. rotation from reflections

21.1. **Quantum circuit for Grover's algorithm.** Overall, Grover's algorithm is given by an iteration of the composite gate DV_f on $|+\cdots+\rangle=H^{\otimes k}|0\cdots0\rangle$, so it can be represented as



for $j_{\text{stop}}\sim\sqrt{N}\frac{\pi}{4}$.

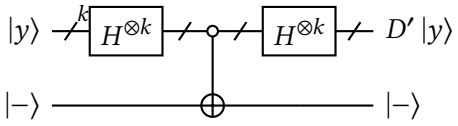
As we saw in Remark 14.1, the phase query V_f can be obtained from the XOR query U_f by adding one extra qubit, as



The diffusion operator D satisfies

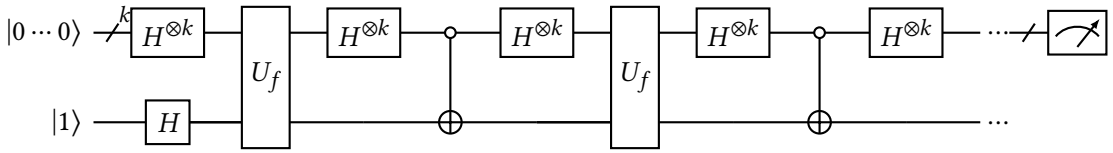
$$D|+\cdots+\rangle = |+\cdots+\rangle\,, \quad D|y_1\cdots y_k\rangle = -|y_1\cdots y_k\rangle$$

for any k-letter string y of $+$ and $-$ that contains at least one $-$. It's better if we consider the operator $D' = -D$, which does not affect the measurement outcome (because the overall states is multiplied by $(-1)^j$ after j steps). Then, D' checks if the input is labelled by $+\cdots+$, and applies "-1 times" operation in that case, otherwise do nothing. Combining this with $H^{\otimes k}|+\cdots+\rangle = |0\cdots0\rangle$ and $H^{\otimes k}|0\cdots0\rangle = |+\cdots+\rangle$, we get a presentation of D' with an auxiliary qubit:



where, in the middle, we have the controlled-NOT gate with control condition $x = 0\cdots0$ for k-bit x.

Overall, we have a $(k+1)$-qubit circuit of the form



21.2. **Variation: multiple marked inputs.** Suppose that there are multiple marked inputs (solutions to $f(x) = 1$) that we call $x^{(1)}, \dots, x^{(c)} \in \{0,1\}^k$. Let us use the same algorithm as above, with the phase query gate V_f and the diffusion operator D. This time, after getting the quantum parallelism vector $v_0 = |+\cdots+\rangle$, the effect of V_f becomes

$$w_0 = \frac{1}{\sqrt{N}}\left(\sum_{x\in\mathcal{U}} |x\rangle - \sum_{i=1}^{c} |x^{(i)}\rangle\right),$$

where $\mathcal{U}$ is the collection of unmarked strings,

$$\mathcal{U} = \{x \in \{0,1\}^k \mid x \neq x^{(i)} \text{ for } i = 1,\dots,c\}.$$

Then D will give the difference $2\bar{a} - a_x$ with the average coefficient

$$\bar{a} = \frac{1}{N}\left((N-c)\frac{1}{\sqrt{N}} - c\frac{1}{\sqrt{N}}\right) = \frac{N-2c}{N^{3/2}}$$

and the original coefficients $a_x = \frac{1}{\sqrt{N}}$ (for unmarked inputs) or $-\frac{1}{\sqrt{N}}$ (for marked inputs), so that we have

$$v_1 = \sum_{x\in\mathcal{U}}\left(\frac{2(N-2c)}{N^{3/2}} - \frac{1}{\sqrt{N}}\right)|x\rangle + \sum_{i=1}^{c}\left(\frac{2(N-2c)}{N^{3/2}} + \frac{1}{\sqrt{N}}\right)|x^{(i)}\rangle\,.$$

Again the state vectors $v_0, v_1, \dots$ live in the real 2-dimensional span of

$$v_m = \frac{1}{\sqrt{c}}\sum_{i=1}^{c}|x^{(i)}\rangle\,, \quad v_u = \frac{1}{\sqrt{N-c}}\sum_{x\in\mathcal{U}}|x\rangle\,. \tag{21.1}$$

(Before $|x^*\rangle$ played the role of v_m and v' played the role of v_u.)

The diffusion operator D is still the reflection along the line spanned by v_0 (since it fixes v_0). The phase query V_f becomes the reflection along the line spanned by v_u (since it fixes v_u). Then DV_f is the rotation of angle 2θ, where θ is characterized by

$$v_0 = (\cos\theta)v_u + (\sin\theta)v_m \Leftrightarrow \sin\theta = \sqrt{\frac{c}{N}}.$$

This means that the good stopping step is $j_{\text{stop}} \sim \frac{\pi}{4}\sqrt{\frac{N}{c}}$. If we know the number c, we also have an estimate for j_{stop}.

What if c is unknown? Let us write $v_j = (DV_f)^j v_0$ as

$$v_j = \alpha(j)v_m + \beta(j)v_u.$$

Since DV_f is the rotation of angle 2θ (for $\theta \sim \sin\theta = \sqrt{\frac{c}{N}}$ that we don't know), we have $\alpha(j) = \sin(2j\theta)$.

If we measure v_j in the computational basis and check if the measurement outcome x satisfies $f(x) = 1$, the success cases $x^{(i)}$ happen with probability $p(i) = \left|\langle v^{(i)}, v_j\rangle\right|^2$ for each i. From (21.1), we have $\langle v^{(i)}, v_m\rangle = \frac{1}{\sqrt{c}}$, so that $\langle v^{(i)}, v_j\rangle = \alpha(j)\frac{1}{\sqrt{c}}$, and the overall success probability is

$$p(1) + \cdots + p(c) = c \cdot \alpha(j)^2 \frac{1}{c} = \sin^2(2j\theta).$$

If we *randomly choose* j between 1 and some fixed big integer M, the success probability will be

$$\frac{1}{M}\sum_{j=1}^{M} \sin^2(2j\theta).$$

This is almost the integral of $\sin^2(Tx)$ from $x = 0$ to 1, for $T = 2M\theta \sim 2M\sqrt{\frac{c}{N}}$. We also have

$$\int_0^1 \sin^2(Tx)dx \sim \frac{1}{2},$$

since a small change of x in the integrand (by $\frac{\pi}{2T}$) would give $\cos^2(Tx) = 1 - \sin^2(Tx)$. This means that a random choice of j and the measurement of v_j in the computational basis would give x that satisfies $f(x) = 1$ with probability $\frac{1}{2}$. (Does this sound paradoxical?)

22. The Harrow–Hassidim–Lloyd algorithm

Linear equations

$$Ax = b \tag{22.1}$$

for an $M \times N$-matrix A and a target vector $b \in \mathbb{C}^M$, and we look for solutions $x \in \mathbb{C}^N$, are ubiquitous in mathematical modeling of various practical and theoretical problems. A quantum algorithm by Harrow, Hassidim, and Lloyd[33] constructs a state vector approximating x out of a one representing b and quantum gates built from A. When A satisfies some conditions, the quantum gate has lower complexity than the standard classical algorithms, potentially offering an exponential speedup. (However, there are several caveats as we will review below, and in particular we cannot hope a good approximation of x for arbitrary target vectors b.) Let us review the main ideas behind this algorithm.

First, by rewriting the given equation (22.1) as

$$\begin{bmatrix} 0_{N\times N} & A^\dagger \\ A & 0_{M\times M} \end{bmatrix}\begin{bmatrix} x \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ b \end{bmatrix},$$

we can assume that A is an self-adjoint matrix.

[33] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations". *Phys. Rev. Lett.* **103** (15), 150502 (2008).

Let us say A is an *invertible* $N \times N$ self-adjoint matrix, and we want an approximation of the solution vector $x \in \mathbb{C}^N$ for (22.1) with error ϵ. We will also use the *condition number* $\kappa_A = \|A\|\|A^{-1}\|$ for A, which can be written as

$$\kappa_A = \frac{\sigma_{\max}(A)}{\sigma_{\min}(A)},$$

where $\sigma_{\max}(A)$ is the absolute value of the eigenvalue of A furthest from 0 and $\sigma_{\min}(A)$ is that of closest to 0 (maximal and minimal singular numbers). The algorithm will involve two numbers C (very small) and t_0 (very big), which will be of order $C = O(\kappa_A^{-1})$ and $t_0 = O(\kappa_A \epsilon^{-1})$.

The Hilbert space will be $\mathbb{C}^Q \otimes \mathbb{C}^N \otimes \mathbb{C}^2$ for some big enough Q. Here, Q can be a power of 2, like $Q = 2^k$. We will write the standard basis of $\mathbb{C}^Q$ ("computational basis") as $|j\rangle$ for $0 \le j < Q$. The algorithm works as follows.

(1) start with the vector $v_1 = u_0 \otimes b \otimes |0\rangle$, where $u_0 = \frac{1}{\sqrt{Q}} \sum_{0 \le j < Q} |j\rangle$;
(2) apply the quantum gate $\sum_j |j\rangle\langle j| \otimes \exp\left(-i\frac{t_0 j}{Q} A\right) \otimes I_2$;
(3) apply the Fourier transform to the first factor, i.e., $F_Q \otimes I_N \otimes I_2$;
(4) then apply the quantum gate $\sum_j |j\rangle\langle j| \otimes I_N \otimes R_{\theta_j}$, where R_θ is the rotation matrix

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

and the angle θ_j satisfies

$$\sin\theta_j = \begin{cases} \frac{Ct_0}{2\pi j} & (0 < j < \frac{Q}{2}) \\ \frac{Ct_0}{2\pi(j-Q)} & (\frac{Q}{2} \le j < Q) \\ 0 & (j = 0); \end{cases}$$

(5) then measure the last qubit in the computational basis, and if we have measured the case 1, the state will give an approximation of x (in the sense we explain below).

To be precise, the initial vector at step 1 is a unit vector exactly when b is a state vector. In general, given a linear equation (22.1), we should rescale the right hand side to a unit vector to fit into this quantum algorithm. Then the linear transform in step 2 is a unitary transform because each matrix exponential $\exp\left(-i\frac{t_0 j}{Q} A\right)$ is unitary (it is of the form $\exp(iH)$ for a self-adjoint operator H). In step 3, we use the Fourier transform matrix given by

$$[F_Q]_{l,j} = \frac{1}{\sqrt{Q}} \exp\left(\frac{2\pi i l j}{Q}\right).$$

For $0 < j < Q$, let us write $\tilde{j}$ for

$$\tilde{j} = \begin{cases} j & (0 < j < \frac{Q}{2}) \\ (j - Q) & (\frac{Q}{2} \le j < Q), \end{cases}$$

so that we have $j = \tilde{j} \bmod Q$ while we can approximate eigenvalues λ_i of A as

$$\lambda_i \sim \frac{2\pi\tilde{j}}{t_0}$$

for some $0 < j < Q$ that depends on i. To be able to approximate eigenvalues close to 0, t_0 has to be big of order κ_A. To have an ϵ-approximation, it needs to be of order ϵ^{-1}, hence we get $t_0 = O(\kappa_A \epsilon^{-1})$. Then, to have the formula for $\sin\theta_j$ to make sense, C has to be small so that the fractions $\frac{Ct_0}{2\pi\tilde{j}}$ have absolute value less than 1. This leads to $C = O(\kappa_A^{-1})$.

Let us keep track of the state vectors at each step. We take the eigenvectors u_i for A (satisfying $Au_i = \lambda_i u_i$) and write the vector b as

$$b = \sum_{i=1}^{N} \beta_i u_i \quad (\beta_i \in \mathbb{C}).$$

After step 2, we have

$$v_2 = \frac{1}{\sqrt{Q}} \sum_{j,i} |j\rangle \otimes \beta_i \exp\left(-i\frac{t_0 j}{Q}\lambda_i\right) u_i \otimes |0\rangle .$$

Then, after step 3, we have

$$v_3 = \frac{1}{Q} \sum_{l,j,i} \exp\left(\frac{2\pi i l j}{Q}\right) \beta_i \exp\left(-i\frac{t_0 j}{Q}\lambda_i\right) |l\rangle \otimes u_i \otimes |0\rangle .$$

Let's fix the index i. The coefficient (as a sequence for l) is the Fourier transform of the sequence $\frac{1}{\sqrt{Q}}\beta_i \exp\left(-i\frac{t_0 j \lambda_i}{Q}\right)$, so it is concentrated around l satisfying

$$t_0\lambda_i \sim 2\pi l \bmod 2\pi Q \Leftrightarrow l \sim \frac{t_0\lambda_i}{2\pi} \bmod Q \Leftrightarrow \tilde{l} \sim \frac{t_0\lambda_i}{2\pi}.$$

Then, step 4 gives $\cos\theta_l |0\rangle + \sin\theta_l |1\rangle$ in the last qubit. Now, the relevant part of the state is

$$v_4 = \sum_i \beta_i |l_i\rangle \otimes u_i \otimes (\cos\theta_{l_i} |0\rangle + \sin\theta_{l_i} |1\rangle),$$

where l_i is related to i as above. After having measured case 1 in step 5, the state v_5 will be (almost) proportional to

$$v_5' = \sum_i \beta_i \sin\theta_{l_i} |l_i\rangle \otimes u_i \otimes |1\rangle = \sum_i \beta_i \frac{C}{\lambda_i} |l_i\rangle \otimes u_i \otimes |1\rangle .$$

Ignoring the constant C (that will go away with normalization to a state vector), we have

$$v_5' = \sum_i |l_i\rangle \otimes \beta_i \lambda_i^{-1} u_i \otimes |1\rangle$$

Now, $x = \sum_i \beta_i \lambda_i^{-1} u_i$ is the solution of (22.1), because we have

$$Ax = \sum_i \beta_i \lambda_i^{-1} A u_i = \sum_i \beta_i \lambda_i^{-1} \lambda_i u_i = \sum_i \beta_i u_i = b.$$

This means that v_5 gives an approximation of the solution vector x. To extract information, we should take a PVM $(P_a)_a$ on $\mathbb{C}^N$, and perform a measurement with the PVM $P_a' = I_Q \otimes P_a \otimes I_2$ against the state vector v_5. Then the measurement outcome probability $\mathbb{P}[a] = \langle v_5, P_a' v_5\rangle$ is close to $\langle x', P_a x'\rangle$, where x' is the normalized state vector we get from the solution vector x.

Classical algorithms (based on the LU decomposition) take steps of order $O(N^2)$. When A is a sparse matrix, then the transform $\exp(-itA)$ for a fixed t can be implemented in steps of order $O(\log N)$, which leads to the hope of exponential speedup. The outstanding problems are the following[34]:

(1) the coefficients β_i has to be concentrated in a small number of indexes. In particular, we cannot allow b to be an arbitrary computational basis, because this would give a too-good speedup for general search problem;
(2) how fast can we prepare the initial state v_1? Note that b has N components, so naively setting each component one by one already defeats the exponential speedup;
(3) if we allow a class of matrices A for different N such that $\kappa_A = O(N^c)$ for some $c > 0$, step 2 will take too many time;
(4) getting precise information about x from the state v_5 also requires many measurements, because we only have a control for the probability $\mathbb{P}[a]$.

23. Epilogue: further topics

There are several important topics in quantum computation that we did not cover in this course, as well as the closely related field of *quantum information*. Let us quickly overview them.

[34] S. Aaronson, "Read the fine print". Nature Physics. **11** (4), 291–293 (2015).

23.1. **Error correction.** Reference: [NC00, Chapter 10; Aar18, Chapters 27 and 28]

We have to allow some margin of errors and noise in operations for quantum computation. For example, qubit states are unit vectors of the Hilbert space $\mathbb{C}^2$, or points on the Bloch sphere, which have continuous parameter. The initial states might be $|0\rangle + v_\epsilon$ for some small error vector v_ϵ, or the concrete implementation of the Hadamard gate might transform this to $|+\rangle + w_\delta$ with even bigger error vector w_δ.

In fact, classical computation also needs to deal with a similar problem, since it is ultimately implemented through physical quantities like electric and magnetic charges which come with noise. For example, if we have a string of bits $0110\ldots$ representing some data, after some manipulation some bits might flip $0 \leftrightarrow 1$ due to an physical error. One solution is to use a *repetition code*, for example use 3-bit sequences $\bar{0} = 000$ and $\bar{1} = 111$ to represent *logical bits*. We then operate on blocks of 3-bits, like

$$\mathrm{AND}(\bar{0}, \bar{1}) = \bar{0}, \quad \mathrm{OR}(\bar{0}, \bar{1}) = \bar{1},$$

and expect results of the form

$$\bar{0}\bar{1}\bar{0}\cdots = 000111000\ldots.$$

A physical error might introduce bit flips, making this sequence $000101000\ldots$. In order to detect and correct errors, we can take each block $x = x_0x_1x_2$ and compute the "majority vote", inferring $\bar{1}$ from 101, etc.

To achieve a similar error correction scheme in quantum computation, we need to be mindful of the following issues:

(1) as we remarked above, states come with continuous parameters;
(2) the no-cloning theorem says that we cannot "duplicate" arbitrary states, so a naive generalization of classical scheme is often not realizable;
(3) a measurement of state will collapse the state to one of finitely many possibilities, and can easily remove a possible benefit of the quantum algorithm.

Shor's 9-qubit code[35] is a careful adaptation of the above 3-bit repetition code to quantum computation. In this scheme, *logical qubits* are given by

$$|\bar{0}\rangle = \left(\frac{|000\rangle + |111\rangle}{\sqrt{2}}\right)^{\otimes 3}, \quad |\bar{1}\rangle = \left(\frac{|000\rangle - |111\rangle}{\sqrt{2}}\right)^{\otimes 3},$$

which are state vectors in the 9-qubit system $((\mathbb{C}^2)^{\otimes 3})^{\otimes 3} = (\mathbb{C}^2)^{\otimes 9}$. We can have a quantum circuit that processes 9-qubit states, and fix *bit flip* error $|0\rangle \leftrightarrow |1\rangle$ (the Pauli-X gate is applied by accident) and *phase flip* error $|+\rangle \leftrightarrow |-\rangle$ (the Pauli-Z gate is applied).

Stabilizer codes arise from a distinguished choice of quantum gates that do *not* form a universal collection. The original example is $\{\mathrm{CNOT}, H, S\}$ we saw in Section 18.2.

Once such a collection $\mathscr{C}$ is fixed, the corresponding *stabilizer circuits* are the quantum circuits we can build up only using the quantum gates in this collection. The *stabilizer states* are the states we get by applying such circuits to the initial vector $|0\cdots 0\rangle$. If the collection $\mathscr{C}$ is sufficiently small (and does not have universality), we can expect the number of k-qubit stabilizer states to be finite. Then a measurement schemes built on such finite possibilieties can have better chance of getting an expected result with high probability. Many concrete algorithms we discussed in this lecture series, from Lectures 10 (key distribution), 11 (superdense coding and teleportation), 13 (the Deutsch–Josza and the Bernstein–Vazirani algorithms) are of this type.

23.2. **Complexity theory.** Reference: [NC00, Section 1.4.5; Aar18, Section 24]

Decision problems (or rather, the algorithms to solve them) that come with control parameters in classical computation can be divided into several different complexity classes. The most important ones are:

P: solvable by a deterministic algorithm in polynomial time $\leq CN^k$ for the control parameter N (*polynomial time*);

[35] P. Shor, "Scheme for reducing decoherence in quantum computer memory". Physical Review A. **52** (4), R2493–R2496 (1995).

NP: a solution candidate can be verified by a deterministic algorithm in polynomial time for the control parameter (*nondeterministic polynomial time*);
NP-hard: the ones to which every NP problem can be reduced, in polynomial time;
NP-complete: both NP and NP-hard;
PSPACE: solvable by an algorithm using a polynomial amount of registers (memory), but possibly with exponential time.

Here are important complexity classes in quantum computation:

BQP: solvable by a quantum algorithm in polynomial time, with success probability $\geq \frac{2}{3}$;
QMA: an analogue of the NP class (or more precisely, the Merlin–Arthur class that contains NP), but instead of "problems" we consider *languages* $\mathscr{L}$ (collections of strings); $\mathscr{L}$ is in this class if we have a *verifier* circuit that can verify with high probability if a given string s is in $\mathscr{L}$ in a polynomial-step, and reject inputs $|s\rangle \otimes v$ with not-small probability if s is not in $\mathscr{L}$ and v is a k-qubit state where k is bounded by a fixed polynomial of the length of s.

For example, Shor's algorithm is in the BQP class, and there is no known P-class algorithm to solve the same problem. For each algorithm in the BQP class, we can find a PSPACE-classical algorithm that simulates it. We say that BQP is contained in PSPACE because of this.

23.3. **Quantum information theory.** Reference: [NC00, Chapters 11 and 12; Aar18, Chapter 11]

Quantum information theory is a field closely related to quantum computation; together they form a coherent body of science about information processing through quantum mechanical principles. The emphasis in quantum information is more on *communication of information*.

Quantum channels play the central role in quantum information theory. We touched on this idea in Lecture 11. For example, *capacity* of quantum channels is an important topic, and *entropy* is a closely related concept. In classical information theory, we look at *the Shannon entropy* of communication channels to understand the information capacity of a communication channel. In quantum information, the corresponding notion is *the von Neumann entropy* (that in fact predates the Shannon entropy).

In another direction, *nonlocal games* give a convenient framework for thought experiments that help us better understand the quantum effects. We touched on the CHSH game in Lecture 12, which is the first example of this theory.

Exercises

Suppose we have two matrices $A = [A_{ij}]_{i,j}$ and $B = [B_{kl}]_{k,l}$ with $i = 1, \dots, m$, $j = 1, \dots, n$, $k = 1, \dots, p$, and $l = 1, \dots, q$. Their *Kronecker product* is

$$\begin{bmatrix} A_{11}B & A_{12}B & \dots & A_{1n}B \\ \vdots & \vdots & \dots & \vdots \\ A_{m1}B & A_{m2}B & \dots & A_{mn}B \end{bmatrix}.$$

(This is a block matrix notation, so $A_{11}B$ represents a block of p rows and q columns with components $A_{11}B_{kl}$, and so on, and overall there are mp rows and nq columns.) We can think of this as a concrete matrix representation of the tensor product $A \otimes B$. For vectors $v \in \mathbb{C}^n$ and $w \in \mathbb{C}^q$, we think of them as $n \times 1$-matrix and $q \times 1$-matrix, and use the Kronecker product to represent $v \otimes w$.

Exercise 1. In the above setting, check $(A \otimes B)(v \otimes w) = Av \otimes Bw$. You can start with $m = n = p = q = 2$ to get the feeling.

Exercise 2 ([NC00, Exercise 2.26]). Consider $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. Write $|{+}{+}\rangle = |+\rangle \otimes |+\rangle$ as a linear combination of vectors $|ij\rangle = |i\rangle \otimes |j\rangle$ with $i, j = 0, 1$. Next, represent $|{+}{+}\rangle$ by the Kronecker product of column vectors.

Consider the *Hadamard gate* operator on $\mathbb{C}^2$, given as

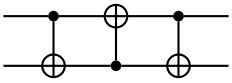
$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

Exercise 3 ([NC00, Exercise 2.51]). Check that H is a unitary matrix. Figure out how H transforms the vectors $|0\rangle, |1\rangle, |+\rangle, |-\rangle, |i\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle)$, and $|-i\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle)$ (i.e., the last vector is the eigenvector of the Pauli Y matrix for eigenvalue $-i$, and not the tensor product vector $|-\rangle \otimes |i\rangle$).

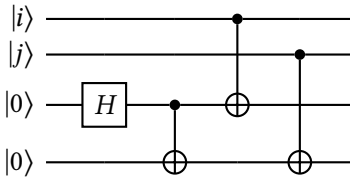
Exercise 4. What kind of rotation does H define on the Bloch sphere?

Exercise 5. Relate the axis of rotation in the previous exercise to the eigenvectors of H.

Exercise 6. What does this two-qubit quantum circut do to separable states?



Exercise 7. Conisder the following quantum circuit, with top two input qubits $|i\rangle, |j\rangle$ in computational basis. A certain measurement scheme of the bottom two qubits lets us decide $i = j$ or not. What is such a scheme?



(Hint: the Hadamard gate and the first CNOT will create the EPR pair state.)

Exercise 8. Consider the following states we had in the superdense coding protocol:

$$w_{00} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \qquad w_{10} = (X \otimes I_2)(w_{00}), \qquad w_{01} = (Z \otimes I_2)(w_{00}), \qquad w_{11} = (ZX \otimes I_2)(w_{00}).$$

How do they look like in the Kronecker product presentation?

Exercise 9. Suppose we have a vector $w \in \mathbb{C}^2 \otimes \mathbb{C}^2$. We can consider the transform

$$T_w \colon \mathbb{C}^2 \to \mathbb{C}^2, \quad v \mapsto (v^\dagger \otimes I_2)(w).$$

(This is not a linear map for the complex coefficient, but instead a *conjugate linear* map.) Check the following:

(1) when $w = w_1 \otimes w_2$, the image of T_w, i.e., $\{T_w(v) \mid v \in \mathbb{C}^2\}$ is a one-dimensional subspace of $\mathbb{C}^2$.
(2) when $w = |00\rangle + |11\rangle$, the image of T_w is all of $\mathbb{C}^2$.

Exercise 10. Consider the Boolean function $f \colon \{0,1\}^2 \to \{0,1\}$ defined by $f(x_1x_2) = x_1 + x_2 \bmod 2$. We then have the XOR query gate U_f on the 3-qubit system defined by

$$U_f|x_1x_2y\rangle = |x_1x_2z\rangle, \quad z = \mathrm{XOR}(f(x_1x_2), y)$$

on the computational basis. Why is this a unitary transform? How about the same construction for $f(x_1x_2) = x_1 \cdot x_2$?

Hint: Compute U_f^2 to find the inverse of U_f. Can you check that $U_f = U_f^\dagger$? You can either use an abstract reasoning, or work out the 8×8-matrix representation of U_f.

Exercise 11. Given a Boolean function $f \colon \{0,1\}^k \to \{0,1\}^k$, recall the XOR query gate from Simon's algorithm:

$$U_f|xy\rangle = |xz\rangle \quad z = f(x) \oplus y$$

with the bit-wise XOR of $f(x)$ and y. Why is this a unitary transform?

In the CHSH game (Lecture 12), Alice and Bob used different measurement schemes depending on the question bits x, y. (For example, if $x = 0$, Alice would measure her part of the state in the computational basis, while if $x = 1$ she would measure it in the $(|+\rangle, |-\rangle)$-basis.)

Suppose they might choose a different set of basis for these questions, and we want to estimate their winning probability. Let's say Alice would use a basis (v, v') of $\mathbb{C}^2$ for the case $x = 0$, and she would use a basis (w, w') for the case $x = 1$. Set

$$A^0 = vv^\dagger - v'v'^\dagger, \quad A^1 = ww^\dagger - w'w'^\dagger.$$

For example, in the original CHSH strategy we would have

$$A^0 = |0\rangle\langle 0| - |1\rangle\langle 1| = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad A^1 = |+\rangle\langle +| - |-\rangle\langle -| = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

For Bob's choice of measurement schemes, we define B^0 and B^1 in the same way. Finally, let's say u is the two-qubit state they are going to use (which was the EPR pair state for the CHSH strategy).

Exercise 12. When the question is $x = 0$ and $y = 0$, check that their winning probability is

$$\left\langle u, \frac{1}{2}(A^0 \otimes B^0 + I_4)u \right\rangle.$$

Hint: A^0 has eigenvalues 1 and -1, corresponding to the cases $a = 0$ and $a = 1$ for Alice's answer. Same kind of relation holds for B^0 with Bob's answer b. Then $A^0 \otimes B^0$ has eigenvalues 1 and -1, corresponding to the different combinations of a and b. Taking $\frac{1}{2}(A^0 \otimes B^0 + I_4)$ instead of $A^0 \otimes B^0$ has the effect of changing the eigenvalues from ± 1 to $1, 0$, while keeping the same conditions about a and b.

Exercise 13. For other choices of questions x and y, find signs in

$$\left\langle u, \frac{1}{2}(\pm A^x \otimes B^y + I_4)u \right\rangle.$$

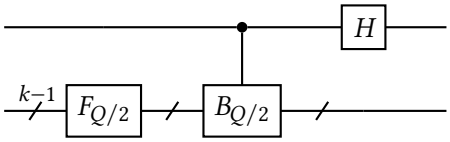
such that this would represent their winning probabilities.

Our convention of the Fourier transform matrix F_Q for $Q = 2^k$ is

$$[F_Q]_{j,l} = \frac{1}{\sqrt{Q}} \exp\left(\frac{2\pi i}{Q} j\tilde{l}\right),$$

where we modify the input index $0 \le l < Q$ to another integer $0 \le \tilde{l} < Q$ by the rule $l = 2^{k-1}x_0 + \cdots + x_{k-1}$ if and only if $\tilde{l} = x_0 + 2x_1 + \cdots + 2^{k-1}x_{k-1}$. (This means that the input binary sequence $x_0 \cdots x_{k-1}$ in the quantum gate F_Q should be interpreted as the integer $\tilde{l}$.)

Exercise 14. Recall that F_Q can be represented by the quantum circuit

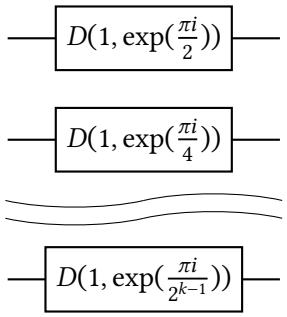


with $B_{Q/2}$ given by

$$B_{Q/2} = \begin{bmatrix} 1 & 0 & \cdots & \\ 0 & \exp\left(\frac{2\pi i}{Q}\right) & 0 & \cdots \\ \vdots & & \ddots & \\ 0 & \cdots & 0 & \exp\left((2^{k-1}-1)\frac{2\pi i}{Q}\right) \end{bmatrix}.$$

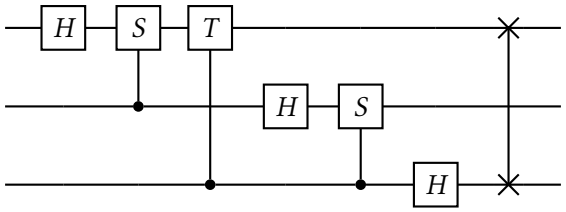
Check this for $k = 2$, i.e., check the effect of this circuit on the computational basis vectors $|x_0x_1\rangle$, and compare with the effect of F_4.

Hint: We had $F_2 = H$ for $k = 1$, the Hadamard gate. As for $B_{Q/2}$, you can use the following $(k-1)$-qubit quantum circuit representation:



see Lecture 17.

Exercise 15. We can find the following quantum circuit diagram for the 3-qubit Fourier transform in the Nielsen–Chuang book, Box 5.1.



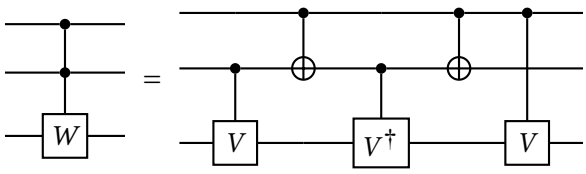
Here, the S-gate and T-gate are given by

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, \quad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}.$$

Can you relate this with the presentation from the previous exercise?

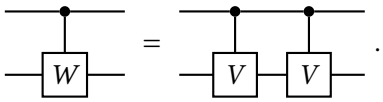
Hint: there are different conventions for the Fourier transform, like how bit-strings correspond to integers. First ignore the bit swap at the end of the above circuit.

Exercise 16. We saw that 2-bit-controlled gates can be implemented as



for a unitary matrix V satisfying $V^2 = W$. How about 3-bit-controlled gates?

Hint: for the above formula it's not important that W is a single-qubit gate, and if we have $V^2 = W$, then we also have



.

A.1. **SymPy.** SymPy is a library for symbolic mathematics. This is useful when we want to understand algebraic manipulations involved in (quantum) algorithms, like using variables to represent the components of state vectors.

[1]:
```
%%script false --no-raise-error # safeguard in case this is unintended - remove this line to install sympy
%pip install sympy
```

[2]:
```
import sympy as sy
```

Vectors can be represented as "matrices" constructed from 1-dimensional arrays, while matrices are constructed from 2-dimensional arrays

[3]:
```
vec = sy.Matrix([1, 0])  # |0⟩
Xmat = sy.Matrix([[0, 1],
                  [1, 0]])
```

The matrix product can be represented by the usual $*$ operator. This also takes care of converting 1-dim arrays to 2-dim column vectors.

[4]:
```
Xmat * vec  # expect |1⟩
```

[4]:
$$\begin{bmatrix}0\\1\end{bmatrix}$$

To create formal variables, use `symbols` command.

[5]:
```
x1, x2, x3, x4 = sy.symbols('a b c d')
test_mat = sy.Matrix([[x1, x2],
                      [x3, x4]])
test_mat
```

[5]:
$$\begin{bmatrix}a & b\\c & d\end{bmatrix}$$

[6]:
```
test_mat * vec  # product with |0⟩ would gives the first column
```

[6]:
$$\begin{bmatrix}a\\c\end{bmatrix}$$

The Kronecker product is `sympy.physics.quantum.TensorProduct`.

[7]:
```
from sympy.physics.quantum import TensorProduct
# block matrix whose blocks are I_2 or the 0 matrix
TensorProduct(Xmat, sy.eye(2))
```

[7]:
$$\begin{bmatrix}0 & 0 & 1 & 0\\0 & 0 & 0 & 1\\1 & 0 & 0 & 0\\0 & 1 & 0 & 0\end{bmatrix}$$

[8]:
```
# block matrix whose blocks are X or the 0 matrix
TensorProduct(sy.eye(2), Xmat)
```

[8]:
$$\begin{bmatrix}0 & 1 & 0 & 0\\1 & 0 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$$

A.2. **NumPy.** NumPy is an extensive scientific computing library. This is useful when we want to simulate algorithms involving some numerics, like probabilistic algorithms.

[9]:
```
import numpy as np
```

Vectors can be represented by 1-dimensional arrays, while matrices are represented by 2-dimensional arrays.

[10]:
```
vec = np.array([1, 0])  # |0⟩
Xmat = np.array([[0, 1],
                 [1, 0]])
```

The matrix product can be represented by the `@` infix operator. This also takes care of converting 1-dim arrays to 2-dim column vectors.

[11]:
```
Xmat @ vec  # expect |1⟩
```

[11]:
```
array([0, 1])
```

The Kronecker product is `np.kron`.

[12]:
```
# block matrix whose blocks are I_2 or the 0 matrix
np.kron(Xmat, np.identity(2))
```

[12]:
```
array([[0., 0., 1., 0.],
       [0., 0., 0., 1.],
       [1., 0., 0., 0.],
       [0., 1., 0., 0.]])
```

[13]:
```
# block matrix whose blocks are X or the 0 matrix
np.kron(np.identity(2), Xmat)
```

[13]:
```
array([[0., 1., 0., 0.],
       [1., 0., 0., 0.],
       [0., 0., 0., 1.],
       [0., 0., 1., 0.]])
```

SciPy. SciPy is a science computing library providing additional tools beyond NumPy.

[14]:
```
%%script false --no-raise-error # safeguard in case this is unintended - remove this line to install scipy
%pip install scipy
```

A.3. **Matplotlib.** Matplotlib is a data visualization library. You can create various forms of graphs (line plot, bar plot, scatter plot, etc., in 2-dimension or 3-dimension), and directly interact with them on Jupyter Notebook, or export as static files in PNG, EPS, etc., to include in other documents.

[15]:
```
%%script false --no-raise-error # safeguard in case this is unintended - remove this line to install matplotlib
%pip install matplotlib
```

[16]:
```
%%script false --no-raise-error # safeguard in case this is unintended - remove this line to install matplotlib
%pip install ipympl
```

To get an interactive plot inside Jupyter Notebook, install `ipympl` and load the `matplotlib` library as

```
import matplotlib.pyplot as plt
%matplotlib widget
```

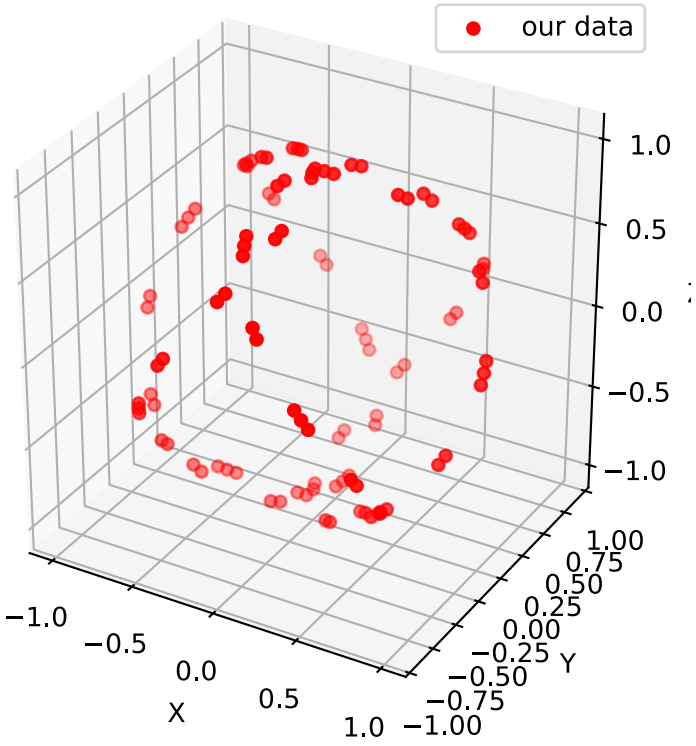
[17]:
```
# prepare matplotlib
# usually this should be at the top of notebook
import matplotlib.pyplot as plt
```

[18]:
```
# %%script false --no-raise-error # remove this for vector graphics
%config InlineBackend.figure_format = 'svg'
%matplotlib inline
```

[19]:
```
%%script false --no-raise-error # remove this for more interactive plot
%matplotlib widget
```

[20]:
```
# prepare the plot coordinates
coords = np.array([[np.cos(3 * t) * np.cos(2 * t), np.cos(3 * t) * np.
  ↪sin(2 * t), np.sin(3 * t)]
                   for t in range(0, 100)])
```

[21]:
```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(coords[:, 0], coords[:, 1], coords[:, 2], c='red', label='our␣
  ↪data')
ax.legend(loc='upper right')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_aspect('equal')
plt.show()
```



A.4. **Plotly.** Plotly is another data visualization library. It is designed to be more friendly for interactive use, and dataframe-based data.

[22]:
```
%%script false --no-raise-error # safeguard in case this is unintended - remove this line to install plotly
%pip install plotly
```

[23]:
```
# initialize plotly
import plotly.offline as pyo
import plotly.graph_objects as go
pyo.init_notebook_mode()
```

[24]:
```
# prepare the plot coordinates
coords = np.array([[np.cos(3 * t) * np.cos(2 * t), np.cos(3 * t) * np.sin(2 * t), np.sin(3 * t)]
                   for t in range(0, 100)])
```

[25]:
```
mydata = []
mydata.append(go.Scatter3d(
    x=coords[:, 0],
    y=coords[:, 1],
    z=coords[:, 2],
    mode='markers',
    name='our data',
    marker=dict(size=4,color='red')
))
```

```
fig = go.Figure(data=mydata)
fig.show()
```

A.5. **Qiskit.** Qiskit is a quantum computing library developed by IBM. It takes a bit of effort to set up, but you can get a good feeling of how a quantum computing service looks like. Besides the official documentation, the tutorial hosted at google is also a good entry point.

Edit and evaluate the next four cells to install the library.

[26]:
```
%%script false --no-raise-error # safeguard in case this is unintended -␣
  ↪remove this line to install qiskit
%pip install qiskit
```

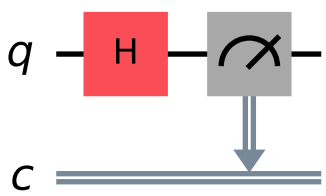
[27]:
```
%%script false --no-raise-error # safeguard in case this is unintended -␣
  ↪remove this line to install qiskit
%pip install qiskit-ibm-runtime
```

[28]:
```
%%script false --no-raise-error # safeguard in case this is unintended -␣
  ↪remove this line to install qiskit
%pip install pylatexenc
```

[29]:
```
import qiskit as qs
```

Construct a quantum circuit with one qubit input. It applies the Hadamard gate to the input qubit, then perform measurement with respect to the computational basis. The result of measurement is recorded in the "classical register" bit.

[30]:
```
# quantum circuit with one qubit and one classical bit
circ = qs.QuantumCircuit(1, 1)
circ.h(0)  # add the Hadamard gate to the qubit wire
# add measurement apparatus, telling it to record the result on the 0-th
# classical bit
circ.measure(0, 0)
# draw the circuit (cregbundle option to suppress unwanted labels)
# we have "%matplotlib widget" above, and that leads to duplicate circuit␣
  ↪plot
# to suppress it we put semicolon at the end
circ.draw("mpl", cregbundle=False);
```



To run the circuit (in a simulation), we need the following code

[31]:
```
from qiskit.providers.basic_provider import BasicProvider
from qiskit.compiler import transpile
simulator = BasicProvider().get_backend()
circ_tr = transpile(circ, simulator)
job = simulator.run(circ_tr)
result = job.result()
result.get_counts(circ)
```

[31]:
```
{'0': 501, '1': 523}
```

The default input qubit state is $|0\rangle$. So the above circuit first turns $|0\rangle$ to $|+\rangle$ by the Hadamard gate. If we measure this state in the computational basis, we expect the case 0 and the case 1 with both 50% probability.

for quantum circuits without measurements, it is also possible to get its effect as a unitary get acting on the input vector (by deault $|0\ldots0\rangle$) as follows:

[32]:
```
from qiskit.quantum_info import Statevector

circ = qs.QuantumCircuit(1)  # quantum circuit with one qubit
circ.h(0)  # add the Hadamard gate to the qubit wire

state = Statevector.from_circuit(circ)
# draw using latex
state.draw('latex')
```

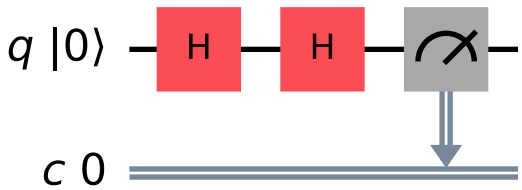
[32]:

$$\frac{\sqrt{2}}{2}|0\rangle + \frac{\sqrt{2}}{2}|1\rangle$$

If we want a different initial state v for the algorithm, we can either add some gates to transform the $|0\rangle$ state into v, or use the `initialize` command of the circuit.

Suppose we want to run the above circuit with initial state $|+\rangle$.

[33]:
```
circ = qs.QuantumCircuit(1, 1)  # same setup
circ.h(0)  # to turn the default state |0⟩ into |+⟩
# the rest is the same
circ.h(0)
circ.measure(0, 0)
circ.draw("mpl", initial_state=True, cregbundle=False);
```

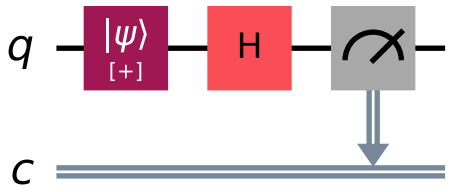


[34]:
```
# then run the circuit
circ_tr = transpile(circ, simulator)
job = simulator.run(circ_tr)
result = job.result()
result.get_counts(circ)
```

[34]:
```
{'0': 1024}
```

If we apply the Hadamard gate to $|+\rangle$, we get $|0\rangle$. So the measurement would be the case 0 for 100% of times.

[35]:
```
circ = qs.QuantumCircuit(1, 1)  # same setup
circ.initialize("+", [0])  # prepare the |+⟩ state on the 0-th qubit input
# the rest is the same
circ.h(0)
circ.measure(0, 0)
circ.draw("mpl", cregbundle=False);
```



[36]:
```
# then run the circuit
circ_tr = transpile(circ, simulator)
job = simulator.run(circ_tr)
result = job.result()
result.get_counts(circ)
```

[36]:
```
{'0': 1024}
```

Qiskit comes with a collection of standard gates that you can add to the circuit by `circ.cx(control-qubit-index,target-qubit-index)` for the CNOT gate, `circ.y(qubit-index)` for the Pauli Y gate, etc. See the document for details.

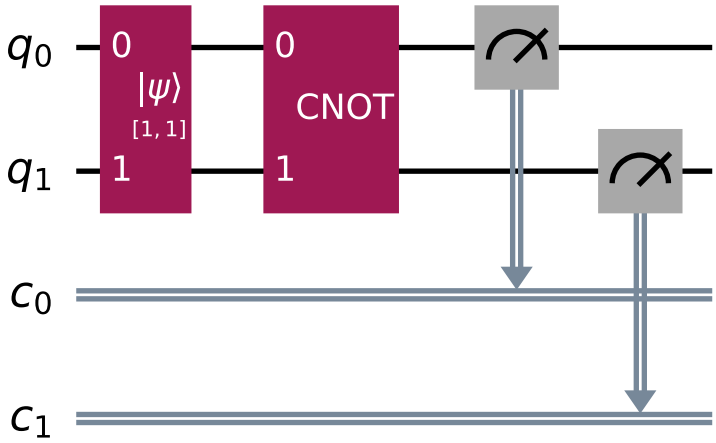
To achieve a more flexible algorithm, you want to supply a unitary matrix representing the desired quantum gate. This can be added to the circuit by the `unitary()` as described in the document.

[37]:
```
# construct the CNOT by hand
# computational basis
zeroket = np.array([1, 0])
oneket = np.array([0, 1])
zerozeroket = np.kron(zeroket, zeroket)
zerooneket = np.kron(zeroket, oneket)
onezeroket = np.kron(oneket, zeroket)
oneoneket = np.kron(oneket, oneket)
```

```
# CNOT trasforms 00 -> 00, 01 -> 01, 10 -> 11, 11 -> 10
# use atleast_2d to force 2d-array interpretation
CNOT_mat = np.atleast_2d(zerozeroket).T @ np.atleast_2d(zerozeroket)
CNOT_mat += np.atleast_2d(zerooneket).T @ np.atleast_2d(zerooneket)
CNOT_mat += np.atleast_2d(onezeroket).T @ np.atleast_2d(oneoneket)
CNOT_mat += np.atleast_2d(oneoneket).T @ np.atleast_2d(onezeroket)
CNOT_mat
```

[37]:
```
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 0, 1],
       [0, 0, 1, 0]])
```

[38]:
```
circ = qs.QuantumCircuit(2, 2)  # two qubits, two classical registers
# prepare the |11⟩ state on the 0-th and the 1-st qubits
circ.initialize("11", [0, 1])
circ.unitary(CNOT_mat, [0, 1], label="CNOT")
circ.measure(0, 0)
circ.measure(1, 1)
circ.draw("mpl", cregbundle=False);
```



[39]:
```
circ_tr = transpile(circ, simulator)
job = simulator.run(circ_tr)
result = job.result()
result.get_counts(circ)  # expect 10
```

[39]:
```
{'10': 1024}
```

Qiskit bit convention

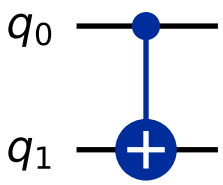
Integers: When interpreting bits as a number, bit 0 is the least significant bit, and bit $n-1$ the most significant. This is helpful when coding because each bit has the value 2^{label} (label being the qubit's index in QuantumCircuit.qubits).

Strings: When displaying or interpreting a list of bits (or qubits) as a string, bit $n-1$ is the leftmost bit, and bit 0 is the rightmost bit. This is because we usually write numbers with the most significant digit on the left, and in Qiskit, bit $n-1$ is interpreted as the most significant bit. This occasionally causes confusion when interpreting a string of bits, as you might expect the leftmost bit to be bit 0, whereas it usually represents bit $n-1$.

This means that the two-qubit states are represented by $v_1 \otimes v_0$, while v_0 still goes to the top wire in the circuit diagram.

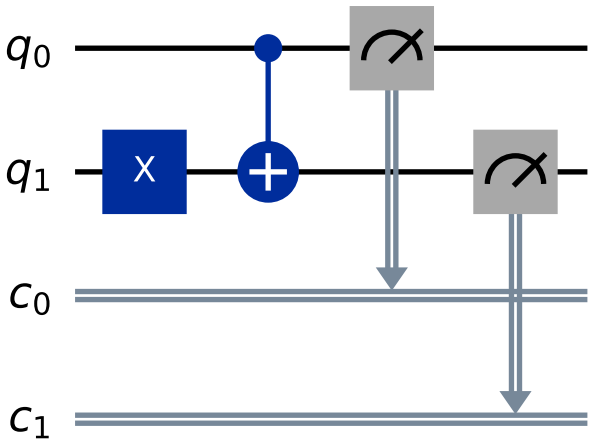
Let's check this with the effect of CNOT gate.

[40]:
```
circ = qs.QuantumCircuit(2)  # two qubits
circ.cx(0, 1)
circ.draw("mpl", cregbundle=False);
```



If we give the input $|ij\rangle = |i\rangle \otimes |j\rangle$ to this circuit, the state $|j\rangle$ will be used as the control.

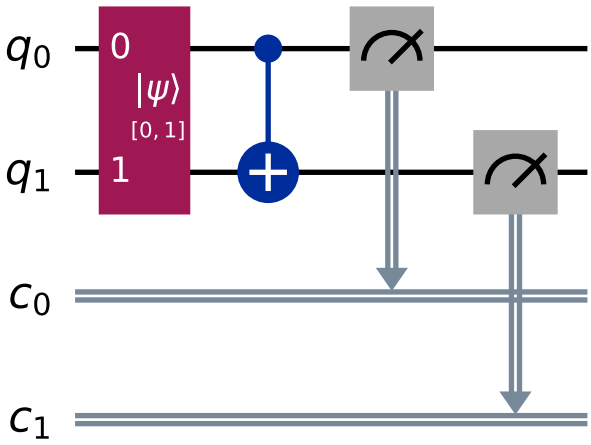
[41]:
```
circ = qs.QuantumCircuit(2, 2)  # two qubits, two classical registers
# transform |00) to |10)
circ.x(1)
# we could also use the following
# circ.initialize("10",[0,1])
# prepare the |10) state on the 0-th and the 1-st qubits
circ.cx(0, 1)
circ.measure(0, 0)
circ.measure(1, 1)
circ.draw("mpl", cregbundle=False);
```



[42]:
```
circ_tr = transpile(circ, simulator)
job = simulator.run(circ_tr)
result = job.result()
result.get_counts(circ)
# expect 10 because 0 for the control bit wouldn't change the target bit
```

[42]:
```
{'10': 1024}
```

[43]:
```
circ = qs.QuantumCircuit(2, 2)
# prepare the |01⟩ state on the 0-th and the 1-st qubits
circ.initialize("01", [0, 1])
circ.cx(0, 1)
circ.measure(0, 0)
circ.measure(1, 1)
circ.draw("mpl", cregbundle=False);
```



[44]:
```
circ_tr = transpile(circ, simulator)
job = simulator.run(circ_tr)
result = job.result()
result.get_counts(circ)
# expect 11 because 1 for the control bit would change the target bit
```

[44]:
```
{'11': 1024}
```

References

[Aar18] Scott Aaronson, *Introduction to Quantum Information Science*, lecture note (2018), available from the author's website.

[NC00] Michael A. Nielsen and Isaac L. Chuang, *Quantum computation and quantum information*, Cambridge University Press, Cambridge, 2000.

[RP11] Eleanor Rieffel and Wolfgang Polak, *Quantum computing*, Scientific and Engineering Computation, MIT Press, Cambridge, MA, 2011. A gentle introduction.